ABOUT FACE

THE ESSENTIALS OF INTERACTION DESIGN

THE COMPLETELY UPDATED CLASSIC ON CREATING DELIGHTFUL USER EXPERIENCES

Alan Cooper, Robert Reimann, David Cronin, Chris Noessel

About Face

The Essentials of Interaction Design

Fourth Edition

Alan Cooper
Robert Reimann
David Cronin
Christopher Noessel
with Jason Csizmadi
and Doug LeMoine

WILEY

For Sue, my best friend through all the adventures of life. —Alan

For Alex and Max, and for Julie. —Robert

For Jasper, Astrid, and Gretchen. —David

For Ben and Miles, for your patience and inspiration. —Christopher

And for all the designers and engineers in our industry who are helping to imagine and build a better future.

ABOUT THE AUTHORS

Alan Cooper has been a pioneer in the software world for more than 40 years, and he continues to influence a new generation of developers, entrepreneurs, and user experience professionals.

Alan started his first company in 1976 and created what has been called "the first serious business software for microcomputers." In 1988, he invented a dynamically extensible visual programming tool and sold it to Bill Gates, who released it to the world as Visual Basic. This accomplishment earned Alan the sobriquet "The Father of Visual Basic."

In 1992, Alan and his wife, Sue, cofounded the first interaction design consulting firm, Cooper. By 1997, Cooper had developed a set of core design methods now used across the industry. Personas, which Alan invented and then popularized in his two best-selling books, *About Face* and *The Inmates Are Running the Asylum*, are employed almost universally by user experience practitioners.

Today, Alan continues to advocate for more humane technology from his farm in the rolling hills north of San Francisco.

Robert Reimann has spent over 20 years pushing the boundaries of digital products as a designer, writer, strategist, and consultant. He has led dozens of desktop, mobile, web, and embedded design projects in consumer, business, scientific, and professional domains for both startups and Fortune 500 companies alike.

One of the first designers at Cooper, Robert led the development and refinement of many of the Goal-Directed Design methods described in this book. In 2005, he became the founding president of IxDA, the Interaction Design Association (www.ixda.org). He has led user experience teams at Cooper, Bose, frog, and Sonos, and is currently Principal Interaction Designer at PatientsLikeMe.

David Cronin is a Design Director at GE and a member of GE's Design and Experience studio leadership team. Prior to that, he was Director of Interaction Design at Smart Design's San Francisco studio and a former Managing Director of Interaction Design at Cooper.

David has helped design products to serve the needs of surgeons, museum visitors, investment portfolio managers, nurses, drivers, dentists, financial analysts, radiologists, field engineers, manufacturing planners, marketers, videographers, and people with chronic diseases. During his time at Cooper, he contributed substantially to the principles, patterns, and practices of Goal-Directed Design.

Christopher Noessel designs products, services, and strategy for health, financial, and consumer domains as Cooper's first Design Fellow. He has helped visualize the future of counterterrorism, built prototypes of new technologies for Microsoft, and designed telehealth devices to accommodate the crazy facts of modern health care.

Prior to working at Cooper, Chris cofounded a small interaction design agency, where he developed exhibitions and environments for museums. He was also Director of Information Design at marchFIRST, where he helped establish the Interaction Design Center of Excellence. In 2012, he coauthored *Make It So: Interaction Design Lessons from Science Fiction*. He publishes regularly to the Cooper Journal and continues to speak and teach around the world.

CREDITS

Acquisitions Editor
Mary James

Project Editor
Adaobi Obi Tulton

Technical Editor
Christopher Noessel

Senior Production Editor
Kathleen Wisor

Copy Editor
Gayle Johnson

**Manager of Content Development
and Assembly**
Mary Beth Wakefield

Director of Community Marketing
David Mayhew

Marketing Manager
Carrie Sherrill

Business Manager
Amy Knies

**Vice President and Executive Group
Publisher**
Richard Swadley

Associate Publisher
Jim Minatel

Project Coordinator, Cover
Todd Klemme

Compositor
Maureen Forys, Happenstance
Type-O-Rama

Proofreader
Nancy Carrasco

Indexer
Johnna VanHoose Dinse

Cover Designer
Jason Csizmadi

Art Director
Jason Csizmadi

ACKNOWLEDGMENTS

CONTENTS

FOREWORD

I began working on the first edition of this book 20 years ago. Fittingly, I wrote a manifesto—a challenge to frustrated practitioners to step forward and begin creating software that users would love. In those days, few designers and precious little software didn't make your head hurt to use. Strong measures were called for.

Today, the technology landscape is much different; consequently, this fourth edition is also much different. In 1994 the state of the art of personal software was an address book or spreadsheet. Today, the digitization of all forms of media has put consumers neck-deep in technology. Powerful handheld apps are now in the hands of amateurs and non-technical users—apps for music listening and production; for photography, video, news, and communications; for home security and environmental control; for health, fitness, and personal tracking; for games and education; and for shopping.

Over a billion people have a full-fledged computer in their pocket and access to millions of applications and websites. The value of making these user-facing products easier to understand and use is clear. We interaction designers have earned our seat at the table and are well established as an integral part of teams that produce successful, widely used digital products.

The primary challenge of the first two decades of interaction design practice was to invent the process, tools, roles, and methods needed to succeed. Now that we have demonstrated our success, our relationship to others in our organization is changing. Each of these best practices is now evolving as we integrate our skills more deeply into our teams. Specifically, we need to work more effectively with business people and developers.

Twenty years ago, developers too had to fight for inclusion and relevance. Although firmly embedded in the corporate hierarchy, they lacked credibility and authority. As

consumer digitization increased, developers grew dissatisfied as the agents of users' misery. They knew they could do better.

The agile movement and, more recently, the growth of lean practices are each efforts by software developers to have more influence on their own destiny. Developers were just as frustrated as designers at the sorry state of digital interaction, and they wanted improvements. They realized that the software construction process had been modeled after industrial archetypes that didn't suit the new digital medium.

A few brave developers began experimenting with unorthodox methods of creating software in smaller increments while maintaining closer contact with their clients. They wanted to avoid lengthy development efforts—"death marches"—that resulted in unhappy users. They also were motivated by a natural desire to find a process that more reliably resulted in better products that they could be proud of.

Although each variant has its adherents and detractors, the course of software development has been forever altered by these new approaches. The notion that the old ways weren't working well is now widely accepted, and the quest for new methods continues.

This new self-awareness in the development community is a huge opportunity for interaction designers. Before, developers saw designers as competing for scarce resources. Now, developers see interaction designers as useful helpers, able to contribute skills, experience, and viewpoints that developers cannot. As developers and designers have begun to cooperate instead of competing, they have discovered that their powers are multiplied by working side by side.

Every practitioner—developer as well as designer—wants to create a product they can be proud of. To improve outcomes, both groups have been rethinking the entire development process, demanding better tools, better guidance, and better access. Historically, though, developers and interaction designers have pursued their common goal separately, developing tools and processes that work in separate silos. The two practices are quite different in many ways, and neither will work in subservience to the other. The challenge, then, is to learn how they can work together, effectively, successfully, in mutual support.

At the most forward-looking companies, you can already see this happening: Developers and designers sit next to each other, working cooperatively and collaboratively. When designers and developers—and the many other practitioners working with them—collaborate fully, the result is far better than any other method we've tried. The speed with which the work gets done is much greater and the quality of the end product much higher. And users are more delighted.

On the business side, executives often misunderstand the role of interaction design. It sometimes seems that the only place where it is truly understood is in tiny start-ups. Although larger companies may have many interaction designers on staff, managers persistently fail to incorporate their design expertise into the process until it's too late.

All the design skill and process in the world won't succeed unless the corporate culture supports interaction design and its objectives. Apple isn't a paragon of user experience because of its employees' design skills, but because Steve Jobs, its former (and legendarily autocratic) leader, was a tireless advocate of the power of design.

Few companies have leaders as bold as Jobs. Those that do tend to be small start-ups. You will find it difficult to convince business people of the value of collaborative design work. But each year will see more success stories—more proof of the value of this new work paradigm. I remember when Apple and Microsoft, not to mention Google and Facebook, were tiny start-ups with many doubters.

The two opportunities that face interaction designers today are finding, or creating, advocates on the business side, and learning how to collaborate with the newly sympathetic development community.

What is indisputable is the awesome power of interaction design: giving technology users a memorable, effective, easy, and rewarding experience as they work, play, and communicate.

—Alan Cooper

INTRODUCTION TO THE FOURTH EDITION

This book is about **interaction design**—the practice of designing interactive digital products, environments, systems, and services. Like most design disciplines, interaction design is concerned with form. However, first and foremost, interaction design focuses on something that traditional design disciplines do not often explore: the design of *behavior.*

Most design *affects* human behavior: Architecture is concerned with how people use physical space, and graphic design often attempts to motivate or facilitate a response. But now, with the ubiquity of silicon-enabled products—from computers to cars and phones to appliances—we routinely create products that *exhibit* complex behavior.

Take a product as basic as an oven. Before the digital age, it was quite simple to operate an oven: You simply turned a single knob to the correct position. There was one position for off, and each point along which the knob could turn resulted in a unique temperature. Every time the knob was turned to a given position, *the exact same thing happened.* You could call this a "behavior," but it is certainly a simple one.

Compare this to modern ovens with microprocessors, LCD screens, and embedded operating systems. They are endowed with buttons labeled with non-cooking-related terms such as Start, Cancel, and Program, as well as the perhaps more expected Bake and Broil. What happens when you press any one of these buttons is much less predictable than what happened when you turned the knob on your old gas range. In fact, the outcome of pressing one of these buttons often depends on the oven's operational state, as well as the sequence of buttons you press before pressing the last one. This is what we mean by *complex behavior.*

This emergence of products with complex behavior has given rise to a new discipline. Interaction design borrows theory and technique from traditional design, usability, and engineering disciplines. But it is greater than the sum of its parts, with its own unique

methods and practices. And to be clear, interaction design is very much a *design* discipline, quite different from science and engineering. Although it employs an analytical approach when required, interaction design is also very much about synthesis and imagining things as they might be, not necessarily as they currently are.

Interaction design is an inherently humanistic enterprise. It is concerned most significantly with satisfying the needs and desires of the people who will interact with a product or service. These goals and needs can best be understood as *narratives*—logical and emotional progressions over time. In response to these user narratives, digital products must express behavioral narratives of their own, appropriately responding not only at the levels of logic and data entry and presentation, but also at a more human level.

In this book we describe a particular approach to interaction design that we call the Goal-Directed Design method. We've found that when designers focus on people's goals—the reasons why they use a product in the first place—as well as their expectations, attitudes, and aptitudes, they can devise solutions that people find both powerful and pleasurable to use.

As even the most casual observer of developments in technology must have noticed, interactive products quickly can become complex. Although a mechanical device may be capable of a dozen visible states, a digital product may be capable of *thousands* of different states (if not more!). This complexity can be a nightmare for users and designers alike. To tame this complexity, we rely on a systematic and rational approach. This doesn't mean that we don't also value and encourage inventiveness and creativity. On the contrary, we find that a methodical approach helps us clearly identify opportunities for revolutionary thinking and provides a way to assess the effectiveness of our ideas.

According to Gestalt Theory, people perceive a thing not as a set of individual features and attributes, but as a unified whole in a relationship with its surroundings. As a result, it is impossible to effectively design an interactive product by decomposing it into a list of atomic requirements and coming up with a design solution for each. Even a relatively simple product must be considered in totality and in light of its context in the world. Again, we've found that a methodical approach helps provide the holistic perspective necessary to create products that people find useful and engaging.

A Brief History of Interaction Design

In the late 1970s and early 1980s, a dedicated and visionary set of researchers, engineers, and designers in the San Francisco Bay area were busy inventing how people would interact with computers in the future. At Xerox Parc, SRI, and eventually Apple Computer, people had begun discussing what it meant to create useful and usable "human interfaces" to digital products. In the mid-1980s, two industrial designers, Bill Moggridge

and Bill Verplank, were working on the first laptop computer, the GRiD Compass. They coined the term *interaction design* for what they were doing. But it would be another 10 years before other designers rediscovered this term and brought it into mainstream use.

When *About Face* was first published in August 1995, the landscape of interaction design was still a frontier wilderness. A small cadre of people brave enough to hold the title user interface designer operated in the shadow of software engineering, rather like the tiny, quick-witted mammals that lurked in the shadows of hulking tyrannosaurs. "Software design," as the first edition of *About Face* called it, was poorly understood and underappreciated. When it was practiced at all, it was usually practiced by developers. A handful of uneasy technical writers, trainers, and product support people, along with a rising number of practitioners from another nascent field—usability—realized that something needed to change.

The amazing growth and popularity of the web drove that change, seemingly overnight. Suddenly, the phrase "ease of use" was on everyone's lips. Traditional design professionals, who had dabbled in digital product design during the short-lived popularity of "multimedia" in the early '90s, leapt to the web en masse. Seemingly new design titles sprang up like weeds: information designer, information architect, user experience strategist, interaction designer. For the first time, C-level executive positions were established to focus on creating user-centered products and services, such as chief experience officer. Universities scrambled to offer programs to train designers in these disciplines. Meanwhile, usability and human-factors practitioners also rose in stature and are now recognized as advocates for better-designed products.

Although the web knocked back interaction design idioms by more than a decade, it inarguably placed user requirements on the radar of the corporate world permanently. After the second edition of *About Face* was published in 2003, the *user experience* of digital products became front-page news in the likes of *Time* and *BusinessWeek*. And institutions such as Harvard Business School and Stanford recognized the need to train the next generation of MBAs and technologists to incorporate design thinking into their business and development plans. People are tired of new technology for its own sake. Consumers are sending a clear message that they want *good* technology: technology that is *designed* to provide a compelling and effective user experience.

In August 2003, five months after the second edition of *About Face* proclaimed the existence of a new design discipline called *interaction design*, Bruce "Tog" Tognazzini made an impassioned plea to the nascent community to create a nonprofit professional organization. A mailing list and steering committee were founded shortly thereafter by Challis Hodge, David Malouf, Rick Cecil, and Jim Jarrett.

In September 2005, IxDA, the Interaction Design Association (www.ixda.org), was incorporated. In February 2008, less than a year after the publication of the third edition of *About*

Face, IxDA hosted its first international design conference, Interaction08, in Savannah, Georgia. In 2012, IxDA presented its first annual Interaction Awards for outstanding designs submitted from all over the world. IxDA currently has over 70,000 members living in more than 20 countries. We're pleased to say that interaction design has truly come into its own as both a design discipline and a profession.

IxD and User Experience

The first edition of *About Face* described a discipline called software design and equated it with another discipline called user interface design. Of these two terms, user interface design has enjoyed more longevity. We still use it occasionally in this book, specifically to connote the layout of widgets on a screen. However, this book discusses a discipline broader than the design of user interfaces. In the world of digital technology, form, function, content, and behavior are so inextricably linked that many of the challenges of designing an interactive product go right to the heart of what a digital product *is* and *does*.

As we've discussed, interaction designers have borrowed practices from more-established design disciplines but also have evolved beyond them. Industrial designers have attempted to address the design of digital products. But like their counterparts in graphic design, their focus traditionally has been on the design of static form, not the design of interactivity, or form that changes and reacts to input over time. These disciplines do not have a language with which to discuss the design of rich, dynamic behavior and changing user interfaces.

A term that has gained particular popularity in the last decade is *user experience (UX) design*. Many people have advocated for the use of this term as an umbrella under which several different design and usability disciplines collaborate to create products, systems, and services. This is a laudable goal with great appeal, but it does not in itself directly address the core concern of interaction design as discussed in this book: how to specifically *design the behavior* of complex interactive systems. It's useful to consider the similarities and synergies between creating a customer experience at a physical store and creating one with an interactive product. However, we believe specific methods are appropriate for designing for the world of bits.

We also wonder whether it is truly possible to *design* an experience. Designers of all stripes hope to manage and *influence* people's experiences, but this is done by carefully manipulating the variables intrinsic to the medium at hand. A graphic designer creating a poster arranges fonts, photos, and illustrations to help create an experience; a furniture designer working on a chair uses materials and construction techniques to help create an experience; an interior designer uses layout, lighting, materials, and even sound to help create an experience.

Extending this thinking into the world of digital products, we find it useful to think that we influence people's experiences by designing the mechanisms for interacting with a product. Therefore, we have chosen Moggridge's term *interaction design* (now abbreviated by many in the industry as IxD) to denote the kind of design this book describes.

Of course, often a design project requires careful attention to the orchestration of a number of design disciplines to achieve an appropriate user experience, as shown in Figure 1. It is in these situations that we feel the term *user experience design* is most applicable.

Form
Industrial designers
Graphic designers

Content
Information architects
Copywriters
Animators
Sound designers

Behavior
Interaction designers
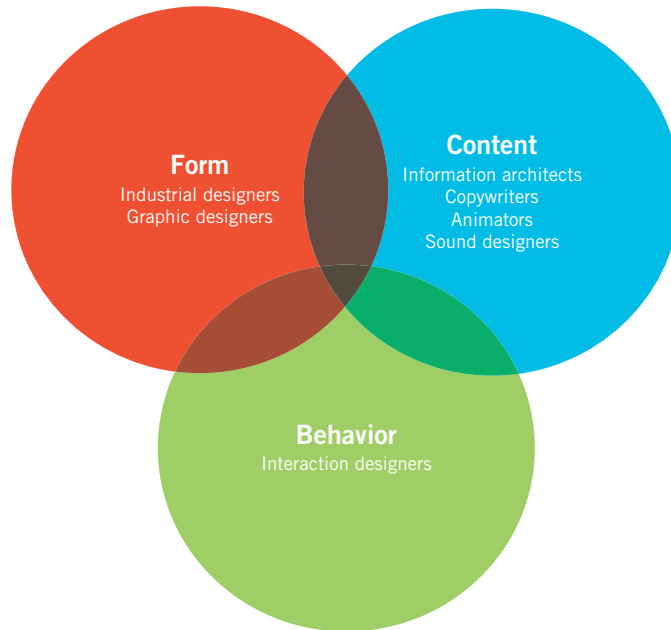
Figure 1: User experience (UX) design has three overlapping concerns: form, behavior, and content. Interaction design focuses on the design of behavior but also is concerned with how that behavior relates to form and content. Similarly, information architecture focuses on the structure of content but also is concerned with behaviors that provide access to content and how the content is presented to the user. Industrial design and graphic design are concerned with the form of products and services but also must ensure that their form supports use, which requires attention to behavior and content.

What This Book Is and What It Is Not

In this book, we attempt to give you effective and practical tools for interaction design. These tools consist of *principles*, *patterns*, and *processes*. Design *principles* encompass broad ideas about the practice of design, as well as rules and hints about how to best use specific user interface and interaction design idioms. Design *patterns* describe sets

of interaction design idioms that are common ways to address specific user requirements and design concerns. Design *processes* describe how to understand and define user requirements, how to then translate those requirements into the framework of a design, and finally how to best apply design principles and patterns to specific contexts.

Although books are available that discuss design principles and design patterns, few books discuss design processes, and even fewer discuss all three of these tools and how they work together to create effective designs. Our goal was to create a book that unites all three of these tools. While helping you design more effective and useful dialog boxes and menus, this book also helps you understand how users comprehend and interact with your digital product. In addition, it helps you understand how to use this knowledge to drive your design.

Integrating design principles, processes, and patterns is the key to designing effective product interactions and interfaces. There is no such thing as an objectively good user interface. Quality depends on the context: who the user is, what she is doing, and what her motivations are. Applying a set of one-size-fits-all principles makes user interface creation *easier*, but it doesn't necessarily make the end result *better*. If you want to create good design solutions, there is no avoiding the hard work of really understanding the people who will actually interact with your product. Only then is it useful to have at your command a toolbox of principles and patterns to apply in specific situations. We hope this book will both encourage you to deepen your understanding of your product's users and teach you how to translate that understanding into superior product designs.

This book does *not* attempt to present a style guide or set of interface standards. In fact, you'll learn in Chapter 17 about the limitations of such tools. That said, we hope that the process and principles described in this book are compatible companions to the style guide of your choice. Style guides are good at answering *what* but generally are weak at answering *why*. This book attempts to address these unanswered questions.

This book discusses four main steps to designing interactive systems: researching the domain, understanding the users and their requirements, defining a solution's framework, and filling in the design details. Many practitioners would add a fifth step: *validation*—testing a solution's effectiveness with users. This is part of a discipline widely known as *usability*.

Although this is an important and worthwhile component to many interaction design initiatives, it is a discipline and practice in its own right. We briefly discuss design validation and usability testing in Chapter 5. We also urge you to refer to the significant and ever-growing body of usability literature for more detailed information about conducting and analyzing usability tests.

How This Book Is Structured

This book is organized in a way that presents its ideas in an easy-to-use reference structure. The book is divided into three parts:

- Part I introduces and describes the Goal-Directed Design process in detail, as well as building design teams and integrating with project teams.
- Part II deals with high-level interaction design principles that can be applied to any interaction design problem on almost any platform.
- Part III covers lower-level and platform-specific interface design principles and idioms for mobile, desktop, the web, and more.

Changes Since the Third Edition

In June 2007, just two months after the third edition of *About Face* was published, Apple changed the digital landscape forever with the introduction of the iPhone and iOS. In 2010 Apple followed with the first commercially successful tablet computer, the iPad. These touch-based, sensor-laden products and the competitors that followed in their footsteps have added an entirely new lexicon of idioms and design patterns to the language of interaction. This fourth edition of *About Face* addresses these and other modern interaction idioms directly.

This new edition retains what still holds true, updates things that have changed, and provides new material reflecting how the industry has changed in the last seven years. It also addresses new concepts we have developed in our practices to address the changing times.

Here are some highlights of the major changes you will find in this edition of *About Face*:

- The book has been reorganized and streamlined to present its ideas in a more concise and easy-to-use structure and sequence. Some chapters have been rearranged for better flow, others have been merged, a few have been condensed, and several new chapters have been added.
- Terminology and examples have been updated to reflect the current state of the art in the industry. The text as a whole has been thoroughly edited to improve clarity and readability.
- Part I describes the Goal-Directed Design process in additional detail and more accurately reflects the most current practices at Cooper. It also includes additional information on building a design team and integrating with development and project teams.
- Part II has been significantly reorganized to more clearly present its concepts and principles, and includes newly updated information on integrating visual design.

Part III has been extensively rewritten, updated, and extended to reflect new mobile and touch-based platforms and interaction idioms. It also offers more detailed coverage of web interactions and interactions on other types of devices and systems. We hope you will find that these additions and changes make *About Face* a more relevant and useful reference than ever before.

Examples Used in This Book

This book is about designing all kinds of interactive digital products. However, because interaction design has its roots in software for desktop computers, and the vast majority of today's PCs run Microsoft Windows, there is certainly a bias in the focus of our discussions of desktop software. Similarly, the first focus of many developers of native mobile apps is iOS, so the bulk of our mobile examples are from this platform.

Having said that, most of the material in this book transcends platform. It is equally applicable across platforms—Mac OS, Windows, iOS, Android, and others. The majority of the material is relevant even for more divergent platforms such as kiosks, embedded systems, 10-foot interfaces, and the like.

A number of the desktop examples in this book are from the Microsoft Office suite and from Adobe Photoshop and Illustrator. We have tried to stick with examples from these mainstream applications for two reasons. First, you're likely to be at least somewhat familiar with the examples. Second, it's important to show that the user interface design of even the most finely honed products can be significantly improved with a goal-directed approach. This edition also contains many new examples from mobile apps and the web, as well as several more-exotic applications.

A few examples in this new edition come from now-moribund software or OS versions. These examples illustrate particular points that we felt were useful enough to retain in this edition. The vast majority of examples are from contemporary software and OS releases.

Who Should Read This Book

While the subject matter is broadly aimed at students and practitioners of interaction design, anyone concerned about users interacting with digital technology will gain insights from reading this book. Developers, designers of all stripes involved with digital product design, usability professionals, and project managers will all find something useful in this book. If you've read earlier editions of *About Face* or *The Inmates*

Are Running the Asylum (Sams, 2004), you will find new and updated information about design methods and principles here.

We hope this book informs you and intrigues you. Most of all, we hope it makes you think about the design of digital products in new ways. The practice of interaction design is constantly evolving, and it is new and varied enough to generate a wide variety of opinions on the subject. If you have an interesting opinion, or if you just want to talk, we'd be happy to hear from you. E-mail us at `alan@cooper.com`, `rmreimann@gmail.com`, `davcron@gmail.com`, or `chrisnoessel@gmail.com`.

Goal-Directed Design

A DESIGN PROCESS FOR DIGITAL PRODUCTS

This book has a simple premise: If we design and develop digital products in such a way that the people who use them can easily achieve their goals, they will be satisfied, effective, and happy. They will gladly pay for our products—and recommend that others do the same. Assuming that we can do so in a cost-effective manner, this will translate into business success.

On the surface, this premise seems obvious: Make people happy, and your products will be a success. Why, then, are so many digital products so difficult and unpleasant to use? Why aren't we all happy and successful when we use them? Why, despite the steady march of faster, cheaper, and more accessible technology, are we still so often frustrated?

The answer, in short, is the *absence of design* as a fundamental and equal part of the product planning and development process.

Design, according to industrial designer Victor Papanek, is *the conscious and intuitive effort to impose meaningful order*. We propose a somewhat more detailed definition of human-oriented design activities:

- Understanding the desires, needs, motivations, and contexts of people using products
- Understanding business, technical, and domain opportunities, requirements, and constraints

- Using this knowledge as a foundation for plans to create products whose form, content, and behavior are useful, usable, and desirable, as well as economically viable and technically feasible

This definition is useful for many design disciplines, although the precise focus on form, content, and behavior varies depending on what is being designed. For example, an informational website may require particular attention to *content*, whereas the design of a simple TV remote control may be concerned primarily with *form*. As discussed in the Introduction, interactive digital products are uniquely imbued with complex *behavior*.

When performed using the appropriate methods, design can, and does, provide the missing human connection in technological products. But most current approaches to the design of digital products don't work as advertised.

The Consequences of Poor Product Behavior

In the nearly 20 years since the publication of the first edition of *About Face*, software and interactive digital products have greatly improved. Many companies have begun to focus on serving people's needs with their products and are spending the time and money needed to support the design process. However, many more still fail to do so—at their peril. As long as businesses continue to focus solely on *technology* and *market data* while shortchanging design, they will continue to create the kind of products we've all grown to despise.

The following sections describe a few of the consequences of creating products that lack appropriate design and thus ignore users' needs and desires. How many of your digital products exhibit some of these characteristics?

Digital products are rude

Digital products often blame users for making mistakes that are not their fault, or should not be. Error messages like the one shown in Figure 1-1 pop up like weeds, announcing that the user has failed yet again. These messages also demand that the user acknowledge his failure by confirming it: OK.

Digital products and software frequently interrogate users, peppering them with a string of terse questions that they are neither inclined nor prepared to answer: "Where did you hide that file?" Patronizing questions like "Are you sure?" and "Did you really want to delete that file, or did you have some other reason for pressing the Delete key?" are equally irritating and demeaning.
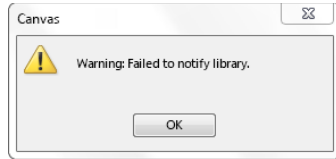
Figure 1-1: Thanks for sharing. Why didn't the application notify the library? Why did it want to notify the library? Why is it telling us? And what are we OKing, anyway? It is not OK that the application failed!

Our software-enabled products also fail to act with a basic level of decency. They forget information we tell them and don't do a very good job of anticipating our needs. Even the iPhone—generally the baseline for good user experience on a digital device—doesn't anticipate that someone might not want to be pestered with a random phone call when he is in the middle of a business meeting *that is sitting right there in the iPhone's own calendar*. Why can't it quietly put a call that isn't from a family member into voicemail?

Digital products require people to think like computers

Digital products regularly assume that people are technology literate. For example, in Microsoft Word, if a user wants to rename a document she is editing, she must know that she must either close the document or use the "Save As..." menu command (and remember to delete the file with the old name). These behaviors are inconsistent with how a normal person thinks about renaming something; rather, they require that a person change her thinking to be more like the way a computer works.

Digital products are also often obscure, hiding meaning, intentions, and actions from users. Applications often express themselves in incomprehensible jargon that cannot be fathomed by normal users ("What is your SSID?") and are sometimes incomprehensible even to experts ("Please specify IRQ.").

Digital products have sloppy habits

If a 10-year-old boy behaved like some software apps or devices, he'd be sent to his room without supper. These products forget to shut the refrigerator door, leave their shoes in the middle of the floor, and can't remember what you told them only five minutes earlier. For example, if you save a Microsoft Word document, print it, and then try to close it, the application again asks you if you want to save it! Evidently the act of printing caused the application to think the document had changed, even though it did not. Sorry, Mom, I didn't hear you.

Software often requires us to step out of the main flow of tasks to perform functions that shouldn't require separate interfaces and extra navigation to access. Dangerous commands, however, are often presented right up front where users can accidentally trigger them. Dropbox, for example, sandwiches Delete between Download and Rename on its

context menus, practically inviting people to lose the work they've uploaded to the cloud for safekeeping.

Furthermore, the appearance of software—especially business and technical applications—can be complex and confusing, making navigation and comprehension unnecessarily difficult.

Digital products require humans to do the heavy lifting

Computers and their silicon-enabled brethren are purported to be labor-saving devices. But every time we go out into the field to watch real people doing their jobs with the assistance of technology, we are struck by how much work they are forced to do simply to manage the proper operation of software. This work can be anything from manually copying (or, worse, retyping) values from one window into another, to attempting (often futilely) to paste data between applications that otherwise don't speak to each other, to the ubiquitous clicking and pushing and pulling of windows and widgets around the screen to access hidden functionality that people use every day to do their job.

The evidence is everywhere that digital products have a lot of explaining to do when it comes to their poor behavior.

Why Digital Products Fail

Most digital products emerge from the development process like a sci-fi monster emerging from a bubbling tank. Instead of planning and executing with a focus on satisfying the needs of the people who use their products, companies end up creating solutions that—while technically advanced—are difficult to use and control. Like mad scientists, they fail because they have not imbued their creations with sufficient humanity.

Why is this? What is it about the technology industry as a whole that makes it so inept at designing the interactive parts of digital products? What is so broken about the current process of creating software-enabled products that it results in such a mess?

There are four main reasons why this is the case:

- **Misplaced priorities** on the part of both product management and development teams
- **Ignorance about real users** of the product and what their baseline needs are for success
- **Conflicts of interest** when development teams are charged with both designing and building the user experience
- **Lack of a design process** that permits knowledge about user needs to be gathered, analyzed, and used to drive the development of the end experience

Misplaced priorities

Digital products come into the world subject to the push and pull of two often-opposing camps—marketers and developers. While marketers are adept at understanding and quantifying a marketplace opportunity, and at introducing and positioning a product within that market, their input into the product design process is often limited to lists of requirements. These requirements often have little to do with what users actually *need* or *desire* and have more to do with chasing the competition, managing IT resources with to-do lists, and making guesses based on market surveys—what people say they'll *buy*. (Contrary to what you might suspect, few users can clearly articulate their needs. When asked direct questions about the products they use, most tend to focus on low-level tasks or workarounds to product flaws. And, what they think they'll buy doesn't tell you much about how—or if—they will use it.)

Unfortunately, reducing an interactive product to a list of a hundred features doesn't lend itself to the kind of graceful orchestration that is required to make complex technology useful. Adding "easy to use" as a checklist item does nothing to improve the situation.

Developers, on the other hand, often have no shortage of input into the product's final form and behavior. Because they are in charge of construction, they decide exactly what gets built. And they too have a different set of imperatives than the product's eventual audience. Good developers are focused on solving challenging technical problems, following good engineering practices, and meeting deadlines. They often are given incomplete, myopic, confusing, and sometimes contradictory instructions and are forced to make significant decisions about the user experience with little time or knowledge of how people will actually use their creations.

Thus, the people who are most often responsible for creating our digital products rarely take into account the users' *goals*, needs, or motivations. At the same time, they tend to be highly reactive to market trends and technical constraints. This can't help but result in products that lack a coherent user experience. We'll soon see why goals are so important in addressing this issue.

The results of poor product vision are, unfortunately, digital products that irritate rather than please, reduce rather than increase productivity, and fail to meet user needs. Figure 1-2 shows the evolution of the development process and where, if at all, design has historically fit in. Most of digital product development is stuck in the first, second, or third step of this evolution, where design either plays no real role or becomes a surface-level patch on shoddy interactions—"lipstick on the pig," as one of our clients called it. The core activities in the design process, as we will soon discuss, should *precede* coding and testing to ensure that products truly meet users' needs.
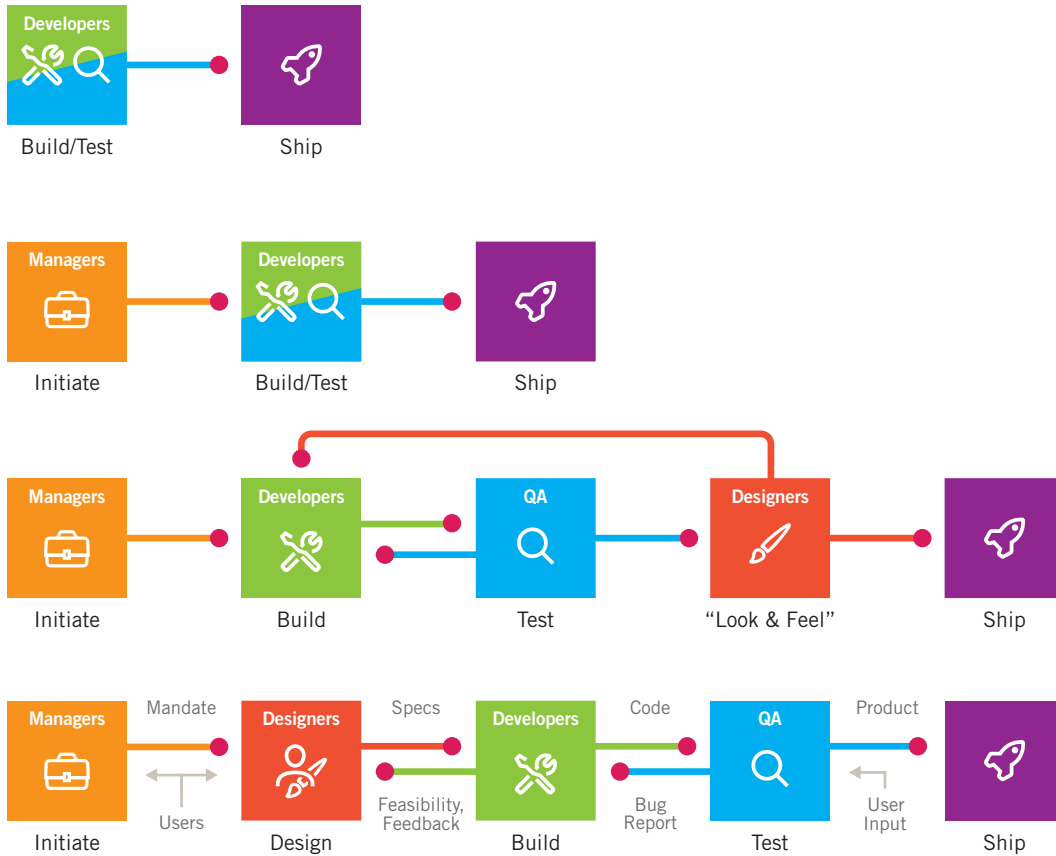
Figure 1-2: The evolution of the software development process. The first diagram depicts the early days of the software industry, when smart developers dreamed up products and then built and tested them. Inevitably, professional managers were brought in to help facilitate the process by translating market opportunities into product requirements. As depicted in the third diagram, the industry matured, and testing became a discipline in its own right. With the popularization of the graphical user interface (GUI), graphic designers were brought in to create icons and other visual elements. The final diagram shows the Goal-Directed approach to software development, where decisions about a product's capabilities, form, and behavior are made before the expensive and challenging construction phase.

Ignorance about real users

It's an unfortunate truth that the digital technology industry doesn't have a good understanding of what it takes to make users happy. In fact, most technology products get built without much understanding of users. We might know what *market segment* our users are in, how much money they make, how they like to spend their weekends, and what sorts of cars they buy. We might even have a vague idea of what kind of jobs they have and some of the major tasks they regularly perform. But does any of this tell us how to make them happy? Does it tell us *how* they will actually use the product we're

building? Does it tell us *why* they are doing whatever it is they might need our product for, *why* they might want to choose our product over our competitors, or *how* we can make sure they do? No, it does not.

However, we should not give up hope. It is possible to understand our users well enough to make excellent products they will love. We'll see how to address the issue of understanding users and their behaviors with products in Chapters 2 and 3.

Conflicts of interest

A third problem affects the ability of vendors and manufacturers to make users happy. The world of digital product development has an important conflict of interest: The people who build the products—developers—are often also the people who design them. They are are also, quite understandably, the people who usually have the final say on what does and doesn't get built. Thus, developers often are required to choose between ease of coding and ease of use. Because developers' performance is typically judged by their ability to code efficiently and meet incredibly tight deadlines, it isn't difficult to figure out what direction most software-enabled products take. Just as we would never permit the prosecutor in a legal trial to also adjudicate the case, we should make sure that the people designing a product are not the same people building it. Even with appropriate skills and the best intentions, it simply isn't possible for a developer (or anyone, for that matter) to advocate effectively for the user, the business, and the technology all at the same time.

We'll see how to address the issue of building design teams and fitting them into the planning and development process in Chapter 6.

Lack of a design process

The last reason the digital product industry isn't cranking out successful, well-designed products is that it has no reliable *process* for doing so. Or, to be more accurate, it doesn't have a *complete* process for doing so. Engineering departments follow—or should follow—rigorous engineering methods that ensure the *feasibility* and quality of the technology. Similarly, marketing, sales, and other business units follow their own well-established methods for ensuring the commercial *viability* of new products. What's left out is a repeatable, predictable, and analytical process for ensuring **desirability**: *transforming an understanding of users into products that meet their professional, personal, and emotional needs.*

In the worst case, decisions about what a digital product will do and how it will communicate with users are simply a by-product of its construction. Developers, deep in their thoughts of algorithms and code, end up "designing" product behaviors in the same way

that miners end up "designing" a landscape filled with cavernous pits and piles of rubble. In unenlightened development organizations, the digital product interaction design process alternates between the accidental and the nonexistent.

Sometimes organizations do adopt a design process, but it isn't quite up to the task. Many companies embrace the notion that integrating customers (or their theoretical proxies, domain experts) directly into the development process can solve human interface design problems. Although this has the salutary effect of sharing the responsibility for design with the user, it ignores a serious methodological flaw: confusing domain knowledge with design knowledge.

Customers, although they might be able to articulate the problems with an interaction, often cannot visualize the solutions to those problems. Design is a specialized skill, just like software development. Developers would never ask users to help them *code*; design problems should be treated no differently. In addition, customers who *purchase* a product may not be the same people who *use* it from day to day, a subtle but important distinction. Finally, experts in a domain may not be able to easily place themselves in the shoes of less-expert users when defining tasks and flows. Interestingly, the two professions that seem to most frequently confuse domain knowledge with design knowledge when building information systems—law and medicine—have notoriously difficult-to-use products. Coincidence? Probably not.

Of course, designers should indeed get feedback on their proposed solutions, both from users and the product team. But hearing about the problems is much more useful to designers—and better for the product—than taking proposed solutions from users at face value. In interpreting feedback, the following analogy is useful: Imagine a patient who visits his doctor with acute stomach pain. "Doctor," he says, "it *really* hurts. I think it's my appendix. You've got to take it out as soon as possible." A responsible physician wouldn't perform surgery based solely on a patient request, even an earnest one. The patient can describe the symptoms, but it takes the doctor's professional knowledge to make the correct diagnosis and prescribe the treatment.

Planning and Designing Product Behavior

The planning of complex digital products, especially ones that interact directly with humans, requires a significant upfront effort by professional designers, just as the planning of complex physical structures that interact with humans requires a significant upfront effort by professional architects. In the case of architects, that planning involves understanding how the humans occupying the structure live and work, and designing spaces to support and facilitate those behaviors. In the case of digital products, the planning involves understanding how the humans using the product live and work, and designing product behavior and form that support and facilitate the human behaviors. Architecture is an old,

well-established field. The design of product and system behavior—**interaction design**—is quite new, and only in recent years has it begun to come of age as a discipline. And this new design has fundamentally changed how products succeed in the marketplace.

In the early days of industrial manufacturing, engineering and marketing processes alone were sufficient to produce *desirable* products: It didn't take much more than good engineering and reasonable pricing to produce a hammer, diesel engine, or tube of toothpaste that people would readily purchase. As time progressed, manufacturers of consumer products realized that they needed to differentiate their products from functionally identical products made by competitors, so design was introduced as a means to increase user desire for a product. Graphic designers were employed to create more effective packaging and advertising, and industrial designers were engaged to create more comfortable, useful, and exciting forms.

The conscious inclusion of design heralded the ascendance of the modern triad of product development concerns identified by Larry Keeley of the Doblin Group: capability, viability, and desirability (see Figure 1-3). If any of these three foundations is weak, a product is unlikely to stand the test of time.

Now enter the general-purpose computer, the first machine capable of almost limitless *behavior* via software programming. The interesting thing about this complex behavior, or interactivity, is that it completely alters the nature of the products it touches. Interactivity is compelling to humans—so compelling that the other aspects of an interactive product become marginal. Who pays attention to the black PC tower that sits under your desk? It is the screen, keyboard, and mouse to which users pay attention. With touch-screen devices like the iPad and its brethren, the only apparent hardware is the interactive surface. Yet the behaviors of software and other digital products, which should receive the majority of design attention, all too frequently receive no attention.

The traditions of design that corporations have relied on to provide the critical pillar of desirability for products don't provide much guidance in the world of interactivity. *Design of behavior* is a different kind of problem that requires greater knowledge of *context*, not just rules of visual composition and brand. It requires an understanding of the user's relationship with the product from before purchase to end of life. Most important is the understanding of how the user wants to use the product, in what ways, and to what ends.

Interaction design isn't merely a matter of aesthetic choice; rather, it is based on an understanding of users and cognitive principles. This is good news, because it makes the design of behavior quite amenable to a repeatable process of analysis and synthesis. It doesn't mean that the design of behavior can be automated, any more than the design of form or content can be automated, but it *does* mean that a systematic approach is possible. Rules of form and aesthetics mustn't be discarded, of course. They must work in harmony with the larger concern of achieving user goals via appropriately designed behaviors.

A successful product is desirable, viable, and buildable.

Desirability — What do people need?

Capability — What can we build?

Viability — What will sustain a business?

Designers

User model
- motivations
- behaviors
- attitudes and aptitudes

Product design
- design schedule
- form and behavior spec

Management

Business model
- funding model
- income / expense projections, etc.

Business plan
- marketing plan
- launch plan
- distribution plan

Technologists

Technology model
- core technologies
- technology components
- build vs. buy

Technology plan
- engineering schedule
- engineering spec

User effectiveness and customer adoption

Sustainable business

Project delivery

Overall product success

You can apply this to companies that have struggled to find the balance:

Apple	Microsoft	Novell
Apple has emphasized desirability but has made many business blunders. Nevertheless, it is sustained by the loyalty created by its attention to user experience.	Microsoft is one of the best run businesses ever, but it has not been able to create highly desirable products. This provides an opening for competition.	Novell emphasized technology and gave little attention to desirability. This made it vulnerable to competition.

Figure 1-3: Building successful digital products. Three major processes need to be followed in tandem to create successful technology products. This book addresses the first and foremost issue: how to create a product people will desire.

This book presents a set of methods to address this new kind of behavior-oriented design, providing a complete process for understanding users' *goals*, needs, and motivations: **Goal-Directed Design**. To understand the process of Goal-Directed Design, we first need to understand the nature of user goals, the *mental models* from which they arise, and how they are the key to designing appropriate interactive behavior.

Recognizing User Goals

So what are user goals? How can we identify them? How do we know that they are real goals, rather than tasks users are forced to perform by poorly designed tools or business processes? Are they the same for all users? Do they change over time? We'll try to answer those questions in the remainder of this chapter.

Users' goals are often quite different from what we might guess them to be. For example, we might think that an accounting clerk's goal is to process invoices efficiently. This is probably not true. Efficient invoice processing is more likely the goal of the clerk's employer. The clerk probably concentrates on goals like appearing competent at his job and keeping himself engaged with his work while performing routine and repetitive tasks—although he may not verbally (or even consciously) acknowledge this.

Regardless of the work we do and the tasks we must accomplish, most of us share these simple, personal goals. Even if we have higher aspirations, they are still more personal than work-related: winning a promotion, learning more about our field, or setting a good example for others, for instance.

Products designed and built to achieve business goals alone will eventually fail; users' personal goals need to be addressed. When the design meets the user's personal goals, business goals are achieved far more effectively, for reasons we'll explore in more detail in later chapters.

If you examine most commercially available software, websites, and digital products, you will find that their user interfaces fail to meet user goals with alarming frequency. They routinely:

- Make users feel stupid.
- Cause users to make big mistakes.
- Require too much effort to operate effectively.
- Don't provide an engaging or enjoyable experience.

Most of the same software is equally poor at achieving its business purpose. Invoices don't get processed all that well. Customers don't get serviced on time. Decisions don't get properly supported. This is no coincidence.

The companies that develop these products have the wrong priorities. Most focus far too narrowly on implementation issues, which distract them from users' needs.

Even when businesses become sensitive to their users, they are often powerless to change their products. The conventional development process assumes that the user interface should be addressed after coding begins—sometimes even after it ends. But just as you cannot effectively design a building after construction begins, you cannot easily make an application serve users' goals as soon as a significant and inflexible code base is in place.

Finally, when companies *do* focus on the users, they tend to pay too much attention to the *tasks* users engage in and not enough attention to their *goals* in performing those tasks. Software can be technologically superb and perform each business task with diligence, yet still be a critical and commercial failure. We can't ignore technology or tasks, but they play only a part in a larger schema that includes designing to meet user goals.

Goals versus tasks and activities

Goals are not the same as tasks or activities. A goal is an expectation of an end condition, whereas both activities and tasks are intermediate steps (at different levels of organization) that help someone to reach a goal or set of goals.

Donald Norman[1] describes a hierarchy in which activities are composed of tasks, which in turn are composed of actions, which are themselves composed of operations. Using this scheme, Norman advocates Activity-Centered Design (ACD), which focuses first and foremost on understanding activities. He claims humans adapt to the tools at hand and that understanding the activities people perform with a set of tools can more favorably influence the design of those tools. The foundation of Norman's thinking comes from Activity Theory, a Soviet-era Russian theory of psychology that emphasizes understanding who people are by understanding how they interact with the world. In recent years this theory has been adapted to the study of human-computer interaction, most notably by Bonnie Nardi.[2]

Norman concludes, correctly, that the traditional task-based focus of digital product design has yielded inadequate results. Many developers and usability professionals still approach interface design by asking what the tasks are. Although this may get the job done, it won't produce much more than an incremental improvement: It won't provide a solution that differentiates your product in the market, and very often it won't really satisfy the user.

While Norman's ACD takes some important steps in the right direction by highlighting the importance of the user's context, we do not believe it goes quite far enough. A method like ACD can be very useful in properly breaking down the "what" of user behaviors, but it really doesn't address the first question any designer should ask: *Why* is a user performing an activity, task, action, or operation in the first place? Goals motivate people to perform activities; understanding goals allows you to understand your users' expectations and aspirations, which in turn can help you decide which activities are truly relevant to your design. Task and activity analysis is useful at the detail level, but only after user goals have been analyzed. Asking, "What are the user's goals?" lets you understand the *meaning* of activities to your users and thus create more appropriate and satisfactory designs.

If you're still unsure about the difference between goals and activities or tasks, there is an easy way to tell the difference between them. Since goals are driven by human motivations, they change very slowly—if at all—over time. Activities and tasks are much more transient, because they are based almost entirely on whatever technology is at hand. For example, when someone travels from St. Louis to San Francisco, his *goals* are likely to include traveling quickly, comfortably, and safely. In 1850, a settler wishing to travel quickly and comfortably would have made the journey in a covered wagon; in the interest of safety, he would have brought along his trusty rifle. Today, a businessman traveling from St. Louis to San Francisco makes the journey in a jet and, in the interest of safety, he is required to leave his firearms at home. The *goals* of the settler and businessman remain unchanged, but their activities and tasks have changed so completely with the changes in technology that they are, in some respects, in direct opposition.

Design based solely on understanding activities or tasks runs the risk of trapping the design in a model imposed by an outmoded technology, or using a model that meets a corporation's goals without meeting the users' goals. Looking through the lens of goals allows you to leverage available technology to eliminate irrelevant tasks and to dramatically streamline activities. Understanding users' goals can help designers eliminate the tasks and activities that better technology renders unnecessary for humans to perform.

Designing to meet goals in context

Many designers assume that making user interfaces and product interactions easier to learn should always be a design target. Ease of learning is an important guideline, but in reality, the design target really depends on the context—who the users are, what they are doing, and their goals. You simply can't create good design by following rules disconnected from the goals and needs of the users of your product.

Consider an automated call-distribution system. The people who use this product are paid based on how many calls they handle. Their most important concern is not ease of learning, but the efficiency with which they can route calls, and the rapidity with which

those calls can be completed. Ease of learning is also important, however, because it affects employees' happiness and, ultimately, turnover rate, so both ease and throughput should be considered in the design. But there is no doubt that throughput is the dominant demand placed on the system by the users, so, if necessary, ease of learning should take a backseat. An application that walks the user through the call-routing process step by step each time merely frustrates him after he's learned the ropes.

On the other hand, if the product in question is a kiosk in a corporate lobby helping visitors find their way around, ease of use for first-time users is clearly a major goal.

A general guideline of interaction design that seems to apply particularly well to productivity tools is that *good design makes users more effective*. This guideline takes into account the universal human goal of not looking stupid, along with more particular goals of business throughput and ease of use that are relevant in most business situations.

It is up to you as a designer to determine how you can make the users of your product more effective. Software that enables users to perform their tasks *without addressing their goals* rarely helps them be truly effective. If the task is to enter 5,000 names and addresses into a database, a smoothly functioning data-entry application won't satisfy the user nearly as much as an automated system that extracts the names from the invoicing system.

Although it is the user's job to focus on her tasks, the designer's job is to look beyond the task to identify *who* the most important users are, and then to determine *what* their goals might be and *why*.

Implementation Models and Mental Models

The computer industry still makes use of the term *computer literacy*. Pundits talk about how some people have it and some don't, how those who have it will succeed in the information economy, and how those who lack it will inevitably fall between the socioeconomic cracks. Computer literacy, however, is really a euphemism for forcing human beings to stretch their thinking to understand the inner workings of application logic, rather than having software-enabled products stretch to meet people's usual ways of thinking.

Let's explore what's really going on when people try to use digital products, and what the role of design is in translating coded functions into an understandable and pleasurable experience for users.

Implementation models

Any machine has a mechanism for accomplishing its purpose. A motion picture projector, for example, uses a complicated sequence of intricately moving parts to create its illusion. It shines a very bright light through a translucent, miniature image for a fraction of a second. It then blocks the light for a split second while it moves another miniature image into place. Then it unblocks the light again for another moment. It repeats this process with a new image 24 times per second. Software-enabled products don't have mechanisms in the sense of moving parts; these are replaced with algorithms and modules of code that communicate with each other. The representation of how a machine or application actually works has been called the *system model* by Donald Norman and others; we prefer the term *implementation model* because it describes the details of how an application is implemented in code.

It is much easier to design software that reflects its implementation model. From the developer's perspective, it's perfectly logical to provide a button for every function, a field for every data input, a page for every transaction step, and a dialog box for every code module. But while this adequately reflects the infrastructure of engineering efforts, it does little to provide coherent mechanisms for a user to achieve his goals. In the end, what is produced alienates and confuses the user, rather like the ubiquitous external ductwork in the dystopian setting of Terry Gilliam's movie *Brazil* (which is full of wonderful tongue-in-cheek examples of miserable interfaces).

Mental models

From the moviegoer's point of view, it is easy to forget the nuance of sprocket holes and light interrupters while watching an absorbing drama. Many moviegoers, in fact, have little idea how the projector works, or how this differs from the way a television works. The viewer imagines that the projector merely throws a picture that moves onto the big screen. This is his *mental model*, or *conceptual model*.

People don't need to know all the details of how a complex mechanism actually works in order to use it, so they create a cognitive shorthand for explaining it. This explanation is powerful enough to cover their interactions with it but doesn't necessarily reflect its actual inner mechanics. For example, many people imagine that, when they plug in their vacuum cleaner and blender, the electricity flows like water from the wall into the appliances through the little black tube of the electrical cord. This mental model is perfectly adequate for using household appliances. The fact that the implementation model of household electricity involves nothing resembling a fluid traveling through a tube and that there is a reversal of electrical potential 120 times per second is irrelevant to the user, although the power company needs to know the details.

In the digital world, however, the differences between a user's mental model and the implementation model are often quite distinct. We tend to ignore the fact that our cell phone doesn't work like a landline phone; instead, it is actually a radio transceiver that might swap connections between a half-dozen different cellular base antennas in the course of a two-minute call. Knowing this doesn't help us understand how to *use* the phone.

The discrepancy between implementation and mental models is particularly stark in the case of software applications, where the complexity of implementation can make it nearly impossible for the user to see the mechanistic connections between his actions and the application's reactions. When we use a computer to digitally edit sound or create video special effects like morphing, we are bereft of analogy to the mechanical world, so our mental models are necessarily different from the implementation model. Even if the connections were visible, they would remain inscrutable to most people.

Striving toward perfection: represented models

Software (and any digital product that relies on software) has a behavioral face it shows to the world that is created by the developer or designer. This representation is not necessarily an accurate description of what is really going on inside the computer, although unfortunately, it frequently is. This ability to *represent* the computer's functioning independent of its true actions is far more pronounced in software than in any other medium. It allows a clever designer to hide some of the more unsavory facts of how the software really gets the job done. This disconnection between what is implemented and what is offered as explanation gives rise to a *third* model in the digital world, the designer's *represented model*—how the designer chooses to represent an application's functioning to the user. Donald Norman calls this the *designer's model*.

In the world of software, an application's represented model can (and often should) be quite different from an application's actual processing structure. For example, an operating system can make a network file server look as though it were a local disk. The model does not represent the fact that the physical disk drive may be miles away. This concept of the represented model has no widespread counterpart in the mechanical world. Figure 1-4 shows the relationship between the three models.

The closer the represented model comes to the user's mental model, the easier he will find the application to use and understand. Generally, offering a represented model that follows the implementation model too closely significantly reduces the user's ability to learn and use the application. This occurs because the user's mental model of his tasks usually differs from the software's implementation model.

Implementation Model
reflects technology

Represented Models
worse   better

Mental Model
reflects user's vision

Figure 1-4: A comparison of the implementation model, mental model, and represented model. The way engineers must build software is often a given, dictated by various technical and business constraints. The model for how the software actually works is called the *implementation model*. The way users perceive the jobs they need to do and how the application helps them do so is their *mental model* of interaction with the software. It is based on their own ideas of how they do their jobs and how computers might work. The way designers choose to represent the working of the application to the user is called the *represented model*. Unlike the other two models, it is an aspect of software over which designers have great control. One of the designer's most important goals should be to make the represented model match a user's mental model as closely as possible. Therefore, it is critical that designers understand in detail how their target users think about the work they do with the software.

We tend to form mental models that are simpler than reality. So, if we create represented models that are simpler than the implementation model, we help the user achieve better understanding. In software, we imagine that a spreadsheet scrolls new cells into view when we click the scrollbar. Nothing of the sort actually happens. There is no sheet of cells out there, but a tightly packed data structure of values, with various pointers between them, from which the application synthesizes a new image to display in real time.

One of the most significant ways in which computers can assist human beings is presenting complex data and operations in a simple, easily understandable form. As a result, user interfaces that are consistent with users' mental models are vastly superior to those that are merely reflections of the implementation model.

> **DESIGN PRINCIPLE**
>
> *User interfaces should be based on user mental models rather than implementation models.*

In Adobe Photoshop Express on the iPad, users can adjust a set of ten different visual filters, including noise, contrast, exposure, and tint. Instead of offering numeric fields or many banks of controls for entering filter values—the implementation model—the interface instead shows a set of thumbnail images of the edited photo, each with a different filter applied (see Figure 1-5). A user can tap the image that best represents the desired result, and can tweak it with a single large slider. The interface more closely follows his

mental model, because the user—likely an amateur photographer—is thinking in terms of how his photo looks, not in terms of abstract numbers.
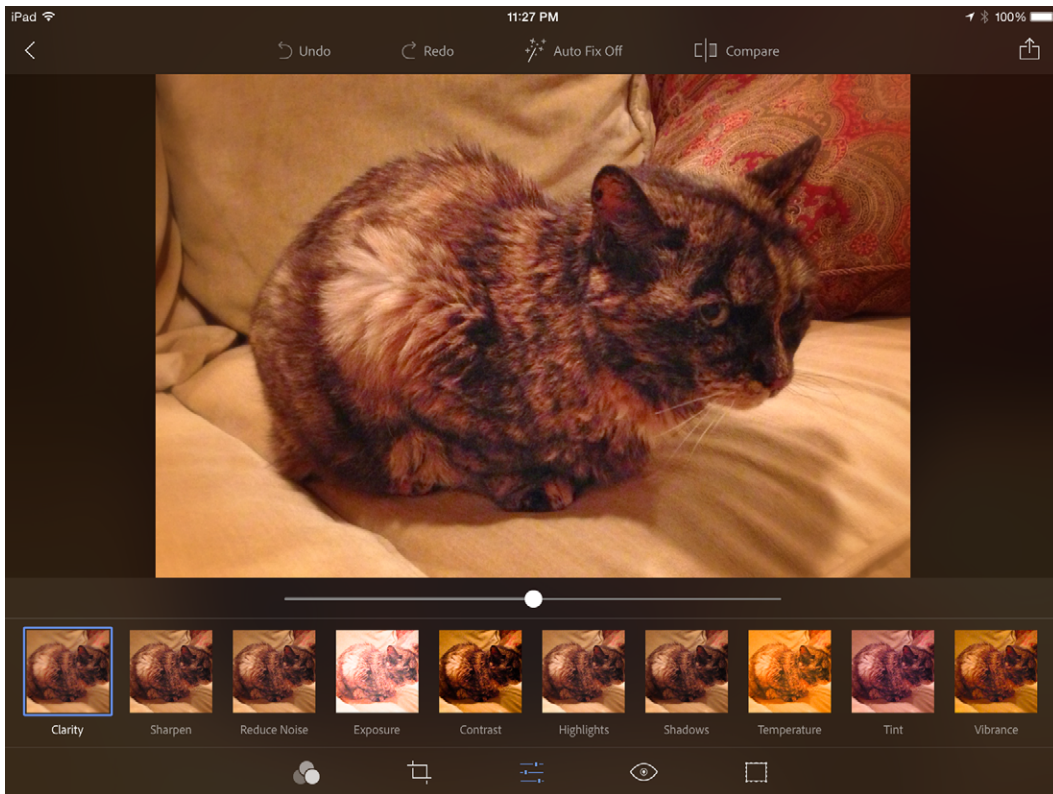
Figure 1-5: Adobe Photoshop Express for iPad has a great example of software design to match user mental models. The interface shows a set of thumbnail images of the photo being edited. A user can tap the thumbnail that best represents the desired setting, which can then be tweaked using the single large slider below the photo. The interface follows mental models of photographers who are after a particular look, not a set of abstract numeric values.

If the represented model for software closely follows users' mental models, it eliminates needless complexity from the user interface by providing a cognitive framework that makes it evident to the user how his goals and needs can be met.

> **DESIGN PRINCIPLE**   *Goal-directed interactions reflect user mental models.*

So, now we know that a missing link prevents the majority of digital products from being truly successful. A design process translates the implementation of features into intuitive and desirable product behaviors that match how people think about performing

tasks toward achieving their goals. But how do we actually do it? How do we know what our users' goals are and what mental models they have of their activities and tasks?

The Goal-Directed Design process, which we describe in the remainder of this chapter and in the remainder of Part I, provides a structure for determining the answers to these questions—a structure by which solutions based on this information can be systematically achieved.

An Overview of Goal-Directed Design

Most technology-focused companies don't have an adequate process for product design, if they have a process at all. But even the more enlightened organizations—those that can boast of an established process—come up against some critical issues that result from traditional ways of approaching the problems of research and design.

In recent years, the business community has come to recognize that user research is necessary to create good products, but the proper nature of that research is still in question in many organizations. Quantitative market research and market segmentation are quite useful for *selling* products but fall short of providing critical information about *how people actually use products*—especially products with complex behaviors. (See Chapter 2 for a more in-depth discussion of this topic.) A second problem occurs after the results have been analyzed: Most traditional methods don't provide a means of *translating research results into design solutions*. A hundred pages of user survey data don't easily translate into a set of product requirements. They say even less about how those requirements should be expressed in terms of a meaningful and appropriate interface structure. Design remains a black box: "A miracle happens here…" This gap between research results and the ultimate design solution is the result of a process that doesn't connect the dots from user to final product. We'll soon see how to address this problem with Goal-Directed methods.

Bridging the gap

As we have briefly discussed, the role of design in the development process needs to change. We need to start thinking about design in new ways and start thinking differently about how product decisions are made.

Design as product definition

Design has, unfortunately, become a limiting term in the technology industry. For many developers and managers, the word stands for what happens in the third process diagram shown in Figure 1-2: a visual facelift of the *implementation model*. But design, when

properly deployed (as in the fourth process diagram shown in Figure 1-2), both identifies user requirements and defines a detailed plan for the behavior and appearance of products. In other words, design provides true *product definition*, based on user goals, business needs, and technology constraints.

Designers as researchers

If design is to become product definition, designers need to take on a broader role than that assumed in traditional practice, particularly when the object in question is complex, interactive systems.

One of the problems with the current development process is that roles in the process are overspecialized: Researchers perform research, and designers perform design (see Figure 1-6). The results of user and market research are analyzed by the usability and market researchers and then thrown over the transom to designers or developers. What is missing in this model is a systematic means of translating and synthesizing the research into design solutions. One of the ways to address this problem is for designers to learn to be researchers.
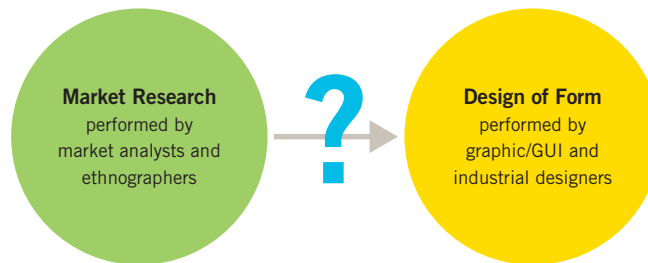
Figure 1-6: A problematic design process. Traditionally, research and design have been separated, with each activity handled by specialists. *Research* has, until recently, referred primarily to market research, and *design* is too often limited to visual design or skin-deep industrial design. More recently, user research has expanded to include qualitative, ethnographic data. Yet, without including designers in the research process, the connection between research data and design solutions remains tenuous at best.

There is a compelling reason to involve designers in the research process. One of the most powerful tools designers offer is empathy: the ability to feel what others are feeling. The direct and extensive exposure to users that proper user research entails immerses designers in the users' world and gets them thinking about users long before they propose solutions. One of the most dangerous practices in product development is isolating designers from the users, because doing so eliminates empathic knowledge.

Additionally, it is often difficult for pure researchers to know what user information is really important from a design perspective. Involving designers directly in research addresses both issues.

In the authors' practice, designers are trained in the research techniques described in Chapter 2 and perform their research without further support or collaboration. This is a satisfactory solution, provided that your team has the time and resources to train your designers fully in these techniques. If not, a cross-disciplinary team of designers and dedicated user researchers is appropriate.

Although research practiced by designers takes us part of the way to Goal-Directed Design solutions, a translation gap still exists between research results and design details. The puzzle is missing several pieces, as we will discuss next.

Between research and blueprint: Models, requirements, and frameworks

Few design methods in common use today incorporate a means of effectively and systematically translating the knowledge gathered during research into a detailed design specification. Part of the reason for this has already been identified: Designers have historically been out of the research loop and have had to rely on third-person accounts of user behaviors and desires.

The other reason, however, is that few methods capture user behaviors in a manner that appropriately directs the definition of a product. Rather than providing information about user goals, most methods provide information at the task level. This type of information is useful for defining layout, work flow, and translation of functions into interface controls. But it's less useful for defining the basic framework of what a product *is*, what it *does*, and how it should meet the user's broad needs.

Instead, we need an explicit, systematic process to bridge the gap between research and design for defining user models, establishing design requirements, and translating those into a high-level interaction framework (see Figure 1-7). Goal-Directed Design seeks to bridge the gap that currently exists in the digital product development process—the gap between user research and design—through a combination of new techniques and known methods brought together in more effective ways.
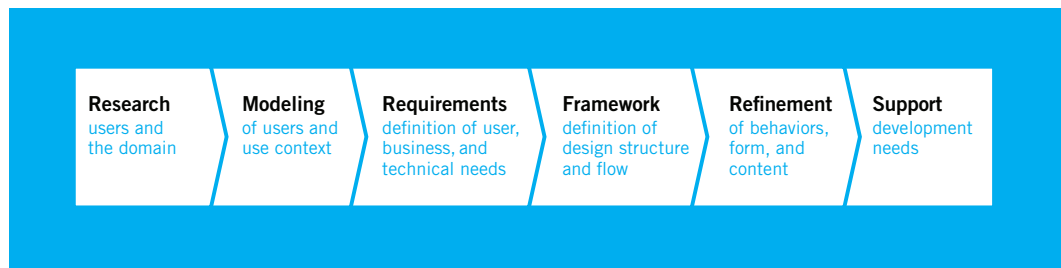
Figure 1-7: The Goal-Directed Design process

A process overview

Goal-Directed Design combines techniques of ethnography, stakeholder interviews, market research, detailed user models, scenario-based design, and a core set of interaction principles and patterns. It provides solutions that meet users' needs and goals while also addressing business/organizational and technical imperatives. This process can be roughly divided into six phases: Research, Modeling, Requirements Definition, Framework Definition, Refinement, and Support (see Figure 1-7). These phases follow the five component activities of interaction design identified by Gillian Crampton Smith and Philip Tabor—understanding, abstracting, structuring, representing, and detailing—with a greater emphasis on modeling user behaviors and defining system behaviors.

The remainder of this chapter provides a high-level view of the six phases of Goal-Directed Design, and Chapters 2 through 6 provide a more detailed discussion of the methods involved in each of these phases. Figure 1-8 shows a more detailed diagram of the process, including key collaboration points and design concerns.

Research

The Research phase employs ethnographic field study techniques (observation and contextual interviews) to provide qualitative data about potential and/or actual users of the product. It also includes competitive product audits as well as reviews of market research, technology white papers, and brand strategy. It also includes one-on-one interviews with stakeholders, developers, subject matter experts (SMEs), and technology experts as suits the particular domain.

One of the principal outcomes of field observation and user interviews is an emergent set of *behavior patterns*—identifiable behaviors that help categorize modes of use of a potential or existing product. These patterns suggest goals and motivations (specific and general desired outcomes of using the product). In business and technical domains, these behavior patterns tend to map into professional roles; for consumer products, they tend to correspond to lifestyle choices. Behavior patterns and the goals associated with them drive the creation of *personas* in the Modeling phase. Market research helps select and filter valid personas that fit business models. Stakeholder interviews, literature reviews, and product audits deepen the designers' understanding of the domain and elucidate business goals, brand attributes, and technical constraints that the design must support.

Chapter 2 provides a more detailed discussion of Goal-Directed research techniques.

Goal-Directed Design

	Activity	Concerns	Stakeholder Collaboration	Deliverable
Research	**Scope** Define project goals and schedule	Objectives, timelines, financial constraints, process, milestones	Meetings Capabilities and Scoping	Document Statement of Work
	Audit Review existing work and product	Business and marketing plans, branding strategy, market research, product portfolio plans, competitors, relevant technologies		
	Stakeholder Interviews Understand product vision and constraints	Product vision, risks, constraints, opportunities, logistics, users	Interviews Stakeholders and Users	
	User interviews & observations Understand user needs and behavior	Users, potential users, behaviors, attitudes, aptitudes, motivations, environments, tools, challenges	Check-in Preliminary Research findings	
Modeling	**Personas** User and customer archetypes	Patterns in user and customer behaviors, attitudes, aptitudes, goals, environments, tools, challenges	Check-in Personas	
	Other Models Represent domain factors beyond individual users and customers	Workflows among multiple people, environments, artifacts		
Requirements Definition	**Context Scenarios** Tell stories about ideal user experiences	How the product fits into the persona's life and environment, and how it helps them achieve their goals	Check-in Scenarios and Requirements	
	Requirements Describe necessary capabilities of the product	Functional and data needs, user mental models, design imperatives, product vision, business requirements, technology	Presentation User and Domain Analysis	Document User and Domain Analysis
Design Framework	**Elements** Define manifestations of information and functionality	Information, functions, mechanisms, actions, domain object models	Check-ins Design Framework	
	Framework Design overall structure of user experience	Object relationships, conceptual groupings, navigation sequencing, principles and patterns, flow, sketches, storyboards		
	Key Path and Validation Scenarios Describe how the persona interacts with the product	How the design fits into an ideal sequence of user behaviors, and accommodates a variety of likely conditions	Presentation Design Vision	
Design Refinement	**Detailed design** Refine and specify details	Appearance, idioms, interface, widgets, behavior, information, visualization, brand, experience, language, storyboards	Check-ins Design Refinement	Document Form and Behavior Specification
Design Support	**Design modification** Accommodate new constraints and timeline	Maintaining conceptual integrity of the design under changing technology constraints	Collaborative Design	Revision Form and Behavior Specification

Figure 1-8: A more detailed look at the Goal-Directed Design process

Modeling

During the Modeling phase, behavior and work flow patterns discovered by analyzing the field research and interviews are synthesized into domain and user models. Domain models can include information flow and work flow diagrams. User models, or personas, are detailed, composite *user archetypes* that represent distinct groupings of behaviors, attitudes, aptitudes, goals, and motivations observed and identified during the Research phase.

Personas are the main characters in a narrative, scenario-based approach to design. This approach iteratively generates design concepts in the Framework Definition phase. It provides feedback that enforces design coherence and appropriateness in the Refinement phase. It also is a powerful communication tool that helps developers and managers understand design rationale and prioritize features based on user needs. In the Modeling phase, designers employ a variety of methodological tools to synthesize, differentiate, and prioritize personas, exploring different *types* of goals and mapping personas across ranges of behavior to ensure that no gaps or duplications exist.

Specific design targets are chosen from the cast of personas through a process of comparing goals and assigning priorities based on how broadly each persona's goals encompass the goals of other personas. A process of designating persona types determines how much influence each persona has on the design's eventual form and behavior.

A detailed discussion of persona and goal development can be found in Chapter 3.

Requirements Definition

Design methods employed by teams during the Requirements Definition phase provide the much-needed connection between user and other models and design's framework. This phase employs scenario-based design methods with the important innovation of focusing the scenarios not on user tasks in the abstract, but first and foremost on meeting the goals and needs of specific user personas. Personas help us understand which tasks are truly important and why, leading to an interface that minimizes necessary tasks (effort) while maximizing return. Personas become the main characters of these scenarios, and the designers explore the design space via a form of role playing.

For each interface/primary persona, the process of design in the Requirements Definition phase involves analyzing persona data and functional needs (expressed in terms of objects, actions, and contexts), prioritized and informed by persona goals, behaviors, and interactions with other personas in various contexts.

This analysis is accomplished through an iteratively refined *context scenario*. It starts with a "day in the life" of the persona using the product, describing high-level product

touch points, and thereafter successively defining detail at ever-deepening levels. In addition to these scenario-driven requirements, designers consider the personas' skills and physical capabilities as well as issues related to the usage environment. Business goals, desired brand attributes, and technical constraints are also considered and balanced with persona goals and needs. The output of this process is a *requirements definition* that balances user, business, and technical requirements of the design to follow.

Chapter 4 covers the process of establishing requirements through the use of *scenarios*.

Framework Definition

In the Framework Definition phase, designers create the overall product concept, defining the basic frameworks for the product's behavior, visual design, and, if applicable, physical form. Interaction design teams synthesize an *interaction framework* by employing two other critical methodological tools in conjunction with context scenarios. The first is a set of general *interaction design principles* that provide guidance in determining appropriate system behavior in a variety of contexts. Part II of this book is devoted to high-level interaction design principles appropriate to the Framework Definition phase.

The second critical methodological tool is a set of *interaction design patterns* that encode general solutions (with variations dependent on context) to classes of previously analyzed problems. These patterns bear close resemblance to the concept of architectural design patterns first developed by Christopher Alexander[3] and more recently brought to the programming field by Erich Gamma and others.[4] Interaction design patterns are hierarchically organized and continuously evolve as new contexts arise. Rather than stifling designer creativity, they often provide needed leverage to approach difficult problems with proven design knowledge.

After data and functional needs are described at this high level, they are translated into design elements according to interaction principles and then organized, using patterns and principles, into design sketches and behavior descriptions. The output of this process is an *interaction framework definition*, a stable design concept that provides the logical and hi-level formal structure for the detail to come. Successive iterations of more narrowly focused scenarios provide this detail in the Refinement phase. The approach is often a balance of top-down (pattern-oriented) design and bottom-up (principle-oriented) design.

When the product takes physical form, interaction designers and industrial designers begin by collaborating closely on various *input methods* and approximate *form factors* the product might take, using scenarios to consider the pros and cons of each. As this is narrowed to a couple of options that seem promising, industrial designers begin producing early physical prototypes to ensure that the overall interaction concept will work. It's

critical at this early stage that industrial designers not create concepts independent of the product's behavior.

When working to design a service, we will collaborate with service designers to draft a *service map* and a *blueprint* that coordinates touchpoints and experiences across channels, both "backstage" with the service providers and "frontstage" experiences from the users' point of view.

As soon as an interaction framework begins to emerge, visual interface designers produce several options for a *visual framework*, which is sometimes also called a *visual language strategy*. They use brand attributes as well as an understanding of the overall interface structure to develop options for typography, color palettes, and visual style.

Refinement

The Refinement phase proceeds similarly to the Framework Definition phase, but with increasing focus on detail and implementation. Interaction designers focus on task coherence, using *key path scenarios* (walkthroughs) and *validation scenarios* focused on storyboarding paths through the interface in great detail. Visual designers define a system of type styles and sizes, icons, and other visual elements that provide a compelling experience with clear affordances and visual hierarchy. Industrial designers, when appropriate, finalize materials and work closely with engineers on assembly schemes and other technical issues. The culmination of the Refinement phase is the detailed documentation of the design—a *form and behavior specification or blueprint*, delivered in either paper or interactive media form as the context dictates.

Chapter 5 discusses in more detail the use of personas, scenarios, principles, and patterns in the Framework Definition and Refinement phases.

Development support

Even a very well-conceived and validated design solution can't possibly anticipate every development challenge and technical question. In our practice, we've learned that it's important to be available to answer developers' questions as they arise during the construction process. It is often the case that as the development team prioritizes their work and makes trade-offs to meet deadlines, the design must be adjusted, requiring scaled-down design solutions. If the interaction design team is not available to create these solutions, developers are forced to do this under time pressure, which has the potential to gravely compromise the integrity of the product's design.

Chapter 6 discusses how interaction design activities and processes can be integrated with the larger product team.

Goals, not features, are the key to product success

Developers and marketers often use the language of features and functions to discuss products. But reducing a product's definition to a list of features and functions ignores the real opportunity—orchestrating technological capability to serve human needs and goals. Too often the features of our products are a patchwork of nifty technological innovations structured around a marketing requirements document or organization of the development team, with too little attention paid to the overall user experience.

The successful interaction designer must maintain her focus on users' goals amid the pressures and chaos of the product-development cycle. Although we discuss many other techniques and tools of interaction in this book, we always return to users' goals. They are the bedrock upon which interaction design should be practiced.

The Goal-Directed process, with its clear rationale for design decisions, makes collaboration with developers and businesspeople easier. It also ensures that the design in question isn't guesswork, the whim of a creative mind, or just a reflection of the team members' personal preferences.

> **DESIGN PRINCIPLE**   *Interaction design is not guesswork.*

Goal-Directed Design is a powerful tool for answering the most important questions that crop up during the definition and design of a digital product:

- Who are my users?
- What are my users trying to accomplish?
- How do my users think about what they're trying to accomplish?
- What kind of experiences do my users find appealing and rewarding?
- How should my product behave?
- What form should my product take?
- How will users interact with my product?
- How can my product's functions be most effectively organized?
- How will my product introduce itself to first-time users?
- How can my product put an understandable, appealing, and controllable face on technology?
- How can my product deal with problems that users encounter?

- How will my product help infrequent and inexperienced users understand how to accomplish their goals?
- How can my product provide sufficient depth and power for expert users?

The remainder of this book is dedicated to answering these questions. We share tools tested by years of experience with hundreds of products that can help you identify key users of your products, understand them and their goals, and translate this understanding into effective and appealing design solutions.

Notes
1. Norman, 2005
2. Nardi, 1996
3. Alexander, 1979
4. Gamma, et al, 1994

UNDERSTANDING THE PROBLEM: DESIGN RESEARCH

The outcome of any design effort ultimately must be judged by how successfully it meets the needs of both the product's users and the organization that commissioned it. No matter how skillful or creative the designer, if she does not have clear and detailed knowledge of the users she is designing for, the problem's constraints, and the business or organizational goals that are driving the design, she will have little chance of success.

Insight into these topics can't easily be achieved by sifting through the numbers and graphs that come from *quantitative* studies such as market surveys (although these can be critical for answering other kinds of questions). Rather, this kind of behavioral and organizational knowledge can best be gathered via *qualitative* research techniques. There are many types of qualitative research, each of which can play an important role in understanding a product's design landscape. In this chapter, we'll focus on the specific qualitative research techniques that support the design methods we'll cover in later chapters. We'll also discuss how qualitative research techniques can enrich and be enriched by certain quantitative techniques. At the end of the chapter, we'll briefly cover some supplemental qualitative techniques and the research contexts for which they are appropriate.

Qualitative versus Quantitative Data in Design Research

"Research" is a term that most people associate with science and objectivity. This association isn't incorrect, but it causes many people to believe that the only valid research is the kind that yields (supposedly) objective results: quantitative data. It is a common perspective in business and engineering that numbers represent truth. But numbers—especially statistics describing human activities—are subject to interpretation and can be manipulated just as easily as textual data.

Data gathered by the hard sciences like physics is simply different from that gathered about human activities. Electrons don't have moods that vary from minute to minute. The tight controls physicists place on their experiments to isolate observed behaviors are next to impossible in the social sciences. Any attempt to reduce human behavior to statistics is likely to overlook important nuances, which can make a huge difference in the design of products. Quantitative research can only answer questions about "how much" or "how many" along a few reductive axes. Qualitative research can tell you what, how, and why in rich detail that reflects the complexities of real human situations.

Social scientists have long realized that human behaviors are too complex and subject to too many variables to rely solely on quantitative data to understand them. Design and usability practitioners, borrowing techniques from anthropology and other social sciences, have developed many qualitative methods for gathering useful data on user behaviors to a more pragmatic end: to help create products that better serve user needs.

Benefits of qualitative methods

Qualitative research helps us understand a product's domain, context, and constraints in different, more useful ways than quantitative research does. It also helps us identify *patterns of behavior* among a product's users and potential users much more quickly and easily than would be possible with quantitative approaches. In particular, qualitative research helps us understand the following:

- Behaviors, attitudes, and aptitudes of potential and existing product users
- Technical, business, and environmental contexts—the *domain*—of the product to be designed

- Vocabulary and other social aspects of the domain in question
- How existing products are used

Qualitative research can also help the progress of design projects:

- It gives the design team credibility and authority, because design decisions can be traced to research results.
- It gives the team a common understanding of domain issues and user concerns.
- It empowers management to make more informed decisions about product design issues that would otherwise be based on guesswork or personal preference.

It's our experience that, in comparison, qualitative methods tend to be faster, less expensive, and more likely to provide useful answers to important questions that lead to superior design:

- How does the product fit into the broader context of people's lives?
- What goals motivate people to use the product, and what basic tasks help people accomplish these goals?
- What experiences do people find compelling? How do these relate to the product being designed?
- What problems do people encounter with their current ways of doing things?

The value of qualitative studies is not limited to helping support the design process. In our experience, spending the time to deeply understand the user population can provide valuable business insights that are not revealed through traditional market research.

For example, a client once asked us to perform a user study for an entry-level consumer video-editing product for Windows users. An established developer of video-editing and -authoring software, the client had used traditional market research techniques to identify a significant business opportunity in developing a product for people who owned a digital video camera and a computer but hadn't yet connected the two.

In the field, we conducted interviews with a dozen users in the target market. Our first discovery was not surprising: The people who did the most videography and had the strongest desire to share edited versions of their videos were parents. The second discovery, however, was startling: Of the 12 people whose homes we visited, only one person

had successfully connected his video camera to his computer—and he had asked the IT guy at work to do it. One of the necessary preconditions of the product's success was that people should be able to transfer videos to their computer for editing. But at the time the project was commissioned, it was extremely difficult to make a FireWire or video-capture card function properly on an Intel-based PC.

As a result of four days of research, we were able to help our client decide to put the product on hold. This decision likely saved them a considerable amount of money.

Strengths and limitations of quantitative methods

The marketing profession has taken much of the guesswork out of determining what motivates people to buy. One of the most powerful tools for doing so is market segmentation. Data from focus groups and market surveys is used to group potential customers by demographic criteria such as age, gender, amount of education, and home zip code. This helps determine what types of consumers will be most receptive to a particular product or marketing message. More-sophisticated consumer data may also include psychographics and behavioral variables, including attitudes, lifestyle, values, ideology, risk aversion, and decision-making patterns. Classification systems such as SRI's VALS segmentation and Jonathan Robbin's geodemographic PRIZM clusters can better clarify the data by predicting consumers' purchasing power, motivation to buy, self-orientation, and resources.

However, understanding if somebody *wants to buy* something is not the same thing as understanding what he or she might *want to do with it* after buying it. Market segmentation is a great tool for identifying and quantifying a market opportunity, but it's an ineffective tool for *defining a product* that will capitalize on that opportunity.

Similarly, quantitative measures such as web analytics and other attempts to numerically describe human behavior may certainly provide insightful answers to the *what* (or, at least, the *how much*) of the equation. But without a robust body of primary qualitative research that provides answers to the *why* of those behaviors, such statistical data may simply raise more questions than it answers.

Quantitative research can help direct design research

Qualitative research is almost always the most effective means for gathering the behavioral knowledge that will help designers define and design products for users. However, quantitative data is not without its uses within the context of design research.

For example, market-modeling techniques can accurately forecast marketplace acceptance of products and services. As such, they are invaluable for assessing a product's *viability*. They can therefore be powerful tools for convincing executives to build a product. After all, if you know X people might buy a product or service for Y dollars, it is easier to evaluate the potential return on investment. Because market research can identify and quantify the *business* opportunity, it is often the necessary starting point for funding a design initiative.

In addition, designers planning to interview and observe users can refer to market research (when it exists) to help select interview targets. Particularly in the case of consumer products and services, demographic attributes such as lifestyle choice and stage of life more strongly influence user behaviors. We'll discuss the differences between market segmentation models and user models (personas) in more detail in Chapter 3.

Similarly, web and other *usage data analytics* are an excellent way to identify design problems that may be in need of solutions. If users are lingering in an area of your website or are not visiting any other areas, that is critical information to have before a redesign. But it will likely take qualitative research to help determine the root cause of such statistically gathered behaviors and thus derive the potential solutions. And, of course, analytics are useful only when you have an existing product to run them on.

User research can inform market research

Qualitative research is almost always the tool of choice for characterizing user behaviors and latent needs. But there is one type of information, often seen as critical by business stakeholders, that qualitative research can't get you by itself: market sizing of the behavioral models. This is an ideal place to employ quantitative techniques (such as surveys) to fill in this missing information.

Once your users have been successfully represented by behavioral models (personas, which you'll read more about in Chapter 3), you can construct a survey. It will distinguish these different user types and capture traditional market demographic data that can then be correlated to the behavioral data. When this is successful, it can help determine—especially for consumer products—which user types should be prioritized when designing features and the overall product experience. We'll discuss this kind of persona-based market sizing a bit more in Chapter 3.

Figure 2-1 shows the relationship between various types of quantitative research and the qualitative Goal-Directed Design research techniques discussed in this chapter.
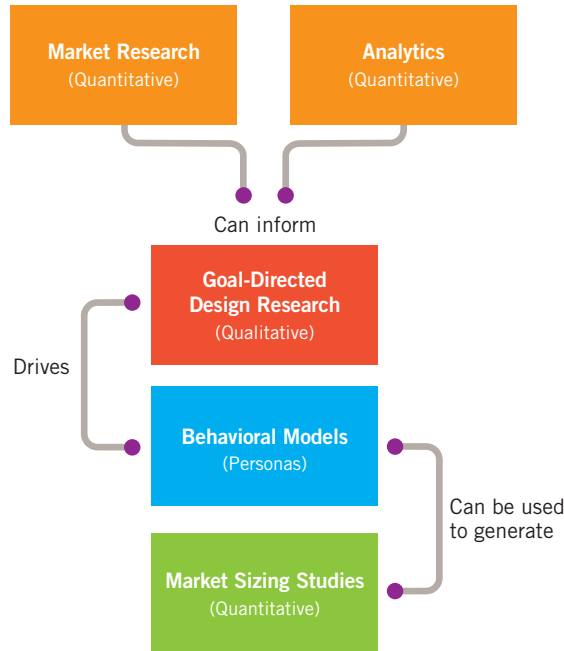
Figure 2-1: The relationship between quantitative research and qualitative, Goal-Directed design research

Goal-Directed Design Research

Social science and usability texts are full of methods and techniques for conducting qualitative research; we encourage you to explore this literature. In this chapter, we focus on techniques that have been proven effective in our practice over the last decade. Occasionally we draw attention to similar techniques practiced in the design and usability fields at large. We avoid getting bogged down in theory, instead presenting these techniques in a succinct and pragmatic manner.

We have found the following qualitative research activities to be most useful in our Goal-Directed design practice (in rough order of execution):

- Kickoff meeting
- Literature review
- Product/prototype and competitive audits
- Stakeholder interviews

- Subject matter expert (SME) interviews
- User and customer interviews
- User observation/ethnographic field studies

Figure 2-2 shows these activities.
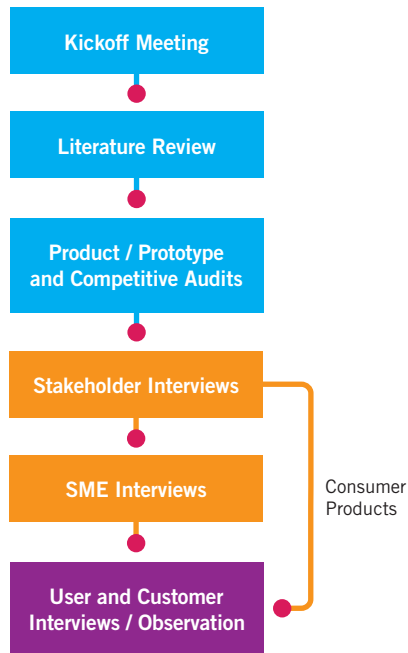
Figure 2-2: An overview of the Goal-Directed design research process

Kickoff meeting

Although the project kickoff meeting isn't strictly a research activity, it contains an important component of research: It is an opportunity for designers to ask initial key questions of some of the most important stakeholders gathered at the meeting:

- What is the product?
- Who will/does use it?
- What do your users need most?
- Which customers and users are the most important to the business?

- What challenges do the design team and the business face moving forward?
- Who do you see as your biggest competitors? Why?
- What internal and external literature should we look at to familiarize ourselves with the product and/or business and technical domain?

Although these questions may seem basic, they give the design team insight into not only the product itself, but also into how stakeholders think about their product, their users, and the design problem ahead. They will likely provide important clues about how to structure stakeholder and user interviews later in the process and will provide pointers to understanding the product domain. (This is particularly important if that domain is narrow or technical.)

Literature review

Prior to or in parallel with stakeholder interviews, the design team should review any literature pertaining to the product or its domain. This can and should include the following types of documents:

- **Internal documents** including product marketing plans, brand strategy, market research studies, user surveys, technology specifications and white papers, competitive research, usability studies and metrics, customer support data such as call center statistics or transcripts, and user forum archives
- **Industry reports** such as business and technical journal articles
- **Web searches** for related and competing products, news items, independent user forums, blog posts, and social media discussion topics

The design team should collect this literature and use it as a basis for developing questions to ask stakeholders and SMEs. Later they can use it to supply additional domain knowledge and vocabulary and to check against compiled user data.

Product/prototype and competitive audits

Prior to or in parallel with stakeholder and SME interviews, it is helpful for the design team to examine any existing version or prototype of the product, as well as its chief competitors. Doing so gives the design team a sense of the state of the art and provides fuel for questions during these interviews. Ideally, the design team should engage in an informal or *expert review* (sometimes also called a heuristic review) of both the current design (if any) and competitive product interfaces. They should compare each against interaction and visual design principles (such as those found later in this book). This procedure both familiarizes the team with the strengths and limitations of what is currently available to users and provides a general idea of the product's current functional scope.

Stakeholder interviews

Research for any new product design should start with an understanding of the business and technical context surrounding the product. In almost all cases, the reason a product is being designed (or redesigned) is to achieve one or several specific business outcomes (most commonly, to make money). It is the designers' obligation to develop solutions without ever losing sight of these business goals. Therefore, it is critical that the design team begin its work by understanding the opportunities and constraints that are behind the design brief.

As Donald Schön so aptly puts it, "design is a conversation with materials."[1] This means that, for a designer to craft an appropriate solution, he must understand the capabilities and limitations of the "materials" that will be used to construct the product, whether they be lines of code or extruded plastic. The best way to begin this understanding is by talking to the people responsible for managing and building the product.

Generally speaking, a *stakeholder* is anyone with authority and/or responsibility for the product being designed. More specifically, stakeholders are key members of the organization commissioning the design work. Typically they include executives, managers, and representative contributors from development, sales, product management, marketing, customer support, design, and usability. They may also include similar people from other organizations in business partnerships with the commissioning organization.

Interviews with stakeholders should occur *before* user research begins. Stakeholder discussions often inform how user research is conducted.

It is most effective to interview each stakeholder in isolation, rather than in a larger, cross-departmental group. A one-on-one setting promotes candor on the part of the stakeholder and ensures that individual views are not lost in a crowd. (One of the most interesting things you can discover in such interviews is the extent to which everyone in a product team shares—or doesn't share—a common vision.) Interviews need not last longer than about an hour. Follow-up meetings may be needed if a particular stakeholder is identified as an exceptionally valuable source of information.

Certain types of information are important to gather from stakeholders:

- **Preliminary product vision**—As in the fable of the blind men and the elephant, you may find that each business department has a slightly different and slightly incomplete perspective on the product to be designed. Part of the design approach therefore must involve harmonizing these perspectives with those of users and customers. If there is a serious disconnect in vision among stakeholders, that situation is a yellow flag to monitor and follow up on early in the process.

- **Budget and schedule**—Discussions on this topic often provide a reality check on the scope of the design effort and give management a decision point if user research indicates that a greater (or lesser) scope is required.

- **Technical constraints and opportunities**—Another important determinant of design scope is a firm understanding of what is technically feasible given budget, time, and technology constraints. It is also often the case that a product is being developed to capitalize on a new technology. Understanding the opportunities underlying this technology can help shape the product's direction.

- **Business drivers**—It is important for the design team to understand what the business is trying to accomplish. This again leads to a decision point, should user research indicate a conflict between business and user needs. The design must, as much as possible, create a win-win situation for users, customers, and providers of the product.

- **Stakeholders' perceptions of their users**—Stakeholders who have relationships with users (such as customer support representatives) may have important insights that will help you formulate your user research plan. You may also find that there are significant disconnects between some stakeholders' perceptions of their users and what you discover in your research. This information can become an important discussion point with management later in the process.

Understanding these issues and their impact on design solutions helps you as a designer better develop a successful product. Regardless of how desirable your designs are to customers and users, if you don't consider the viability and feasibility of the proposed solution, it's unlikely that the product will thrive.

Discussing these topics is also important to developing a common language and understanding among the design, management, and engineering teams. As a designer, your job is to develop a vision that the entire team believes in. If you don't take the time to understand everyone's perspective, it's unlikely that they will feel that proposed solutions reflect their priorities. Because these people have the responsibility and authority to deliver the product to the real world, they are guaranteed to have important knowledge and opinions. If you don't ask for it upfront, it is likely to be forced on you later, often in the form of a critique of your proposed solutions.

Keep in mind that although perspectives gathered from stakeholders are obviously important, you shouldn't accept them at face value. As you will later discover in user interviews, some people may express problems by trying to propose solutions. It's the designer's job to read between the lines of these suggestions, root out the real problems, and propose solutions appropriate to both the business and the users.

Subject matter expert (SME) interviews

Early in a design project, it is often invaluable to identify and meet with several *subject matter experts* (SMEs)—authorities on the domain within which the product will operate. This is of critical importance in domains that are highly complex or very technical, or for which legal considerations exist. (The healthcare domain touches all three of these points.)

Many SMEs were users of the product or its predecessors at one time and may now be trainers, managers, or consultants. Often they are experts hired by stakeholders, rather than stakeholders themselves. Similar to stakeholders, SMEs can provide valuable perspectives on a product and its users. But designers should be careful to recognize that SMEs represent a somewhat skewed perspective because often, by necessity, they are invested in their understanding of the product/domain as it currently exists. This in-depth knowledge of product quirks and domain limitations can be at once a boon and a hindrance to innovative design.

Here are some other points to consider about using SMEs:

- **SMEs are often expert users.** Their long experience with a product or its domain means that they may have grown accustomed to current interactions. They may also lean toward expert controls rather than interactions designed for perpetual intermediates. (To understand the importance of this consideration, see Chapter 10.) SMEs often are not current users of the product and may have more of a management perspective.

- **SMEs are knowledgeable, but they aren't designers.** They may have many ideas on how to improve a product. Some of these may be valid and valuable, but the most useful pieces of information to glean from these suggestions are the causative *problems* that lead to their proposed solutions. As with users, when you encounter a proposed solution, ask how it would help you or the user.

- **SMEs are necessary in complex or specialized domains.** If you are designing for a technical domain such as medical, scientific, or financial services, you will likely need some guidance from SMEs, unless you are one yourself. Use SMEs to gather information on industry best practices and complex regulations. SME knowledge of user roles and characteristics is critical for planning user research in complex domains.

- **You will want access to SMEs throughout the design process.** If your product domain requires the use of SMEs, you will need to bring them in at different stages of the design to help perform reality checks on design details. Make sure that you secure this access in your early interviews.

Customer interviews

It is easy to confuse users with customers. With consumer products, customers are often the same as users, but in corporate or technical domains, users and customers rarely describe the same sets of people. Although both groups should be interviewed, each has its own perspective on the product that needs to be factored quite differently into an eventual design.

Customers of a product are those who decide to purchase it. For consumer products, customers frequently are also users of the product. For products aimed at children or teens, the customers are parents or other adult supervisors of children. In the case of most enterprise, medical, or technical products, the customer is someone very different from the user—often an executive or IT manager—with distinct goals and needs. It's important to understand customers and satisfy their goals to make a product *viable*. It is also important to realize that customers seldom actually use the product themselves, and when they do, they use it quite differently from how their users do.

When interviewing customers, you will want to understand the following:

- Their goals in purchasing the product
- Their frustrations with current solutions
- Their decision process for purchasing a product of the type you're designing
- Their role in installing, maintaining, and managing the product
- Domain-related issues and vocabulary

Like SMEs, customers may have many opinions about how to improve the product's design. It is important to analyze these suggestions, as in the case of SMEs, to determine what issues or problems underlie the ideas offered, because better, more integrated solutions may become evident later in the design process.

User interviews

Users of a product should be the main focus of the design effort. They are the people who personally use the product to accomplish a goal (not their managers or support team). If you are redesigning or refining an existing product, it is important to speak to both current and *potential users*. These are people who do not currently use the product but who are good candidates for using it in the future because they have needs that the product can meet and are in the target market for the product. Interviewing both current and potential users illuminates the effect that experience with the current version of a product may have on how the user behaves and thinks about things.

Here is some information we are interested in learning from users:

- The context of how the product (or analogous system, if no current product exists) fits into their lives or work flow: when, why, and how the product is or will be used

- Domain knowledge from a user perspective: What do users need to know to do their jobs?

- Current tasks and activities: both those the current product is required to accomplish and those it doesn't support

- Goals and motivations for using their product

- Mental model: how users think about their jobs and activities, as well as what expectations users have about the product

- Problems and frustrations with current products (or an analogous system if no current product exists)

User observation

Most people are incapable of accurately assessing their own behaviors,[2] especially when these behaviors are removed from the context of people's activities. It is also true that out of fear of seeming dumb, incompetent, or impolite, many people may avoid talking about software behaviors that they find problematic or incomprehensible.

It then follows that interviews performed outside the context of the situations the designer hopes to understand will yield less-complete and less-accurate data. You can talk to users about how they think they behave, or you can observe their behavior first-hand. The latter route provides superior results.

Perhaps the most effective technique for gathering qualitative user data combines interviewing and observation, allowing the designers to ask clarifying questions and direct inquiries about situations and behaviors they observe in real time.

Many usability professionals use technological aides such as audio or video recorders to capture what users say and do. Interviewers must take care not to make these technologies too obtrusive; otherwise, the users will be distracted and behave differently than they would when not being recorded. In our practice, we've found that a notebook and a digital camera allow us to capture everything we need without compromising the honest exchange of information. Typically, we don't bring out the camera until we feel that we've established a good rapport with the interview subject. Then we use it to capture elements and artifacts in the environment that are difficult to jot down in our notes. Video, when used with care, can sometimes be a powerful rhetorical tool for achieving stakeholder buy-in to contentious or surprising research results. Video may also prove useful in situations where note-taking is difficult, such as in a moving car.

For consumer products, it can be hard to get a true picture of people's behaviors, especially if they will be using the product outside or in public. For these kinds of projects, a *man-on-the-street* approach to user observation can be effective. Using this approach, the design team casually observes product-related behaviors of people in public spaces. This type of technique is useful for understanding brick-and-mortar commerce-related behaviors that may translate into online behaviors, mobile-related behaviors of all sorts, or behaviors associated with specialized types of environments, such as theme parks and museums.

Interviewing and Observing Users

Drawing on years of design research in practice, we believe that a combination of observation and one-on-one interviews is the most effective and efficient tool in a designer's arsenal for gathering qualitative data about users and their goals. The technique of *ethnographic interviews* is a combination of immersive observation and directed interview techniques.

Hugh Beyer and Karen Holtzblatt pioneered an ethnographic interviewing technique they call *contextual inquiry*. Their method has, for good reason, rapidly gained traction in the industry and provides a sound basis for qualitative user research. It is described in detail in the first four chapters of their book, *Contextual Design* (Morgan Kaufmann, 1998). Contextual inquiry methods closely parallel the methods described here, but with some subtle and important differences.

Contextual inquiry

Contextual inquiry, according to Beyer and Holtzblatt, is based on a *master-apprentice model* of learning: observing and asking questions of the user as if she is the master craftsman, and the interviewer the new apprentice. Beyer and Holtzblatt also enumerate four basic principles of engaging in ethnographic interviews:

- **Context**—Rather than interviewing the user in a clean white room, it is important to interact with and observe the user in her normal work environment, or whatever physical context is appropriate for the product. Observing users as they perform activities and questioning them in their own environments, filled with the artifacts they use each day, can bring to light the all-important details of their behaviors.

- **Partnership**—The interview and observation should take the tone of a collaborative exploration with the user, alternating between observation of work and discussion of its structure and details.

- **Interpretation**—Much of the designer's work is reading between the lines of facts gathered about users' behaviors, their environment, and what they say. The designer

must take these facts together as a whole and analyze them to uncover the design implications. Interviewers must be careful, however, to avoid assumptions based on their own interpretation of the facts without verifying these assumptions with users.

- **Focus**—Rather than coming to interviews with a set questionnaire or letting the interview wander aimlessly, the designer needs to subtly direct the interview so as to capture data relevant to design issues.

Improving on contextual inquiry

Contextual inquiry forms a solid theoretical foundation for qualitative research, but as a specific method it has some limitations and inefficiencies. The following process improvements, in our experience, result in a more highly leveraged research phase that better sets the stage for successful design:

- **Shorten the interview process.** Contextual inquiry assumes full-day interviews with users. The authors have found that interviews as short as one hour can be sufficient to gather the necessary user data, provided that a sufficient number of interviews (about six well-selected users for each hypothesized role or type) are scheduled. It is much easier and more effective to find a diverse set of users who will consent to an hour with a designer than it is to find users who will agree to spend an entire day.

- **Use smaller design teams.** Contextual inquiry assumes a large design team that conducts multiple interviews in parallel, followed by debriefing sessions in which the full team participates. We've found that it is more effective to conduct interviews sequentially with the same designers in each interview. This allows the design team to remain small (two or three designers). But even more importantly, it means that the entire team interacts with all the interviewed users directly, allowing the members to most effectively analyze and synthesize the user data.

- **Identify goals first.** Contextual inquiry feeds a design process that is fundamentally task-focused. We propose that ethnographic interviews identify and prioritize user goals before determining the tasks that relate to these goals.

- **Look beyond business contexts.** The vocabulary of contextual inquiry assumes a business product and a corporate environment. Ethnographic interviews are also possible in consumer domains, although the focus of questioning is somewhat different, as discussed next.

Preparing for ethnographic interviews

Ethnography is a term borrowed from anthropology; it means the systematic and immersive study of human cultures. In anthropology, ethnographic researchers spend years living in the cultures they study and record. Ethnographic interviews take the spirit of

this type of research and apply it on a micro level. Rather than trying to understand behaviors and social rituals of an entire culture, the goal is to understand the behaviors and rituals of people interacting with individual products.

Identifying candidates

Because the designers must capture an entire range of user behaviors regarding a product, it is critical that the designers identify an appropriately diverse sample of users and user types when planning a series of interviews. Based on information gleaned from stakeholders, SMEs, and literature reviews, designers need to create a hypothesis that serves as a starting point in determining what sorts of users and potential users to interview.

The persona hypothesis

We label this starting point the *persona hypothesis*, because it is the first step toward identifying and synthesizing personas—the behavioral user models we will discuss in detail in the next chapter. The persona hypothesis should be based on likely behavior patterns and the factors that differentiate these patterns, not purely on demographics. It is often the case with consumer products that demographics are used as screening criteria to select interview subjects. But even in this case, they should serve as a proxy for a hypothesized behavior pattern.

The nature of a product's domain makes a significant difference in how a persona hypothesis is constructed. Business users are often quite different from consumer users in their behavior patterns and motivations, and different techniques are used to build the persona hypothesis in each case.

The persona hypothesis is a first try at defining the different kinds of users (and sometimes customers) for a product. The hypothesis is the basis for initial interview planning; as interviews proceed, new interviews may be required if the data indicates the existence of user types not originally identified.

The persona hypothesis attempts to address, at a high level, these three questions:

- What different sorts of people might use this product?
- How might their needs and behaviors vary?
- What ranges of behavior and types of environments need to be explored?

Roles in business and consumer domains

For business products, *roles*—common sets of tasks and information needs related to distinct classes of users—provide an important initial organizing principle. For example, for an office phone system, we might find these rough roles:

- People who make and receive calls from their desks
- People who travel a lot and need to access the phone system remotely
- Receptionists who answer the phone for many people
- People who technically administer the phone system

In business and technical contexts, roles often map roughly to job descriptions. Therefore, it is relatively easy to get a reasonable first cut of user types to interview by understanding the kinds of jobs held by users (or potential users) of the system.

Unlike business users, consumers don't have concrete job descriptions, and their use of products may cross multiple contexts. Therefore, it often isn't meaningful to use roles as an organizing principle for the persona hypothesis for a consumer product. Rather, it is often the case that you will see the most significant patterns emerge from users' attitudes and aptitudes, lifestyle choices, or stage of life, all of which can influence their behaviors.

Behavioral and demographic variables

In addition to roles, a persona hypothesis should be based on variables that help differentiate between various kinds of users based on their needs and behaviors. This is often the most useful way to distinguish between different types of users (and it forms the basis for the persona-creation process described in the next chapter). Despite the fact that these variables can be difficult to fully anticipate without research, they often become the basis of the persona hypothesis for consumer products. For example, for an online store, we might identify several ranges of behavior concerning shopping:

- Frequency of shopping (from frequent to infrequent)
- Desire to shop (from loves to shop to hates to shop)
- Motivation to shop (from bargain hunting to searching for just the right item)

Although consumer user types can often be roughly defined by the combination of behavioral variables they map to, behavioral variables are also important for identifying types of business and technical users. People within a single business-role definition may have different needs and motivations. Behavioral variables can capture this, although often not until user data has been gathered.

Given the difficulty in accurately anticipating behavioral variables before user data is gathered, another helpful approach in building a persona hypothesis is making use of *demographic variables*. When planning your interviews, you can use market research to identify ages, locations, gender, and incomes of the target markets for the product. Interviewees should be distributed across these demographic ranges in the hope of interviewing a sufficiently diverse group of people to identify the significant behavior patterns.

Domain expertise versus technical expertise

One important type of behavioral distinction is the difference between technical expertise (knowledge of digital technology) and domain expertise (knowledge of a specialized subject area pertaining to a product). Different users will have varying amounts of technical expertise. Similarly, some users of a product may be less expert in their knowledge of the product's domain (for example, accounting knowledge in the case of a general ledger application). Thus, depending on who the design target of the product is, domain support may be a necessary part of the product's design, as well as technical ease of use. A relatively naïve user will likely never be able to use more than a small subset of a domain-specific product's functions without domain support provided in the interface. If naïve users are part of the target market for a domain-specific product, care must be taken to support domain-naïve behaviors.

Environmental considerations

A final consideration, especially in the case of business products, is the cultural differences between organizations in which the users are employed. At small companies, for example, workers tend to have a broader set of responsibilities and more interpersonal contact. Huge companies often have multiple layers of bureaucracy, and their workers tend to be highly specialized. Here are some examples of these environmental variables:

- Company size (from small to multinational)
- Company location (North America, Europe, Asia, and so on)
- Industry/sector (electronics manufacturing, consumer packaged goods, and so on)
- IT presence (from ad hoc to draconian)
- Security level (from lax to tight)

Like behavioral variables, these may be difficult to identify without some domain research, because patterns vary significantly by industry and geographic region.

Putting together a plan

After you have created a persona hypothesis, complete with potential roles and behavioral, demographic, and environmental variables, you need to create an interview plan that can be communicated to the person in charge of coordinating and scheduling the interviews.

In our practice, we've observed that, for enterprise or professional products, each presumed behavioral pattern requires about a half-dozen interviews to verify or refute (occasionally more if a domain is particularly complex). What this means in practice is that each identified role, behavioral variable, demographic variable, and environmental variable identified in the persona hypothesis should be explored in four to six interviews (or occasionally more, as mentioned earlier).

However, these interviews can overlap. Suppose we believe that use of an enterprise product may differ by geographic location, industry, and company size. Research at a single small electronics manufacturer in Taiwan would allow us to cover several variables at once. By being clever about mapping variables to interviewee-screening profiles, you can keep the interviews to a manageable number.

Consumer products typically have much more variation in behavior, so more interviews typically are required to really delineate the differences. A good rule of thumb is to double the numbers just discussed: 8 to 12 interviews for each user type postulated in the persona hypothesis. As before, a complex consumer product may occasionally require even more interviews to accurately capture the range of behaviors and motivations. Something to keep in mind with consumer products is that lifestyle choices and *life stages* (single, married, young children, older children, empty nest) can multiply the interviews for certain kinds of products.

Conducting ethnographic interviews

After the persona hypothesis has been formulated and an interview plan has been derived from it, you are ready to interview—assuming you get access to interviewees! While formulating the interview plan, designers should work closely with project stakeholders who have access to users. Stakeholder involvement generally is the best way to make interviews happen, especially for business and technical products.

For enterprise or technical products, it is often straightforward to work with stakeholders to help you get in touch with a robust set of interviewees. This can be more challenging for consumer products (especially if the company you are working with doesn't currently have a good relationship with its users).

If stakeholders can't help you get in touch with users, you can contact a market or usability research firm that specializes in finding people for surveys and focus groups. These firms are useful for reaching consumers with diverse demographics. The difficulty with this approach is that it can sometimes be challenging to find interviewees who will permit you to interview them in their homes or places of work.

As a last alternative for consumer products, designers can recruit friends and relatives. This makes it easier to observe the interviewees in a natural environment but also is quite limiting as far as diversity of demographic and behavioral variables are concerned.

Interview teams and timing

The authors favor a team of two designers per interview. The **moderator** drives the interview and takes light notes, and the **facilitator** takes detailed notes and looks for any holes in the questioning. These roles can switch halfway through the interview if the team agrees. One hour per user interviewed is often sufficient, except in the case of complex domains such as medical, scientific, and financial services. These may require more time to fully understand what the user is trying to accomplish. Be sure to budget travel time between interview sites. This is especially true of consumer interviews in residential neighborhoods, or interviews that involve "shadowing" users as they interact with a (usually mobile) product while moving from place to place. Teams should try to limit interviews to six per day. This will give them adequate time for debriefing and strategizing between interviews and will keep the interviewers from getting fatigued.

Phases of ethnographic interviews

A complete set of ethnographic interviews for a project can be grouped into three distinct, chronological phases. The approach of the interviews in each successive phase is subtly different from the previous one, reflecting the growing knowledge of user behaviors that results from each additional interview. Focus tends to be broad at the start, aimed at gross structural and goal-oriented issues, and more narrow for interviews at the end of the cycle, zooming in on specific functions and task-oriented issues.

- **Early interviews** are exploratory in nature and focus on gathering domain knowledge from the user's point of view. Broad, open-ended questions are common, with a lesser degree of drilldown into details.
- **Middle interviews** are where designers begin to see patterns of use and ask open-ended and clarifying questions to help connect the dots. Questions in general are more focused on domain specifics, now that the designers have absorbed the domain's basic rules, structures, and vocabularies.

- **Late interviews** confirm previously observed patterns, further clarifying user roles and behaviors and making fine adjustments to assumptions about task and information needs. More closed-ended questions are used, tying up loose ends in the data.

After you have an idea who your interviewees will be, it can be useful to work with stakeholders to schedule individuals most appropriate for each phase in the interview cycle. For example, in a complex, technical domain it is often a good idea to perform early interviews with more patient and articulate interview subjects. In some cases, you may also want to loop back and interview this particularly knowledgeable and articulate subject again at the end of the interview cycle to address any topics you were unaware of during your initial interview.

Basic methods

The basic methods of ethnographic interviewing are simple, straightforward, and very low-tech. Although the nuances of interviewing subjects take some time to master, any practitioner should, if he or she follows these suggestions, be rewarded with a wealth of useful qualitative data:

- Interview where the interaction happens.
- Avoid a fixed set of questions.
- Assume the role of an apprentice, not an expert.
- Use open-ended and closed-ended questions to direct the discussion.
- Focus on goals first and tasks second.
- Avoid making the user a designer.
- Avoid discussing technology.
- Encourage storytelling.
- Ask for a show-and-tell.
- Avoid leading questions.

We describe each of these methods in more detail in the following sections.

Interview where the interaction happens

Following the first principle of contextual inquiry, it is of critical importance that subjects be interviewed in the places where they actually use the products. Not only does this allow the interviewers to witness the product being used, but it also gives the interview

team access to the environment in which the interaction occurs. This can give you tremendous insight into product constraints and user needs and goals.

Observe the environment closely: It is likely to be crawling with clues about tasks the interviewee might not have mentioned. Notice, for example, the kind of information he needs (papers on his desk or adhesive notes on his screen border), inadequate systems (cheat sheets and user manuals), the frequency and priority of tasks (inbox and outbox), and the kind of work flows he follows (memos, charts, calendars). Don't snoop without permission, but if you see something that looks interesting, ask your interviewee to discuss it.

Avoid a fixed set of questions

If you approach ethnographic interviews with a fixed questionnaire, you not only run the risk of alienating the interview subject, but you also can cause the interviewers to miss out on a wealth of valuable user data. The entire premise of ethnographic interviews (and contextual inquiry) is that we as interviewers don't know enough about the domain to presuppose the questions that need asking. We must learn what is important from the people we talk to. That said, it's certainly useful to have *types* of questions in mind. Depending on the domain, it may also be useful to have a standardized set of *topics* you want to be sure to cover during your interview. This list of topics may evolve over the course of your interviews, but it will help you ensure that you gather enough details from each interview that you can recognize significant behavior patterns.

Here are some **goal-oriented questions** to consider:

- **Goals**—What makes a good day? A bad day?
- **Opportunity**—What activities currently waste your time?
- **Priorities**—What is most important to you?
- **Information**—What helps you make decisions?

Another useful type of question is the **system-oriented question**:

- **Function**—What are the most common things you do with the product?
- **Frequency**—What parts of the product do you use most?
- **Preference**—What are your favorite aspects of the product? What drives you crazy?
- **Failure**—How do you work around problems?
- **Expertise**—What shortcuts do you employ?

For business products, **work flow-oriented questions** can be helpful:

- **Process**—What did you do when you first came in today? What did you do after that?
- **Occurrence and recurrence**—How often do you do this? What things do you do weekly or monthly, but not every day?
- **Exception**—What constitutes a typical day? What would be an unusual event?

To better understand user motivations, you can employ **attitude-oriented questions**:

- **Aspiration**—What do you see yourself doing five years from now?
- **Avoidance**—What would you prefer not to do? What do you procrastinate on?
- **Motivation**—What do you enjoy most about your job (or lifestyle)? What do you always tackle first?

Assume the role of an apprentice, not an expert

During your interviews, you want to shed your expert designer or consultant cap and take on the role of the apprentice. Your goal is to omnivorously and nonjudgmentally absorb everything your interviewees want to tell you and to encourage them to actively engage in detailed, thoughtful explanations. Don't be afraid to ask naïve questions. They put people at ease, and you'd be surprised how often seemingly silly questions end up pushing past assumptions and lead to real insights. Be a sympathetic and receptive listener, and you'll find that people will be willing to share almost any kind of information with you.

Use open-ended and closed-ended questions to direct the discussion

As Kim Goodwin discusses in her excellent book, *Designing for the Digital Age* (Wiley, 2009), the use of open-ended and closed-end questions allows interviewers to help keep user interviews productive and on the right track.

Open-ended questions encourage detailed responses from the interviewee. Use these types of questions to elicit more detail about a topic on which you need to gather more information. Typical open-ended questions begin with "Why," "How," or "What."

Closed-ended questions encourage a brief response. Use closed-ended questions to shut down a line of inquiry or to get an interviewee back on track if he has begun to take the interview in an unproductive direction. Closed-ended questions generally expect a yes or no answer and typically begin with "Did you," "Do you," or "Would you."

After an interviewee answers your closed-ended question, usually a natural pause in the conversation occurs. Then you can redirect the discussion to a new line of questioning using a new open-ended question.

Focus on goals first and tasks second

Unlike contextual inquiry and the majority of other qualitative research methods, the first priority of ethnographic interviewing is understanding the *why* of users. You need to know what motivates the behaviors of individuals in different roles and *how* they hope to ultimately accomplish this goal, not the *what* of the tasks they perform. Understanding the tasks is important, and the tasks must be recorded diligently. But these tasks will ultimately be restructured to better match user goals in the final design.

Avoid making the user a designer

Guide the interviewee toward examining problems and away from expressing solutions. Most of the time, those solutions reflect the interview subject's personal priorities. Although they sound good to *him*, they tend to be shortsighted and idiosyncratic. They also lack the balance and refinement that an interaction designer can bring to a solution based on adequate research and years of experience. That said, a proposed design solution can be a useful jumping-off point to discuss a user's goals and the problems he encounters with the current systems. If a user blurts out an interesting idea, ask, "What problem would that solve for you?" or "Why would that be a good solution?"

Avoid discussing technology

Just as you don't want to treat the user like a designer, you also don't want to treat him like a software engineer. Discussing technology is meaningless without first understanding the purpose underlying any technical decisions. In the case of technical or scientific products, where technology is always an issue, distinguish between domain-related technology and product-related technology, and steer away from the latter. If an interview subject insists on talking about how the product should be implemented, return to his goals and motivations by asking, "How would that help you?"

Encourage storytelling

Far more useful than asking users for design advice is encouraging them to tell specific stories about their experiences with a product (whether an old version of the one you're redesigning, or an analogous product or process). Ask them how they use it, what they think of it, who else they interact with when using it, where they go with it, and so forth. Detailed stories of this kind usually are the best way to understand how users relate to

and interact with products. Encourage stories that deal with typical cases and also more exceptional ones.

Ask for a show-and-tell

After you have a good idea of the flow and structure of a user's activities and interactions, and you have exhausted other questions, it is often useful to ask the interviewee for a show-and-tell or *grand tour* of artifacts related to the design problem. These can be domain-related artifacts, software interfaces, paper systems, tours of the work environment, or ideally all of these. Be sure not to just record the artifacts themselves (digital or video cameras are handy at this stage); also pay attention to *how* the interviewee describes them. Be sure to ask plenty of clarifying questions as well.

As you look to capture artifacts from the user's environment, be on particular lookout for signs of unmet needs or failures in the existing design. For example, in one of our past projects, we were engaged to redesign the software interface as an expensive piece of scientific equipment. As we interviewed users—who were primarily chemists—we observed that next to every machine we looked at was a notebook filled with details of every experiment run on that machine: which scientist did it, what it was about, and when it was run. It turned out that although the equipment recorded detailed results for each experiment, it recorded no other information about the experiment other than a sequential ID number. If the paper notebook were lost, the experimental results would be nearly useless, because the ID number meant nothing to the users. This was clearly an unmet user need!

Avoid leading questions

One important thing to avoid in interviews is the use of *leading questions*. Just as in a courtroom, where lawyers can, by virtue of their authority, bias witnesses by suggesting answers to them, designers can inadvertently bias interview subjects by implicitly (or explicitly) suggesting solutions or opinions about behaviors. Here are some examples of leading questions:

- Would feature X help you?
- You like X, don't you?
- Do you think you'd use feature X if it were available?
- Does X seem like a good idea to you?

After the interviews

After each interview, teams compare notes and discuss any particularly interesting trends observed or specific points brought up in the most recent interview. If they have the time, they should also look back at old notes to see whether unanswered questions from other interviews and research have been answered properly. This information should be used to strategize about the approach to take in subsequent interviews.

After the interview process is finished, the design team should make a pass through all the notes, marking or highlighting trends and patterns in the data. This is very useful for the next step of creating personas from the cumulative research. This marking, when it also involves organizing the marked responses into topic groups, is called *coding* by serious ethnographers. Although that level of organization usually is overkill, it may be useful in particularly complex or nuanced domains.

If it is helpful, the team can create a binder of the notes, review any video recordings, and print artifact images to place in the binder or on a public surface, such as a wall, where they are all visible simultaneously. This will be useful in later design phases.

Other Types of Qualitative Research

The majority of this chapter has focused on the qualitative research techniques we employ to construct the robust user and domain models described in the next chapter. Design and usability professionals incorporate many other forms of research, ranging from detailed task analysis to focus groups and usability tests. While these activities may indeed contribute to the creation of useful and desirable products, we have found that the Goal-Directed approach described earlier in this chapter provides the most value to the process of digital product design. Put simply, the Goal-Directed approach helps answer questions about the product at both the big-picture and functional-detail level with a relatively small amount of effort and expense. No other research technique we're aware of can claim this.

If you're interested in exploring alternative design research techniques, *Observing the User Experience* (Morgan Kaufmann, 2012) by Elizabeth Goodman, Mike Kuniavsky, and Andrea Moed is an excellent resource. It describes a wide range of user research methods that can be used at many points in the design and development process.

In the remainder of this chapter, we'll discuss a few of the more prominent of these research methods and how they can fit into the overall design and development effort.

Focus groups

Marketing organizations are particularly fond of gathering user data via *focus groups*. Representative users, usually chosen to match previously identified demographic segments of the target market, are gathered in a room and asked a structured set of questions and provided a structured set of choices. Often the meeting is recorded on audio or video media for later reference. Focus groups are a standard technique in traditional product marketing. They are also quite useful for gauging initial reactions to a product's *form*—its visual appearance or industrial design. Focus groups can also gather reactions to a product that the respondents have been using for some time.

Although focus groups may appear to provide the requisite user contact, the method is in many ways inappropriate as an interaction design tool. Focus groups excel at eliciting information about products that people own or are willing (or unwilling) to purchase, but they are weak at gathering data about what people actually do with those products, or how and why they do it. Also, because they are a group activity, focus groups tend to end up at consensus. The majority or loudest opinion often becomes the group opinion. This is counterproductive for an interaction design process, where designers must understand all the different patterns of behavior a product must address. Focus groups tend to stifle exactly the diversity of behavior and opinion that interaction designers must accommodate.

Usability testing

Usability testing (also known, somewhat unfortunately, as "user testing") is a collection of techniques used to measure characteristics of a user's interaction with a product. The goal usually is to assess that product's usability. Typically, usability testing is focused on measuring how well users can complete specific, standardized tasks, as well as what problems they encounter in doing so. Results often reveal areas where users have problems understanding and using the product, as well as places where users are more likely to be successful.

Usability testing requires a fairly complete and coherent design artifact to test against. Whether you are testing production software, a clickable prototype, or even a paper prototype, the point of the test is to validate a product design. This means that the appropriate place for usability testing is quite late in the design cycle, after there is a coherent concept and sufficient detail to generate such prototypes. We discuss evaluative usability testing as part of design refinement in Chapter 5.

A case could be made for usability testing at the beginning of a redesign effort. The technique is certainly capable of finding opportunities for improvement in such a project. However, we find that we can better assess product inadequacies through our qualitative

studies. Perhaps the budget is limited in a way that allows usability testing only once in a product design initiative. If so, we find much more value in performing the tests after we have a candidate solution, as a means of testing the specific elements of the new design.

Card sorting

Popularized by information architects, card sorting is a technique to understand how users organize information and concepts. Although the technique has a number of variations, it is typically performed by asking users to sort a deck of cards, each containing a piece of functionality or information related to the product or website. The tricky part is analyzing the results, either by looking for trends or using statistical analysis to uncover patterns and correlations.

While this can be a valuable tool to uncover one aspect of a user's mental model, the technique assumes that the subject has refined organizational skills and that how he sorts a group of abstract topics will correlate to how he will end up wanting to use your product. In our experience, this is not always the case.

One way to overcome these potential challenges is to ask the users to sequence the cards based on the completion of tasks that the product is being designed to support. Another way to enhance the value of a card sort study is to debrief the subject afterwards to understand any organizational principles he has employed in his sort (again, attempting to understand his mental model).

Ultimately, we believe that properly conducted open-ended interviews can explore these aspects of the user's mental model more effectively. By asking the right questions and paying close attention to how a subject explains his activities and the domain, you can decipher how he mentally associates different bits of functionality and information.

Task analysis

Task analysis refers to a number of techniques that involve using either questionnaires or open-ended interviews to develop a detailed understanding of how people currently perform specific tasks. Such a study is concerned with the following:

- Why the user is performing the task (that is, the underlying goal)
- The frequency and importance of the task
- Cues—what initiates or prompts the execution of the task
- Dependencies—what must be in place to perform the task, as well as what is dependent on the completion of the task
- People who are involved, and their roles and responsibilities

- Specific actions that are performed

- Decisions that are made

- Information that is used to support decisions

- What goes wrong—errors and exception cases

- How errors and exceptions are corrected

Once the questionnaires are compiled or the interviews are completed, tasks are formally decomposed and analyzed. Typically they are incorporated into a flowchart or similar diagram that communicates the relationships between actions and often the relationships between people and processes.

We've found that this type of inquiry should be incorporated into ethnographic user interviews. Furthermore, as we'll discuss in the next chapter, the analysis activities are a useful part of our modeling activities. Task analysis is a critical way of understanding how users *currently* do something, as well as identifying pain points and opportunities for improvement. However, task analysis does little to illuminate users' *goals*. How people do things today is often a product of the obsolete systems and organizations they are forced to interact with. How people do things often bears little resemblance to how they would *like* to do things or how they can be most effective.

Research Is Critical to Good Design

User research is the critical foundation on which your designs are built. Take the time to plan your user research and match the appropriate technique to the appropriate place in your development cycle. Your product will benefit, and you'll avoid wasting time and resources. Putting a product to the test in a lab to see whether it passes or fails may provide a lot of data, but not necessarily a lot of value. Using ethnographic interviews at the beginning of the process allows you, as a designer, to truly understand your users, their needs, and their motivations. Once you have a solid design concept based on qualitative user research and the models that research feeds, your usability testing will become an even more efficient tool for judging the effectiveness of design choices you have made. Goal-Directed design research allows you to do the heavy lifting early in the process.

Notes
1. Schön, D. and Bennett, J., 1996
2. Pinker, 1999

MODELING USERS: PERSONAS AND GOALS

Once you have spent some time out in the field investigating your users' lives, motivations, and environs, the question naturally arises: How do you use all this great research data to forge a successful product design? You have notebooks full of conversations and observations, and it is very likely that each person you spoke to was slightly different from the others. It is hard to imagine digging through hundreds of pages of notes every time you need to make a design decision. Even if you had the time to do so, it isn't entirely obvious how these notes should inform your thinking. How do you make sense of them and prioritize?

We solve this problem by applying the powerful concept of a *model*.

Why Model?

Models are used in the natural and social sciences to represent complex phenomena with a useful abstraction. Good models emphasize the salient features of the structures and relationships they represent and de-emphasize the less significant details. Because we are designing for users, it is important that we can understand and visualize the salient aspects of their relationships with each other, what they want, with their social and physical environments, and, of course, with the products we hope to design.

Thus, much as economists create models to describe the behavior of markets, and physicists create models to describe the behavior of subatomic particles, we have found that using our research to create descriptive models of users is a uniquely powerful tool for interaction design. We call these user models **personas**.

Personas provide us with a precise way of thinking and communicating about how groups of users behave, how they think, what they want to accomplish, and why. Personas are not real people, but they are assembled from the behaviors and motivations of the many actual users we encounter in our research. In other words, personas are *composite archetypes* based on *behavior patterns* uncovered during the course of our research, which we formalize for the purpose of informing the product design. By using personas, we can develop an understanding of our users' goals in specific contexts—a critical tool for ideating and validating design concepts.

Personas, like many powerful tools, are simple in concept but must be applied with considerable nuance and sophistication. It is not enough to whip up a couple of user profiles based on stereotypes and generalizations, nor is it particularly useful to attach a stock photograph to a job title and call it a "persona." For personas to be effective tools for design, considerable rigor and finesse must be applied during the process of identifying the significant and meaningful patterns in user behavior and determining how these behaviors translate into archetypes that accurately represent the appropriate cross section of users.

Although other useful models can serve as tools for the interaction designer, such as work flow models and physical models, we've found that personas are the most effective and fundamental of these tools. Also, it is often possible to incorporate the best of these other modeling techniques into the structure of your personas.

This chapter focuses primarily on personas and their goals. Other models are considered briefly at the end of the chapter.

The Power of Personas

To create a product that must satisfy a diverse audience of users, logic might tell you to make its functionality as broad as possible to accommodate the most people. *This logic, however, is flawed.* The best way to successfully accommodate a variety of users is to design for *specific types of individuals with specific needs.*

When you broadly and arbitrarily extend a product's functionality to include many constituencies, you increase the cognitive load and navigational overhead for all users. Facilities that may please some users will likely interfere with the satisfaction of others, as shown in Figure 3-1.

Figure 3-1: If you try to design an automobile that pleases every possible driver, you end up with a car with every possible feature that pleases nobody. Software today is too often designed to please too many users, resulting in low user satisfaction. Figure 3-2 provides an alternative approach.

Alesandro's goals
• Go fast
• Have fun

Marge's goals
• Be safe
• Be comfortable

Dale's goals
• Haul big loads
• Be reliable
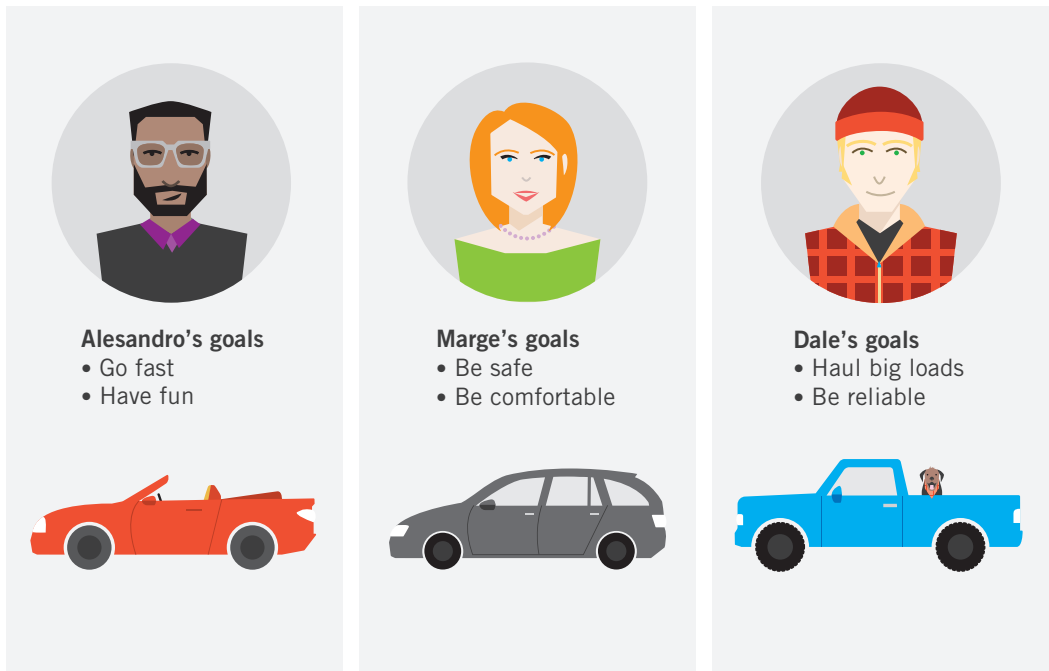
Figure 3-2: By designing different cars for different people with different specific goals, we can create designs that other people with needs similar to our target drivers also find satisfying. The same holds true for the design of digital products and software.

The key to this approach is to first choose the right individuals to design for—users whose needs best represent the needs of a larger set of key constituents (see Figure 3-2). Then you prioritize these individuals so that the needs of the most important users are met without compromising our ability to meet the needs of secondary users. Personas provide a powerful tool for communicating about different types of users and their needs and then deciding which users are the most important to target in the design of form and behavior.

Strengths of personas as a design tool

The persona is a powerful, multipurpose design tool that helps overcome several problems that plague the development of digital products. Personas help designers do the following:

- **Determine** what a product should do and how it should behave. Persona goals and tasks provide the foundation for the design effort.
- **Communicate** with stakeholders, developers, and other designers. Personas provide a common language for discussing design decisions and also help keep the design centered on users at every step in the process.
- **Build consensus and commitment** to the design. With a common language comes a common understanding. Personas reduce the need for elaborate diagrammatic models; it's easier to understand the many nuances of user behavior through the narrative structures that personas employ. Put simply, because personas resemble real people, they're easier to relate to than feature lists and flowcharts.
- **Measure** the design's effectiveness. Design choices can be tested on a persona in the same way that they can be shown to a real user during the formative process. Although this doesn't replace the need to test with real users, it provides a powerful reality-check tool for designers trying to solve design problems. This allows design iteration to occur rapidly and inexpensively at the whiteboard, and it results in a far stronger design baseline when it's time to test with actual people.
- **Contribute** to other product-related efforts such as marketing and sales plans. The authors have seen personas repurposed across their clients' organizations, informing marketing campaigns, organizational structure, customer support centers, and other strategic planning activities. Business units outside of product development want sophisticated knowledge of a product's users and typically view personas with great interest.

Design pitfalls that personas help avoid

Personas also can resolve three design issues that arise during product development:

- The elastic user
- Self-referential design
- Edge cases

The elastic user

Although satisfying the users of our products is our goal, the term *user* causes trouble when applied to specific design problems and contexts. Its imprecision makes it dangerous as a design tool, because every person on a product team has his own conceptions of who the user is and what the user needs. When it's time to make product decisions, this "user" becomes *elastic*, conveniently bending and stretching to fit the opinions and presuppositions of whoever is talking.

If the product development team finds it convenient to use a confusing tree control containing nested, hierarchical folders to provide access to information, they might define the user as a computer-literate "power user." Other times, when it is more convenient to step through a difficult process with a wizard, they define the user as an unsophisticated first-time user. Designing for the elastic user gives a product team license to build what it pleases, while still apparently serving "the user." Of course, our goal should be to design products that appropriately meet the needs of *real* users. Real users—and the personas representing them—are not elastic, but instead have specific requirements based on their goals, abilities, and contexts.

Even focusing on user roles or job titles rather than specific archetypes can introduce unproductive elasticity to the focus of design activities. For example, in designing clinical products, it might be tempting to lump together all nurses as having similar needs. However, if you have any experience in a hospital, you know that trauma nurses, pediatric intensive-care nurses, and operating room nurses are quite different from each other, each with their own attitudes, aptitudes, needs, and motivations. Nurses new to the role look at their work differently than nurses who are veteran to the job. A lack of precision about the user can lead to a lack of clarity about how the product should behave.

Self-referential design

Self-referential design occurs when designers or developers project their own goals, motivations, skills, and mental models onto a product's design. Many "cool" product designs fall into this category. The audience doesn't extend beyond people like the designer. This is fine for a narrow range of products but is inappropriate for most others. Similarly, developers apply self-referential design when they create implementation-model products. *They* understand perfectly how the data is structured and how software works and are comfortable with such products. Few non-developers would concur.

Edge cases

Another syndrome that personas help prevent is designing for edge cases—situations that might happen but that usually won't for most people. Typically, edge cases must be designed and programmed for, but they should never be the design *focus*. Personas provide

a reality check for the design. We can ask, "Will Julie want to perform this operation very often? Will she ever?" With this knowledge, we can prioritize functions with great clarity.

Why Personas Are Effective

Personas are user models that are represented as specific, individual human beings. As we've discussed, they are not actual people but are synthesized directly from research and observations of real people. One of the reasons personas are so successful as user models is that they are *personifications*[1]: They engage the *empathy* of the design and development team around the users' goals.

Empathy is critical for the designers, who will be making their decisions about design frameworks and details based on the persona's cognitive *and* emotional dimensions, as typified by the persona's goals. (We will discuss the important connections between goals, behaviors, and personas later in this chapter.) However, the power of empathy should not be discounted for other team members. Not only do personas help make our design solutions better at serving real user needs, but they also make these solutions more compelling to stakeholders. When personas have been carefully and appropriately crafted, stakeholders and engineers begin to think about them as if they are real human beings and become much more interested in creating a product that will give this person a satisfying experience.

We're all aware of the power of fictional characters in books, movies, and television programs to engage viewers. Jonathan Grudin and John Pruitt have discussed how this can relate to interaction design.[2] They also note the power of **Method acting** as a tool that actors use to understand and portray realistic characters. In fact, the process of creating personas from user observation, and then imagining and developing scenarios from the perspective of these personas, is in many ways analogous to Method acting. Our colleague Jonathan Korman used to call the Goal-Directed use of personas the Stanislavski Method of interaction design.

Personas are based on research

Personas, like any models, should be based on real-world observation. As discussed in the preceding chapter, the primary source of data used to synthesize personas should be in-context interviews borrowing from ethnographic techniques, contextual inquiry, or other similar dialogues with and observation of actual and potential users. The quality of the data gathered following the process (outlined in Chapter 2) impacts the efficacy of personas in clarifying and directing design activities. Other data can support and supplement the creation of personas (listed in rough order of effectiveness):

- Interviews with users outside of their use contexts
- Information about users supplied by stakeholders and subject matter experts (SMEs)

- Market research data such as focus groups and surveys
- Market-segmentation models
- Data gathered from literature reviews and previous studies

However, none of this supplemental data can take the place of direct user interviews and observation. Almost every aspect of a well-developed persona can be traced back to sets of user statements or behaviors.

Personas represent types of users of a specific product

Although personas are depicted as specific individuals, because they function as arche-types, they *represent* a class or type of user of a *specific* interactive product. A persona encapsulates a distinct set of **behavior patterns** regarding the use of a particular product (or analogous activities if a product does not yet exist). You identify these behaviors by analyzing interview data. They are supported by supplemental data (qualitative or quantitative) as appropriate. These patterns, along with specific motivations or goals, define our personas. Personas are also occasionally called **composite user archetypes** because personas are, in a sense, composites assembled by grouping related usage patterns observed across individuals in similar roles during the Research phase.[3]

Personas used across multiple products

Organizations with more than one product often want to reuse the same personas. However, to be effective, personas must be context-specific: They should focus on the behaviors and goals related to the specific domain of a particular product. Personas, because they are constructed from specific observations of users interacting in specific contexts, cannot easily be reused across products, even when those products form a closely linked suite.[4]

For a set of personas to be an effective design tool for multiple products, the personas must be based on research concerning the usage contexts for all these products. In addition to broadening the scope of the research, an even larger challenge is to identify manageable and coherent sets of behavior patterns across all the contexts. You shouldn't assume that just because two users exhibit similar behaviors in regard to one product, those two users would behave similarly with respect to a different product.

As the focus expands to encompass more and more products, it becomes increasingly difficult to create a concise and coherent set of personas that represents the diversity of real-world users. We've found that, in most cases, personas should be researched and developed individually for different products.

Archetypes versus stereotypes

Don't confuse persona archetypes with **stereotypes**. Stereotypes are, in most respects, the antithesis of well-developed personas. Stereotypes usually are the result of designer or product team biases and assumptions, rather than factual data. Personas developed by drawing on inadequate research (or synthesized with insufficient empathy and sensitivity to interview subjects) run the risk of degrading to caricatures. Personas must be developed and treated with dignity and respect for the people they represent. If the designer doesn't respect his personas, nobody else will either.

Personas sometimes bring issues of social and political consciousness to the forefront.[5] Because personas provide a precise design target and also serve as a communication tool for the development team, the designer must choose particular demographic characteristics with care. Ideally, persona demographics should be a composite reflection of what researchers have observed in the interview population, modulated by broader market research. Personas should be *typical* and believable, but not stereotypical. If the data is inconclusive or the characteristic is unimportant to the design or its acceptance, we prefer to err on the side of gender, ethnic, age, and geographic diversity.

Personas explore ranges of behavior

The target market for a product describes demographics as well as lifestyles and sometimes job roles. What it does not describe are the ranges of different behaviors exhibited by members of that target market regarding the product and related situations. Ranges are distinct from *averages*: Personas do not seek to establish an average user; they express *exemplary* or definitive behaviors within these identified ranges.

Because products must accommodate *ranges* of user behavior, attitudes, and aptitudes, designers must identify a **persona set** associated with any given product. Multiple personas carve ranges of behavior into discrete clusters. Different personas represent different correlated behavior patterns. You arrive at these correlations by analyzing research data. This process of identifying behaviors is discussed in greater detail later in this chapter.

Personas have motivations

All humans have motivations that drive their behaviors; some are obvious, but many are subtle. It is critical that personas capture these motivations in the form of *goals*. The goals we enumerate for our personas (discussed at length later in this chapter) are shorthand for motivations that not only point to specific usage patterns but also provide a reason why those behaviors exist. Understanding *why* a user performs certain tasks

gives designers great power to improve or even eliminate tasks yet still accomplish the same goals. We can't overemphasize how important goals are for personas. In fact, we would go so far as to say that if your user model doesn't have any goals, what you have is not a persona.

Personas can represent relevant nonusers

The users and potential users of a product should always be an interaction designer's primary concern. However, sometimes it is useful to represent the needs and goals of people who do not use the product but nevertheless must be considered in the design process. For example, it is commonly the case with enterprise software (and children's toys) that the person who purchases the product is not the same person who uses it. In these cases, it may be useful to create one or more **customer personas**, distinct from the set of user personas. Of course, these should also be based on behavior patterns observed through ethnographic research, just as user personas are.

Similarly, for many medical products, patients do not interact directly with the user interface, but they have motivations and objectives that may be very different from those of the clinician using the product. Creating a **served persona** to represent patients' needs can be useful in these cases. We discuss served and customer personas in greater depth later in this chapter.

Nearly all networked software products have to take into account potential pranksters and malicious hackers. And sometimes for political reasons you need to embody a persona for whom the product specifically does *not* intend to serve. Each of these types of nonusers can be embodied in an **anti-persona** to have on hand in strategy, security, and design discussions.

Personas are more appropriate design tools than other user models

Many other user models are commonly employed in the design of interactive products, including user roles, user profiles, and market segments. These are similar to personas in that they seek to describe users and their relationship to a product. However, personas and the methods by which they are created and employed as a design tool differ significantly from these other user models in several key ways.

User roles

A user role or role model, as defined by Larry Constantine, is an *abstraction*—a defined relationship between a class of users and their problems, including needs, interests,

expectations, and patterns of behavior.[6] Agile development processes similarly reduce its users to their role.[7] As abstractions (generally taking the form of a list of attributes), they are not imagined as people and typically do not attempt to convey broader human motivations and contexts.

Holtzblatt and Beyer's use of roles in consolidated flow and cultural, physical, and sequence models is similar. It attempts to abstract various attributes and relationships from the people possessing them.[8]

We find these methods limiting for several reasons:

- It is more difficult to clearly communicate human behaviors and relationships in the abstract, isolated from the people who possess them. The human power of empathy cannot easily be brought to bear on abstract classes of people.
- Both methods focus myopically on *tasks* almost exclusively and neglect the use of goals as an organizing principle for design thinking and synthesis.
- Holtzblatt and Beyer's consolidated models, although useful and encyclopedic in scope, are difficult to bring together as a coherent tool for developing, communicating, and measuring design decisions.

Personas address each of these problems. Well-developed personas describe the same types of behaviors and relationships that user roles do, but they express them in terms of goals and examples in narrative. This makes it possible for designers and stakeholders to understand the implications of design decisions in human terms. Describing a persona's goals provides context and structure for tasks, incorporating how culture and work flow influence behavior.

In addition, focusing on user roles rather than on more complex behavior patterns can oversimplify important distinctions and similarities between users. It is possible to create a persona that represents the needs of several user roles. (For example, when a mobile phone is designed, a traveling salesperson might also represent the needs of a busy executive who's always on the road.) It is also possible that there are several people in the same role who think and act differently. (Perhaps a procurement planner in the chemical industry thinks about her job very differently from a procurement planner in the consumer electronics industry.) In consumer domains, roles are next to useless. If you're designing a website for a car company, "car buyer" is meaningless as a design tool. Different people approach the task in very different ways.

In general, personas provide a more holistic model of users and their contexts, whereas many other models seek to be more reductive. Personas can certainly be used in

combination with these other modeling techniques. As we'll discuss at the end of the chapter, some other models make extremely useful complements to personas.

Personas versus user profiles

Many usability practitioners use the terms **persona** and **user profile** synonymously. There is no problem with this if the profile is synthesized from first-hand ethnographic data and encapsulates the depth of information the authors have described. Unfortunately, all too often, the authors have seen user profiles that reflect Webster's definition of **profile** as a "brief biographical sketch." In other words, user profiles often consist of a name and picture attached to a short, mostly demographic description, along with a short paragraph of information unrelated to the design task at hand, for example, describing the kind of car this person drives, how many kids he has, where he lives, and what he does for a living. This kind of user profile is likely to be based on a stereotype. Although we give our personas names, and sometimes even cars and family members, these are employed sparingly. Supporting fictive detail plays only a minor part in persona creation. It is used just enough to make the persona come to life in the minds of the designers and product team.

Personas versus market segments

Marketing professionals may be familiar with a process similar to persona development because it shares some process similarities with market definition. The main difference between market segments and design personas is that the former are based on demographics, distribution channels, and purchasing behavior, whereas the latter are based on usage behavior and goals. The two are not the same and don't serve the same purpose. Marketing personas shed light on the sales process, whereas design personas shed light on the product definition and development process.

However, market segments play a role in persona development. They can help determine the demographic range within which to frame the persona hypothesis (see Chapter 2). Personas are segmented along ranges of usage behavior, not demographics or buying behavior, so there is seldom a one-to-one mapping of market segments to personas. Rather, market segments can act as an initial filter to limit the scope of interviews to people within target markets (see Figure 3-3). Also, we typically use the prioritization of personas as a way to make strategic product definition decisions (see the discussion of persona types later in this chapter). These decisions should incorporate market intelligence; understanding the relationship between user personas and market segments can be an important consideration here.
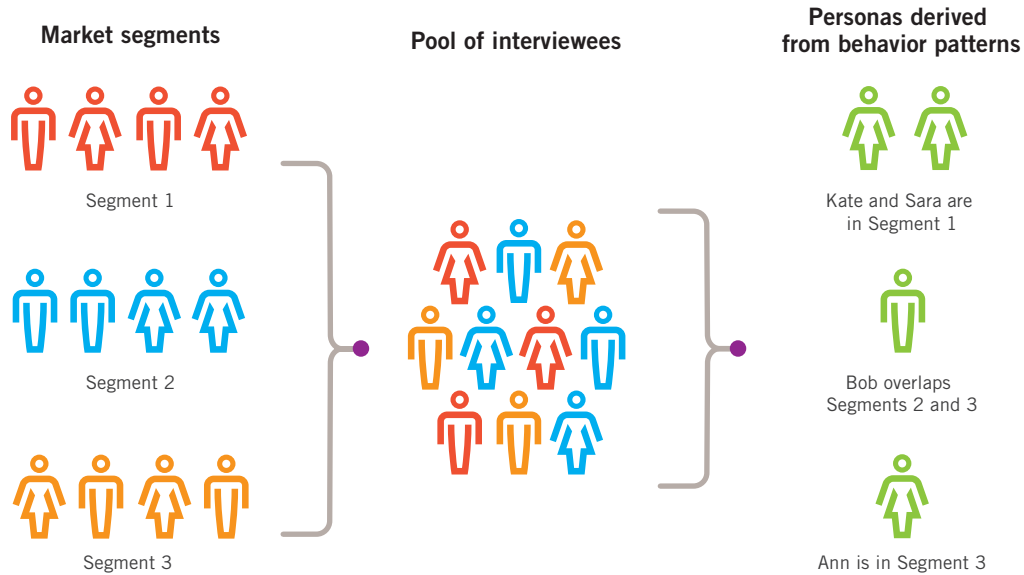
Figure 3-3: Personas versus market segments. Market segments can be used in the Research phase to limit the range of personas to target markets. However, there is seldom a one-to-one mapping between market segments and personas.

Understanding Goals

If personas provide the context for sets of observed behaviors, **goals** are the drivers behind those behaviors. User goals serve as a lens through which designers must consider a product's functions. The product's function and behavior must address goals via tasks—typically, as few tasks as necessary. Remember, tasks are only a means to an end; goals are that end.

Goals motivate usage patterns

People's or personas' goals motivate them to behave as they do. Thus, goals don't just provide an answer to why and how personas want to use a product. They also can serve as shorthand in the designer's mind for the sometimes complex behaviors in which a persona engages and, therefore, for their tasks as well.

Goals should be inferred from qualitative data

You usually can't ask a person what his goals are directly. Either he'll be unable to articulate them, or he'll be inaccurate or even imperfectly honest. People simply are unprepared to answer such self-reflective questions accurately. Therefore, designers and researchers need to carefully reconstruct goals from observed behaviors, answers to

other questions, nonverbal cues, and clues from the environment, such as the titles of books on shelves. One of the most critical tasks in the modeling of personas is identifying goals and expressing them succinctly: Each goal should be expressed as a simple sentence.

User goals and cognitive processing

Don Norman's book *Emotional Design* (Basic Books, 2005) introduced the idea that product design should address three different levels of cognitive and emotional processing: visceral, behavioral, and reflective. Norman's ideas, based on years of cognitive research, provide an articulated structure for modeling user responses to product and brand and a rational context for many intuitions long held by professional designers:

- **Visceral** is the most immediate level of processing. Here we react to a product's visual and other sensory aspects that we can perceive before significant interaction occurs. Visceral processing helps us make rapid decisions about what is good, bad, safe, or dangerous. This is one of the most exciting types of human behavior, and one of the most challenging to effectively support with digital products. Malcolm Gladwell explores this level of cognitive processing in his book *Blink* (Little, Brown and Company, 2005). For an even more in-depth study of intuitive decision making, see Gary Klein's *Sources of Power* (MIT Press, 1998) or *Hare Brain, Tortoise Mind* by Guy Claxton (Ecco, 1999).

- **Behavioral** is the middle level of processing. It lets us manage simple, everyday behaviors. According to Norman, these constitute the majority of human activity. Norman states—and rightly so—that historically, interaction design and usability practices have nearly exclusively addressed this level of cognitive processing. Behavioral processing can *enhance* or *inhibit* both lower-level visceral reactions and higher-level reflective responses. Conversely, both visceral and reflective processing can enhance or inhibit behavioral processing.

- **Reflective** is the least immediate level of processing. It involves conscious consideration and reflection on past experiences. Reflective processing can enhance or inhibit behavioral processing but has no direct access to visceral reactions. This level of cognitive processing is accessible only via memory, not through direct interaction or perception. The most interesting aspect of reflective processing as it relates to design is that, through reflection, we can integrate our experiences with designed artifacts into our broader life experiences and, over time, associate meaning and value with the artifacts themselves.

Designing for visceral response

Designing for the visceral level means designing what the senses initially perceive, before any deeper involvement with a product or artifact occurs. For most of us, that means

designing visual appearance and motion, although sound can also play a role—think of the distinctive Mac power-up chord. People designing devices may design for tactile sensations as well.

A misconception often arises when visceral-level design is discussed: that designing for visceral response is about designing *beautiful* things. Battlefield software and radiation-therapy systems are just two examples where designing for beauty may not be the proper focus. Visceral design is actually about designing for *affect*—that is, eliciting the appropriate psychological or emotional response for a particular context—rather than for aesthetics alone. Beauty—and the feelings of transcendence and pleasure it evokes—is really only a small part of the possible affective design palette. For example, an MP3 player and an online banking system require very different affects. We can learn a great deal about affect from architecture, the cinema and stage, and industrial design.

However, in the world of consumer products and services, attractive user interfaces *are* typically appropriate. Interestingly, usability researchers have demonstrated that users initially judge attractive interfaces to be more usable, and that this belief often persists long after a user has gained sufficient experience with an interface to have direct evidence to the contrary.[9] Perhaps the reason for this is that users, encouraged by perceived ease of use, make a greater effort to learn what may be a challenging interface and are then unwilling to consider their investment ill spent. For the scrupulous designer, this means that, when a user interface promises ease of use at the visceral level—or whatever else the visceral promise of an interaction may be—it should then be sure to deliver on that promise at the behavioral level.

Designing for behavior

Designing for the behavioral level means designing product behaviors that complement the user's own behaviors, implicit assumptions, and mental models. Of the three levels of design Norman contemplates, behavioral design is perhaps the most familiar to interaction designers and usability professionals.

One intriguing aspect of Norman's three-level model as it relates to design is his assertion that behavioral processing, uniquely among his three levels, has direct influence on and is influenced directly by both of the other two levels of processing. This would seem to imply that the day-to-day behavioral aspects of interaction design should be the primary focus of our design efforts, with visceral and reflective considerations playing a supporting role. Getting behavior design right—assuming that we also pay adequate attention to the other levels—provides our greatest opportunity to positively influence how users construct their experience with products.

Not following this line of reasoning can cause users' initial impressions to be out of sync with reality. Also, it is difficult to imagine designing for reflective meaning in memory

without a solid purpose and set of behaviors in place for the here and now. The user experience of a product or artifact, therefore, should ideally *harmonize elements of visceral design and reflective design with a focus on behavioral design*.

Designing for reflection

Reflective processing—and, particularly, what it means for design—is perhaps the most challenging aspect of the three levels of processing that Norman discusses. What is clear is that designing for the reflective level means designing to build long-term product relationships. What is unclear is the best way to ensure success—if that's even possible—at the reflective level. Does chance drive success here—being in the right place at the right time—or can premeditated design play a part in making it happen?

In describing reflective design, Norman uses several high-concept designs for commodity products as examples, such as impractically configured teapots and the striking Phillipe Starck juicer that graces the cover of his book. It is easy to see how such products—whose value and purpose are, in essence, the aesthetic statements they make—could appeal strongly to people's reflective desire for uniqueness or cultural sophistication that perhaps may come from an artistic or stylish self-image.

It is more difficult to see how products that also serve a truly useful purpose need to balance the stylistic and the elegant with the functional. The Apple iPhone comes very close to achieving this balance. Its direct manipulation touch interface merges almost seamlessly with its sleek industrial design. Its reflective potential is also significant, because of the powerful emotional connection people experience with their personal communications and their music (the iPhone is, of course, also an iPod). It's a winning combination that no single competitor has been able to challenge.

Few products become iconic in people's lives in the way that, say, the Sony Walkman or the iPhone has. Clearly some products stand little chance of ever becoming symbolic in people's lives—like Ethernet routers, for instance—no matter how wonderful they look or how well they behave. However, when the design of a product or service addresses users' goals and motivations—possibly going beyond the product's primary purpose, yet somehow connected to it via personal or cultural associations—the opportunity to create reflective meaning is greatly enhanced.

The three types of user goals

In *Emotional Design*, Norman presents his three-level theory of cognitive processing and discusses its potential importance to design. However, Norman does not suggest a method for systematically integrating his model of cognition and affect into the practice of design or user research. In our practice, we've found that the key to doing so lies

in properly delineating and modeling three specific types of user goals as part of each persona's definition.[10]

Three types of user goals correspond to Norman's visceral, behavioral, and reflective processing levels (see Figure 3-4):

- Experience goals
- End goals
- Life goals

Figure 3-4: The three types of user goals

Experience goals

Experience goals are simple, universal, and personal. Paradoxically, this makes them difficult for many people to talk about, especially in the context of impersonal business. Experience goals express how someone *wants to feel* while using a product, or the quality of his or her interaction with the product. These goals provide focus for a product's visual and aural characteristics, its interactive feel—such as animated transitions, latency, touch response, and a button's snap ratio (clickiness)— its physical design, and its micro-interactions. These goals also offer insights into persona motivations that express themselves at the visceral level:

- Feel smart and in control
- Have fun
- Feel reassured about security and sensitivity
- Feel cool or hip or relaxed
- Remain focused and alert

When products make users feel stupid or uncomfortable, it's unpleasant, and their effectiveness and enjoyment plummets, regardless of their other goals. Their level of resentment also increases. If they experience enough of this type of treatment, users will be primed to subvert the system. Any product that egregiously violates experience goals will ultimately fail, regardless of how well it purports to achieve other goals.

Interaction, visual, and industrial designers must translate persona experience goals into form, behavior, motion, and auditory elements that communicate the proper feel, affect, emotion, and tone. Visual language studies, as well as mood or inspiration boards, which attempt to establish visual themes based on persona attitudes and behaviors, are a useful tool for defining personas' tonal expectations.

End goals

End goals represent the user's motivation for performing the *tasks* associated with using a specific product. When you pick up a cell phone or open a document with a word processor, you likely have an outcome in mind. A product or service can help accomplish such goals directly or indirectly. These goals are the focus of a product's interaction design and information architecture and the functional aspects of industrial design. Because behavioral processing influences both visceral and reflective responses, end goals should be among the most significant factors in determining the overall product experience. End goals must be met for users to think that a product is worth their time and money.

Here are some examples of end goals:

- Be aware of problems before they become critical.
- Stay connected with friends and family.
- Clear my to-do list by 5:00 p.m. every day.
- Find music that I'll love.
- Get the best deal.

Interaction designers must use end goals as the foundation for a product's behaviors, tasks, look, and feel. Context or day-in-the-life scenarios and cognitive walkthroughs are effective tools for exploring users' goals and mental models, which, in turn, facilitate appropriate behavioral design.

Life goals

Life goals represent the user's personal aspirations that typically go beyond the context of the product being designed. These goals represent deep drives and motivations that

help explain *why* the user is trying to accomplish the end goals he seeks to accomplish. Life goals describe a persona's long-term desires, motivations, and self-image attributes, which cause the persona to connect with a product. These goals are the focus of a product's overall design, strategy, and branding:

- Live the good life.
- Succeed in my ambitions to…
- Be a connoisseur of…
- Be attractive, popular, and respected by my peers.

Interaction designers must translate life goals into high-level system capabilities, formal design concepts, and brand strategy. Mood boards and context scenarios can be helpful in exploring different aspects of product concepts, and broad ethnographic research and cultural modeling are critical for discovering users' behavior patterns and deeper motivations. Life goals rarely figure directly into the design of an interface's specific elements or behaviors. However, they are very much worth keeping in mind. A product that the user discovers will take him closer to his life goals, not just his end goals, will win him over more decisively than any marketing campaign. Addressing users' life goals makes the difference (assuming that other goals are also met) between a satisfied user and a fanatically loyal user.

User goals are user motivations

In summary, it's important to remember that understanding personas is more about understanding motivations and goals than it is about understanding specific tasks or demographics. Linking persona goals with Norman's model, top-level user motivations include the following:

- Experience goals, which are related to visceral processing: how the user wants to *feel*
- End goals, which are related to behavior: what the user wants to *do*
- Life goals, which are related to reflection: who the user wants to *be*

Using personas, goals, and scenarios (as you'll learn in upcoming chapters) provides the key to unlocking the power of visceral, behavioral, and reflective design and bringing these together into a harmonious whole. While some of our best designers seem to understand and act on these aspects of design almost intuitively, consciously designing for all levels of human cognition and emotion offers tremendous potential for creating more satisfying and affective user experiences.

Nonuser goals

User goals are not the only type of goals that designers need to take into account. Customer goals, business goals, and technical goals are all nonuser goals. Typically, these goals must be acknowledged and considered, but they do not form the basis of the design direction. Although these goals do need to be addressed, they must not be addressed at the user's expense.

Customer goals

Customers, as already discussed, have different goals than users. The exact nature of these goals varies quite a bit between consumer and enterprise products. Consumer customers often are parents, relatives, or friends who often have concerns about the safety and happiness of the people for whom they are purchasing the product. Enterprise customers typically are IT managers or procurement specialists, and they often have concerns about security, ease of maintenance, ease of customization, and price. Customer personas also may have their own life, experience, and especially end goals in relation to the product if they use it in any capacity. Customer goals should never trump end goals but need to be considered within the overall design.

Business and organizational goals

Businesses and other organizations have their own requirements for products, services, and systems, which you also should model and consider when devising design solutions. The goals of businesses, where users and customers work, are captured in user and customer personas, as well as organizational "personas" (discussed later in this chapter). It is important to identify the business goals of the organization commissioning the design and developing and selling (or otherwise distributing) the product early in the design process. Clearly, these organizations are hoping to accomplish something with the product (which is why they are willing to spend money and effort on design and development).

Business goals include the following:

- Increase profit.
- Increase market share.
- Retain customers.
- Defeat the competition.
- Use resources more efficiently.
- Offer more products or services.
- Keep its IP secure.

You may find yourself designing on behalf of an organization that is not necessarily a business, such as a museum, nonprofit, or school (although many such organizations are run as businesses these days). These organizations also have goals that must be considered, such as the following:

- Educate the public.
- Raise enough money to cover overhead.

Technical goals

Most of the software-based products we use every day are created with technical goals in mind. Many of these goals ease the task of software creation, maintenance, scalability, and extensibility, which are developers' goals. Unfortunately meeting these goals often comes at the expense of user goals. Technical goals include the following:

- Run in a variety of browsers.
- Safeguard data integrity.
- Increase application execution efficiency.
- Use a particular development language or library.
- Maintain consistency across platforms.

Technical goals in particular are essential to the development staff. It is important to stress early in the education process that these goals must ultimately serve user and business goals. Technical goals are not terribly meaningful to a product's success unless they are derived from the need to meet other, more human-oriented goals. It might be a software company's *task* to use new technology, but it is rarely the *user's goal* for them to do so. In most cases, users don't care if their job is accomplished with hierarchical databases, relational databases, object-oriented databases, flat-file systems, or black magic. What we care about is getting our job done swiftly, effectively, and with a modicum of ease and dignity.

Successful products meet user goals first

"Good design" has meaning only for someone who uses a product for a particular purpose. You cannot have purposes without people. The two are inseparable. This is why personas are such an important tool in the process of designing behavior: They represent specific people with specific purposes or goals.

The most important purposes or goals to consider when designing a product are the goals of the individuals who actually use the product, not necessarily the goals of its purchasers or its developers. A real person, not a corporation or even an IT manager,

interacts with your product. Therefore, you must regard her personal goals as more significant than those of the corporation that employs her, the IT manager who supports her, or the developer who builds for her. Your users will do their best to achieve their employer's business goals, while at the same time looking after their own personal goals. The user's most important goal is always to retain her human dignity and not feel stupid.

We can reliably say that we make the user feel stupid if we let her make big mistakes, keep her from getting an adequate amount of work done, or bore her.

DESIGN PRINCIPLE   *Don't make the user feel stupid.*

This is probably the most important interaction design guideline. This book examines numerous ways in which existing software makes the user feel stupid, and we explore ways to avoid that trap.

The essence of good interaction design is devising interactions that achieve the goals of the manufacturer or service provider and its partners while supporting user goals.

Constructing Personas

As previously discussed, personas are derived from qualitative research—especially the behavioral patterns observed during interviews with and observations of a product's users and potential users (and sometimes customers). Gaps in this data are filled by supplemental research and data provided by SMEs, stakeholders, quantitative research, and other available literature. Our goal in constructing a set of personas is to represent the diversity of observed motivations, behaviors, attitudes, aptitudes, constraints, mental models, work or activity flows, environments, and frustrations with current products or systems.

Creating believable and useful personas requires an equal measure of detailed analysis and creative synthesis. A standardized process aids both of these activities significantly. The process described in this section is the result of an evolution in practice over the span of hundreds of interaction design projects, developed by industry veterans Robert Reimann, Kim Goodwin, and Lane Halley during their tenure at Cooper.

There are a number of effective methods for identifying behavior patterns in research and turning these into useful user archetypes. We've found the transparency and rigor of this process to be an ideal way for designers new to personas to learn how to properly

construct personas. It also helps experienced designers to stay focused on actual behavior patterns, especially in consumer domains. Figure 3-5 shows the principle steps:

1. Group interview subjects by role.
2. Identify behavioral variables.
3. Map interview subjects to behavioral variables.
4. Identify significant behavior patterns.
5. Synthesize characteristics and define goals.
6. Check for completeness and redundancy.
7. Designate persona types.
8. Expand the description of attributes and behaviors.
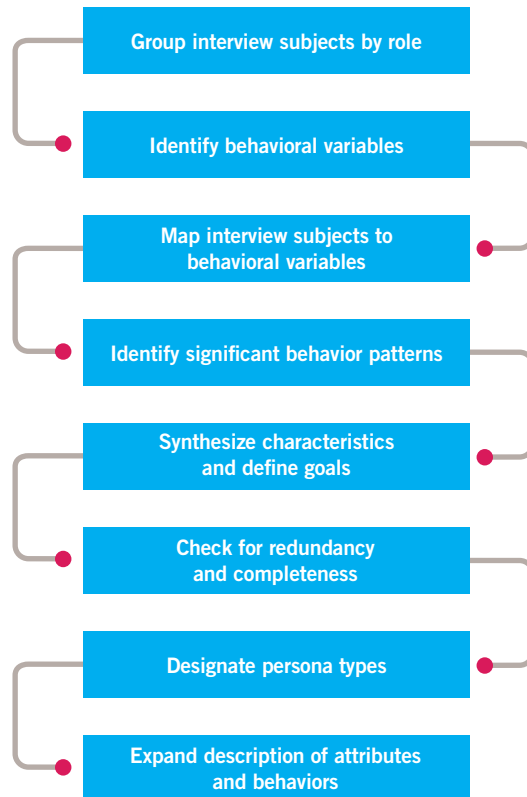
Figure 3-5: Overview of the persona creation process

Step 1: Group interview subjects by role

After you have completed your research and performed a cursory organization of the data, group your interviewees according to their *roles*. For enterprise applications, roles

are easy to delineate, because they usually map to job roles or descriptions. Consumer products have more subtle role divisions, including family roles, attitudes or approaches to relevant activities, or interests and aptitudes regarding lifestyle choices.

Step 2: Identify behavioral variables

Once you've grouped your interviewees by role, list the distinct aspects of observed behavior for each role as a set of **behavioral variables**. Demographic variables such as age or geographic location may sometimes seem to affect behavior. But be wary of focusing on demographics, because behavioral variables will be far more useful in developing effective user archetypes.

Generally, we see the most important distinction between behavior patterns emerge by focusing on the following types of variables:

- **Activities**—What the user does; frequency and volume
- **Attitudes**—How the user thinks about the product domain and technology
- **Aptitudes**—What education and training the user has; ability to learn
- **Motivations**—Why the user is engaged in the product domain
- **Skills**—User abilities related to the product domain and technology

Although the number of variables will differ from project to project, it is typical to find 15 to 30 variables per role.

These variables may be very similar to those you identified as part of your persona hypothesis. Compare behaviors identified in the data to the assumptions made in the persona hypothesis. Were the possible roles that you identified truly distinct? Were the behavioral variables (see Chapter 2) you identified valid? Were there additional, unanticipated ones, or ones you anticipated that weren't supported by data?

List the complete set of behavioral variables observed. If your data is at variance with your assumptions, you need to add, subtract, or modify the roles and behaviors you anticipated. If the variance is significant enough, you may consider additional interviews to cover any gaps in the new behavioral ranges you've discovered.

Step 3: Map interview subjects to behavioral variables

After you are satisfied that you have identified the set of significant behavioral variables exhibited by your interview subjects, the next step is to map each interviewee against each variable. Some of these variables will represent a continuous range of behavior,

such as confidence in using technology. Others will represent multiple discrete choices, such as using a digital camera versus using a film camera.

Mapping the interviewee to a precise point in the range isn't as critical as identifying the placement of interviewees in relationship to each other. In other words, it doesn't matter if an interviewee falls at precisely 45 or 50 percent on the scale. There's often no good way to measure this precisely; you must rely on your gut feeling based on your observations of the subject. The desired outcome of this step is to accurately represent how multiple subjects cluster with respect to each significant variable, as shown in Figure 3-6.
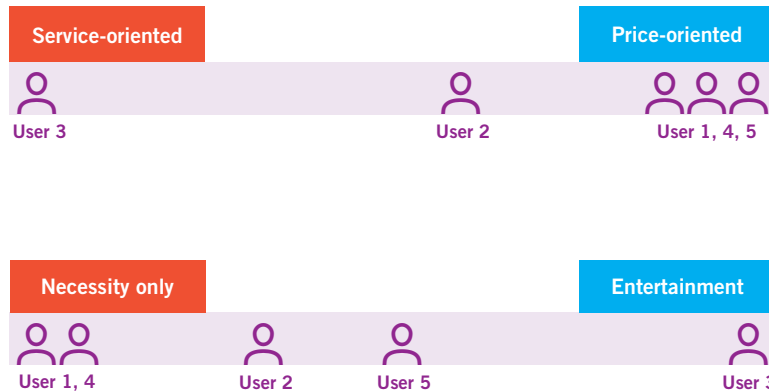
Figure 3-6: Mapping interview subjects to behavioral variables. This example is from an online store. Interview subjects are mapped across each behavioral axis. Precision of the absolute position of an individual subject on an axis is less important than its relative position to other subjects. Clusters of subjects across multiple axes indicate significant behavior patterns.

Step 4: Identify significant behavior patterns

After you have mapped your interview subjects, look for clusters of subjects that occur across multiple ranges or variables. A set of subjects who cluster in six to eight different variables will likely represent a significant **behavior pattern** that will form the basis of a persona. Some specialized roles may exhibit only one significant pattern, but typically you will find two or even three such patterns.

For a pattern to be valid, there must be a logical or causative connection between the clustered behaviors, not just a spurious correlation. For example, there is clearly a logical connection if data shows that people who regularly purchase CDs also like to download MP3 files. But there is probably no logical connection if the data shows that interviewees who frequently purchase CDs online are also vegetarians.

Step 5: Synthesize characteristics and define goals

We derive a persona's goals and other attributes from their behaviors. These behaviors are synthesized from what was observed/identified in the research process as representing meaningful, typical use of the product over a period of time that adequately captures the relevant set of user actions. We call this a "day in the life," but the period of time really depends on the product or service being used and exactly what the persona does with it. Executive personas, for example, often have particular behaviors associated with quarterly and yearly earnings reports. Consumers often have behaviors particular to their culture's holidays that should be investigated.

For each significant behavior pattern you identify, you must synthesize details from your data. These details should include the following at a minimum:

- The behaviors themselves (activities *and* the motivations behind them)
- The use environment(s)
- Frustrations and pain points related to the behavior using current solutions
- Demographics associated with the behavior
- Skills, experience, or abilities relating to the behavior
- Attitudes and emotions associated with the behavior
- Relevant interactions with other people, products, or services
- Alternate or competing ways of doing the same thing, especially analog techniques

At this point, brief bullet points describing the behavior's characteristics are sufficient. Stick to observed behaviors as much as possible. A description or two that sharpens the personalities of your personas can help bring them to life. However, too much fictional biography, especially if it includes quirky details, is a distraction and makes your personas distracting. Remember that you are creating a design tool, not a character sketch for a novel. Only concrete data can support the design and business decisions your team will ultimately make.

One fictional detail at this stage *is* important: the personas' first and last name. The name should evoke the type of person the persona is without tending toward distracting uniqueness, caricature, or stereotype. We use a number of techniques to help create persona names, but a standard go-to is the app and site `http://random-name-generator.info/`. You can also, at this time, add some demographic information such as age, geographic location, relative income (if appropriate), and job title. This information primarily helps you visualize the persona better as you assemble the behavioral details. From this point on, you should refer to the persona by his or her name.

Defining goals

Goals are the most critical detail to synthesize from your interviews and observations of behaviors. Goals are best derived by analyzing the behavior patterns comprising each persona. By identifying the logical connections between each cluster of interviewees' behaviors, you can begin to infer the goals that lead to those behaviors. You can infer goals both by observing actions (what interview subjects in each persona cluster are trying to accomplish and why) and by analyzing subject responses to goal-oriented interview questions (see Chapter 2).

To be effective as design tools, goals must always relate directly, in some way, to the product being designed. Typically, the majority of useful goals for a persona are *end goals*. You can expect most personas to have three to five end goals associated with them. Life goals are most useful for personas of consumer-oriented products, but they can also make sense for enterprise personas in transient job roles. Zero or one life goal is appropriate for most personas. General experience goals such as "Don't feel stupid" and "Don't waste time" can be taken as implicit for almost any persona. Occasionally, a specific domain may dictate the need for more specific experience goals; zero to two experience goals is appropriate for most personas.

Personas and social relationships

It sometimes makes sense for a product's set of personas to be part of the same family or team within an organization and to have interpersonal or social relationships with each other.

When considering whether it makes sense for personas to have business or social relationships, think about these two points:

- Whether you observed any behavioral variations in your interview subjects related to variations in company size, industry, or family/social dynamic (In this case, you'll want to make sure that your persona set represents this diversity by being situated in at least a couple of different businesses or social settings.)
- If it is critical to illustrate workflow or social interactions between coworkers or members of a family or social group

If you create personas that work for the same company or that have social relationships with each other, you might run into difficulties if you need to express a significant goal that doesn't belong with the pre-established relationship. A single social relationship between your set of personas is easier to define than several different, unrelated social relationships between individual personas and minor players outside the persona set. However, it can be much better to put the initial effort into developing

diverse personas than to risk the temptation of bending more diverse scenarios to fit a single social dynamic.

Step 6: Check for completeness and redundancy

At this point, your personas should be starting to come to life. You should check your mappings and personas' characteristics and goals to see if any important gaps need filling. This again may point to the need to perform additional research to find particular behaviors that are missing from your behavioral axes. You might also want to check your notes to see if you need to add any political personas to satisfy stakeholder assumptions or requests. We've occasionally had to add personas with identical goals but different locales to satisfy branches of the client organization in those locales, that their constituents' voices were heard and are represented in the design.

If you find that two personas seem to vary only by demographics, you may choose to eliminate one of the redundant personas or tweak your personas' characteristics to make them more distinct. Each persona should vary from all the others in at least one significant behavior. If you've done a good job of mapping, this shouldn't be an issue.

By making sure that your persona set is complete and that each persona is meaningfully distinct, you ensure that your personas sufficiently represent the diversity of behaviors and needs in the real world. You also ensure that you have as compact a design target as possible, which reduces work when you begin designing interactions.

Step 7: Designate persona types

By now, your personas should feel very much like a set of real people you know. The next part of persona construction is a key step in the process of turning your qualitative research into a powerful set of design tools.

Design requires a target—the audience upon whom the design is focused. The more specific the target, the better. Trying to create a design solution that simultaneously serves the needs of even three or four personas can be an overwhelming task.

What we must do then is *prioritize* our personas to determine which should be the primary design target. The goal is to find a single persona from the set whose needs and goals can be completely and happily satisfied by a single interface without disenfranchising any of the other personas. We accomplish this through a process of designating *persona types*. There are six types of personas, and they are typically designated in roughly the order listed here:

- Primary
- Secondary
- Supplemental
- Customer
- Served
- Negative

We discuss each of these persona types and their significance from a design perspective in the following sections.

Primary personas

Primary personas are the main target of interface design. A product can have only one primary persona per *interface*, but it is possible for some products (especially enterprise products) to have multiple distinct interfaces, each targeted at a distinct primary persona. For example, a healthcare information system might have separate clinical and financial interfaces, each targeted at a different persona. It should be noted that we use the term *interface* in an abstract sense here. In some cases, two separate interfaces might be two separate applications that act on the same data. In other cases, the two interfaces might simply be two different sets of functionality served to two different users based on their role or customization.

A primary persona will not be satisfied by a design targeted at any other persona in the set. However, if the primary persona is the target, all other personas will not, at least, be dissatisfied. (As you'll see, we will then figure out how to satisfy these other personas without disturbing the primary.)

DESIGN PRINCIPLE	*Focus the design for each interface on a single primary persona.*

Choosing the primary persona is a process of elimination: You must test each persona by comparing its goals against goals of the others. If no clear primary persona is evident, it could mean one of two things: Either the product needs multiple interfaces, each with a suitable primary persona (which is often the case for enterprise and technical products), or the product is trying to accomplish too much. If a consumer product has multiple primary personas, its scope may be too broad.

Avoid the trap of simply selecting the persona who maps to the largest market segment. The OXO Good Grips line of products was originally designed to be easy to use for people

with arthritis. It turns out that satisfying this user with the most constraints (and a tiny portion of the total market) satisfies the bulk of customers greatly. The largest segment may not be your primary or most leveraged persona.

Secondary personas

A *secondary persona* is mostly satisfied with the primary persona's interface. However, it has specific additional needs that can be accommodated without upsetting the product's ability to serve the primary persona. We do not always have a secondary persona. More than three or four secondary personas can be a sign that the proposed product's scope may be too large and unfocused. As you work through solutions, your approach should be to first design for the primary, and then to adjust the design to accommodate the secondary.

Supplemental personas

User personas that are not primary or secondary are *supplemental personas*. Their needs are completely represented by a combination of primary and secondary personas and are completely satisfied by the solution we devise for one of our primaries. Any number of supplemental personas can be associated with an interface. Often political personas— the ones added to the cast to address stakeholder assumptions—become supplemental personas.

Customer personas

Customer personas address the needs of customers, not end users, as discussed earlier in this chapter. Typically, customer personas are treated like secondary personas. However, in some enterprise environments, some customer personas may be primary personas for their own administrative interface.

Served personas

Served personas are somewhat different from the persona types already discussed. They are not users of the product, but they are *directly affected by the use of the product*. A patient being treated by a radiation therapy machine is not a user of the machine's interface, but she is very much *served* by a good interface. Served personas provide a way to track second-order social and physical ramifications of products. These are treated like secondary personas.

Negative personas

Negative personas (also sometimes called *anti-personas*) are used to communicate to stakeholders and product team members that the product is *not* being built to serve specific types of users. Like served personas, they aren't users of the product. Their use is purely rhetorical: to help communicate to other members of the team that a persona should definitely *not* be the product's design target. Good candidates for negative personas are often technology-savvy early-adopter personas for consumer products, criminals, less-harmful pranksters and "trolls," and IT specialists for business-user enterprise products.

Step 8: Expand the description of attributes and behaviors

Your list of bullet-point characteristics and goals arrived at in Step 5 points to the essence of complex behaviors but leaves much implied. Third-person narrative is far more powerful at conveying the persona's attitudes, needs, and problems to other team members. It also deepens the designer/authors' connection to and empathy for the personas and their motivations.

The persona narrative

A typical persona description should be a synthesis of the most important details observed during research, relevant to this persona. This becomes an effective communication tool. Ideally, the majority of your user research findings should be contained in your persona description. This will be the manner in which your research directly informs design activities (as you will see in the upcoming chapters).

This narrative should be no longer than one or two pages of prose (or Powerpoint slides). (A paragraph for every one or two bullet points from the characteristics in Step 5 is appropriate.) The persona narrative does not need to contain every observed detail. Ideally, the designers also performed the research, and most people outside the design team do not require more detail than this.

The narrative must, by nature, contain some fictional situations, but, as previously discussed, it is not a short story. The best narrative quickly introduces the persona in terms of his job or lifestyle. It briefly sketches a day in his life, including peeves, concerns, and interests that have direct bearing on the product. Details should be an expansion of your list of characteristics, with additional data derived from your observations and interviews. The narrative should express what the persona is looking for in the product by way of a conclusion.

Be careful about the amount of detail in your descriptions. The detail should not exceed the depth of your research. In scientific disciplines, if you record a measurement of 35.421 meters, this implies that your measurements are accurate to .001 meters. A detailed persona description implies a similar level of observation in your research.

Also be sure not to introduce hints about design solutions into the persona description. This narrative is about describing the persona's behaviors and pain points, *not* about how you plan to address them. That is the next step in the design process and is covered in Chapter 4.

In summary:

- *Do* include summarizing descriptions of all significant behavior patterns in your narrative.
- *Do not* include excessive fictional descriptions. Include just enough detail to cover basic demographics and to weave the behavior patterns into a story.
- *Do not* add levels of detail to your behavioral descriptions that you did not observe.
- *Do not* introduce solutions into your persona narrative. Rather, highlight pain points.

Finally, never list ranges or averages as details for personas. Personas are individuals, and would never have 1.5 kids making $35,000-45,000 per year. Those are values for market segments. If these details are important to your persona, pick specifics.

The persona photo

When you start developing your narrative, choose photographs for your personas. Photographs make them feel more real as you create the narrative and engage others on the team when you are finished. You should take great care in choosing a photograph. The best photos capture demographic information, hint at the environment (a persona for a nurse should be wearing a nurse's uniform and be in a clinical setting, perhaps with a patient), and capture the persona's general attitude (a photo for a clerk overwhelmed by paperwork might look harried). The authors keep handy several searchable databanks of stock photography as well as sources of Creative-Commons licensed repositories to help us find the right persona pictures. Here are some other things to watch for with persona photos:

- *Do not* use photos with unusual camera angles or distortions. This is distracting and makes the persona look like a caricature.
- *Do not* use photos with exaggerated expressions. These also will look like a caricature.
- *Do not* use photos in which people are obviously posed and smiling for the camera.
- *Do* use photos where the subjects look like average people, rather than models.

- *Do* use photos in which the subject is engaged in an appropriate activity against a realistic background.
- *Try* to keep photos across persona sets similar in style and crop.

We have also found it sometimes useful to create photographic collages for each persona to convey more emotional and experiential forces that drive the persona (see Figure 3-7). Numerous small images juxtaposed have the potential to convey things that are difficult to describe in words. There are also times that we find it useful to create models of the personas' environments (for example, in the form of a floor plan). Again, this helps make these environmental considerations more tangible.

Figure 3-7: Collages such as this, combined with carefully written narratives, are an effective way to convey the emotional and experiential aspects of a persona.

When creating such communication aides, it's important to remember that personas are design and decision-making tools, not an end in themselves. While power can come from creating a holistic image of a persona, too much embellishment and theater can potentially make personas seem like a fluffy waste of time. This can ultimately reduce their usefulness as user models.

Personas in Practice

Over the last decade, since the persona creation process was described in the second edition of this book, questions have arisen about the proper use of personas. This section endeavors to answer some of the criticisms leveled at persona-based design methods. It also discusses additional persona-related concepts we've used in our practice.

Misconceptions about personas

The Inmates Are Running the Asylum first introduced personas as an approach to generating Goal-Directed design concepts back in 1998. Since that time, personas have remained a topic of controversy among some designers and user researchers. Unfortunately, many of the sometimes strenuous objections to the persona method are the result of misunderstandings about how personas should be constructed, confusion about what they are best used to accomplish, or a reaction to the misapplication of persona methods. We'll try to set the record straight by clarifying some of these misapprehensions.

Designers "make up" personas

Perhaps the biggest criticism we have heard about personas is that they are just made up. Nothing could be further from the truth when personas are constructed correctly. The behavior patterns that are captured using personas are real and ideally come from actual ethnographic data—from user interviews and firsthand observations. Persona goals are constructed from inferences and deductions made by the designer when interpreting this data.

This misconception most likely comes from the fact that fictional names, superficial (but true to actual gathered data) demographic information, and narrative storytelling techniques are overlaid onto the behavioral data. This is done to better engage the empathy of designers and to communicate user needs to product team members. These narrative constructs are communication aids only. They do not affect the real behavioral data used to characterize the personas, and upon which design decisions will ultimately be made.

Unfortunately, not everyone claiming to use personas follows the detailed data gathering process outlined in Chapter 2, or the persona creation process described in this chapter. Sometimes demographic user profiles may be presented as personas with little more than a bit of narrative window dressing. If you are involved with a product team, design team, or client who claims to be using personas, ask them how their personas were constructed, what user data was gathered, and how it was analyzed to create them. An immediate yellow flag is a "persona" that does not have any goals associated with it. While demographic information alone may help teams communicate a little about the

makeup of their user base, that information is insufficient to build personas that will be of use in generating detailed design concepts.

Personas aren't as useful as involving real people

Personas are purposely constructed from aggregated data gathered from real (and potential) users. Over the years, some practitioners have questioned whether bringing actual users with real photos, real demographics, and real, particular behaviors into the design process wouldn't be more effective and "true."

This approach, known as *participatory design*, seems to solve some problems from a philosophical and political standpoint, because you can't argue about what a real user would do when you have one there to consult. In reality, however, it actually creates some significant problems for the conceptualization of the design. Clustering and analyzing the behaviors of many people serves an essential purpose: It allows the designers to separate critical behaviors and needs that are common across a broad set of users from the *idiosyncratic* behaviors that are particular to a given user. Focusing on individual users rather than aggregated sets of user behaviors also makes it more likely that you may *miss* key behaviors that your individual user (with his idiosyncratic behaviors) just doesn't happen to do—or does differently than most users.

If your client or product team insists on involving real users, you can first explain to them that personas are created from observations of real users. Then offer to provide them with audio or video transcripts of your interview sessions (make sure that at least some of your interviewees are open to this). Or you can invite a stakeholder along on an interview. Once your team has some proof that your personas are based on actual user behavior patterns, you may find that such objections settle themselves.

If that doesn't work, you'll need to convince them that any individual user feedback has to be synthesized against broader patterns in research, or aggregated with other users providing feedback in the same session.

People don't do tasks

Especially with consumer products and social systems, it is fair to say that people rarely think in terms of tasks. Few people hop on Facebook, turn on a television, or visit a news site to accomplish a particular thing that they have in mind. It can be more to "just see what's happening," and respond. Some designers have subsequently advocated that a task-based approach will not serve these domains, and argue to get rid of personas and design just from inspiration. While it might be correct to say that not all users think in terms of tasks, don't throw the baby out with the bathwater. Tasks aren't the beating

heart of personas. Goals are, and "Be caught up on what's happening" is a perfectly reasonable goal.

Personas are traceable

While every major persona trait should be derived from research, some designers try to only include persona traits if that exact trait was seen in a user interview. This is useful for organizations that want to ensure that a strong user-centered methodology was followed and that designers "aren't just making it up." But few interviewees state their goals succinctly out loud. Many times, the best quote is one that embodies what many interviewees said, but that no one particular interviewee said. If you receive pressure to ensure traceability, counter that personas are traceable to the patterns seen across research, not from specific, idiosyncratic interviews.

Quantifying personas

Some design practitioners believe quantitative data is needed to *validate* personas. In fact, if you follow the process described in Chapter 2 and this chapter closely, you should have all the validation you need, in the form of detailed *qualitative* information.

The typical response to this from stakeholders or teams very wed to quantitative data is "How do you know that these personas truly represent the majority of users?" This question comes from a confusion of personas with *market segments*. Market segmentation divides potential customers into groups based on demographic and psychographic differences. Personas, on the other hand, represent behaviors using a product and, in terms of the interface, do not always represent exclusive groupings. A given interface design may support the needs of one or more secondary personas (as well as supplemental personas) in addition to the primary persona that determines the structure of that interface. So, while the primary persona may not represent a majority of the market alone, the combination of primary, secondary, and supplemental personas served by an interface typically does.

That said, it can be useful to understand the market sizing of your personas. This is true especially when it's time for the development team to prioritize features at the detail level (taking into account the information about non-exclusive persona groupings). As mentioned in the preceding chapter, it is possible to construct "persona personality" surveys that identify which persona each participant has the closest affinity to. The process for doing so is as follows:

1 Revisit your behavioral variables and interviewee mappings to them.

2 For each variable, construct a multiple-choice question, the answer to which will distinguish between the different personas. (Note that sometimes multiple personas will have similar behaviors for a given variable.)

3 Construct two to four more questions for each variable that ask the same question in a different way. This helps ensure that participants are answering accurately.

4 Arrange the survey questions in random order.

5 Field the survey to participants. Sample size is important: You can use an online calculator such as `http://www.surveysystem.com/sscalc.htm` to find the appropriate sample size for your product.

6 Tabulate each participant's responses, tracking how many answers match each persona. The persona with the most responses for a given participant is the persona that participant has an affinity to.

7 Tabulate how many participants have an affinity to each persona, and divide this number by the number of total participants. This is the market size (percentage) for your personas.

Remember, it's okay if your primary persona isn't the largest segment; you also need to figure the effect of secondary and supplemental personas, all of whom will be using a single design.

Organizational "personas"

Personas are a tool for characterizing people's behavior patterns. However, we have found that a similar but much simpler concept is also useful for describing the behaviors of *organizations* that our personas are employed by or affiliated with. For example, if you are designing a payroll system, the needs of a small business and how the personas in it interact are different from those of a multinational corporation. The personas are likely different (perhaps with more specialized roles than in a small business), and so is how they interact, and so, in addition, are the rules and behaviors of the business itself. You can imagine that this would also be true for other types of organizations for which you may need to design—perhaps even for social units such as families at different stages of life.

As you gathered information for your personas, you undoubtedly captured information about the organizations they worked for or were otherwise associated with. It is often helpful to develop aggregate, fictional organizational "personas" with which to affiliate your personas, using similar narrative approaches. Usually an evocative organization name and one or two paragraphs describing the organization's behaviors and pain points regarding the product or service being designed are enough to provide the necessary context. In place of a photo, our designers have created logos for these companies for use in presentation materials.

When resources are limited: provisional personas

Although it is highly desirable that personas be based on detailed qualitative data, sometimes there is simply not enough time, resources, budget, or corporate buy-in to perform the necessary fieldwork. In these cases, *provisional* personas (or, as Don Norman calls them, "ad hoc" personas) can be useful rhetorical tools to clearly communicate assumptions about who the important users are and what they need. These personas also can enforce rigorous thinking about serving specific user needs (even if these needs are not validated).

Provisional personas are structured much like real personas but rely on available data and designer best guesses about behaviors, motivations, and goals. They are typically based on a combination of stakeholder and subject matter expert knowledge of users (when available), as well as what is understood about users from existing market data. Provisional personas are, in fact, a more fleshed-out persona hypothesis (as described in Chapter 2).

Our experience is that, regardless of a lack of research, using provisional personas yields better results than no user models. Like real personas, provisional personas can help focus the product team and build consensus around product features and behaviors. There are caveats, however. Provisional personas are called this because they should be recognized as stand-ins for personas based on definitive qualitative data. While provisional personas may help focus your design and product team, if you do not have data to back up your assumptions, you may do the following:

- Focus on the wrong design target.
- Focus on the right target but miss key behaviors that could differentiate your product.
- Have a difficult time getting buy-in from individuals and groups who did not participate in their creation.
- Discredit the value of personas, causing your organization to reject the use of personas in the long term.

If you are using provisional personas, it's important to do the following:

- Clearly label and explain them as such. We often give them only first names.
- Represent them visually with sketches, not photos, to reinforce their provisional nature.
- Try to use as much existing data as possible (market surveys, domain research, subject matter experts, field studies, or personas for similar products).
- Document what data was used and what assumptions were made.
- Steer clear of stereotypes (which is more difficult to do without field data).
- Focus on behaviors and goals, not demographics.

Other Design Models

Personas are extremely useful tools, but they certainly are not the only tool to help model users and their environment. Holtzblatt and Beyer's *Contextual Design* (Morgan Kaufmann, 1993) provides a wealth of information on the models briefly discussed here.

Work flow models

Work flow or **sequence models** are useful for capturing information flow and decision-making processes inside organizations. They usually are expressed as flow charts or directed graphs that capture several phenomena:

- A process's goal or desired outcome
- The frequency and importance of the process and each action
- What initiates or prompts the execution of the process and each action
- Dependencies—what must be in place to perform the process and each action, as well as what is dependent on the completion of the process and each action
- People who are involved, and their roles and responsibilities
- Specific actions that are performed
- Decisions that are made
- Information that is used to support decisions
- What can go wrong—errors and exception cases
- How errors and exceptions are corrected

A well-developed persona should capture individual work flows, but work flow models are still necessary for capturing exhaustive, interpersonal, or organizational work flows. Interaction design based primarily on work flows often fails in the same way as "implementation model" software whose interaction is based primarily on its internal technical structure. Because work flow is to business what structure is to programming, work flow-based design typically yields a kind of "business implementation model" that captures all of the functionality but little of the humanity.

Artifact models

As the name suggests, **artifact models** represent different artifacts that users employ in their tasks and work flows. Often these artifacts are online or paper forms. Artifact models typically capture commonalities and significant differences between similar artifacts for the purpose of extracting and replicating best practices in the eventual design. Artifact models can be useful later in the design process. Remember that direct

translation of paper systems to digital systems, without a careful analysis of goals and application of design principles (found in Part II of this book), usually leads to usability issues.

Physical models

Physical models, like artifact models, endeavor to capture elements of the user's environment. Physical models focus on capturing the layout of physical objects that comprise the user's workspace, which can provide insight into frequency-of-use issues and physical barriers to productivity. Good persona descriptions incorporate some of this information. But it may be helpful in complex physical environments (such as hospital floors and assembly lines) to create discrete, detailed physical models (maps or floor plans) of the user environment.

Personas and other models make sense of otherwise overwhelming and confusing user data. Now that you are empowered with sophisticated models as design tools, the next chapter will show you how to employ these tools to translate user goals and needs into workable design solutions.

Notes
1. Constantine and Lockwood, 2002
2. Grudin and Pruitt, 2002
3. Mikkelson and Lee, 2000
4. Grudin and Pruitt, 2002
5. Grudin and Pruitt, 2002
6. Constantine and Lockwood, 1999
7. Steinberg and Palmer, 2003
8. Beyer and Holtzblatt, 1998
9. Dillon, 2001
10. Goodwin, 2001

SETTING THE VISION: SCENARIOS AND DESIGN REQUIREMENTS

In the two previous chapters, we talked about how to gather qualitative information about users and create models using that information. By carefully analyzing user research and synthesis of personas and other models, we create a clear picture of our users and their respective goals, as well as their current situation. This brings us, then, to the crux of the whole method: how we use this understanding of people to create design solutions that satisfy and inspire users, while simultaneously addressing business goals and technical constraints.

Bridging the Research-Design Gap

Product teams often encounter a serious hurdle soon after they begin a new project. Things start out great. They gather a bunch of research data—or, more typically, hire someone to gather it for them—be it market research, user research, or competitive product research. Or perhaps they dispense with research, and brainstorm and collect a set of ideas that seem particularly cool or useful.

Conducting research certainly yields insights about users, and having brainstorming sessions is fun and gets the team inspired. But when it's time to make detailed design

and development decisions, the team rapidly realizes that they are missing a critical link between the research and the actual product design. Without a compass pointing the way through the research, without an organizing principle that highlights features and functions that are relevant to real users and describes *how they all fit together into a coherent product* that satisfies both user and business needs, no clear solution is in sight.

This chapter describes the first half of a process for bridging this research-design gap. It employs personas as the main characters in a set of techniques that rapidly arrive at design solutions in an iterative, repeatable, and testable fashion. This process has four major activities:

- Developing stories or *scenarios* as a means of imagining ideal user interactions
- Using those scenarios to extract *design requirements*
- Using these requirements in turn to define the product's fundamental *interaction framework*
- Filling in that framework with ever-increasing amounts of design detail

The glue that holds this process together, the compass in the wilderness of research data and potential product features, is *narrative*: using personas to create stories that point to user satisfaction.

Scenarios: Narrative as a Design Tool

Narrative, or storytelling, is one of the oldest human activities. Much has been written about the power of narrative to *communicate* ideas. However, narrative is also one of our most powerful creative methods. From a very young age, we are accustomed to using stories to think about possibilities, and this is an incredibly effective way to *imagine* a new and better future for our users. Imagining a story about a person using our product leverages our creativity to a greater power than when we just imagine a better form factor or configuration of screen elements. Furthermore, because of the intrinsically social aspect of narrative, it is a very effective and compelling way to share good ideas among team members and stakeholders. Ultimately, experiences designed around narrative tend to be more comprehensible and engaging for users because they are structured around a story.

Evidence of the effectiveness of narrative as a design tool is all around us. The famous Disney Imagineers would be lost without the modern-day myths they use as the foundation for the experiences they build. The experiences we create for digital products have their own (perhaps more grounded) narratives from which interactions can be built.

Much has been written about this idea. Brenda Laurel explored the concept of structuring interaction around dramatic principles in her book *Computers as Theatre*, in which she urges us to "...focus on designing the action. The design of objects, environments, and characters is all subsidiary to this central goal."[1] John Rheinfrank and Shelley Evenson also talk about the power of "stories of the future" for developing conceptually complex interactive systems,[2] and John Carroll has created a substantial body of work about scenario-based design, which we discuss later in this chapter.

Narrative also lends itself to effective visual depictions of interactive products. Interaction design is first and foremost the design of *behavior that occurs over time*. Therefore, a narrative structure combined with the support of fast and flexible visualization tools (such as the humble whiteboard) is perfectly suited for motivating, envisioning, representing, and validating interaction concepts.

Interaction design narratives are quite similar to the comic-book-like sequences called storyboards that are used in the motion picture industry. They share two significant characteristics: plot and brevity. Just as storyboards breathe life into a movie script, design solutions should be created and rendered to follow a plot—a story. Putting too much detail into the storyboards simply wastes time and money and has a tendency to tie us to suboptimal ideas simply because drawing them consumes significant resources (leaving less time for concept- and behavior-level refinement).

In the earliest phases of the process, we focus only on the "plot points," which allows us to be fluid as we explore design concepts. Because they are enough to convey the action and the potential experience, many millions of Hollywood dollars have been invested on the basis of simple pencil sketches or line drawings. By focusing on the narrative, we can quickly and flexibly arrive at a high-level design solution without getting bogged down by the inertia and expense inherent in high-production-value renderings. (However, such renderings are certainly appropriate once a working design framework is in place.)

Scenarios versus use cases and user stories

Scenarios and use cases are both methods of describing the user's interaction with a system. However, they serve very different functions. Goal-Directed scenarios are an iterative means of defining a product's *behavior* from the standpoint of specific users (personas). This includes not only the system's functionality, but the priority of functions and how those functions are expressed in terms of what the user sees and how she interacts with the system.

Use cases, on the other hand, are a technique based on exhaustive descriptions of the system's *functional* requirements, often of a transactional nature, focusing on low-level user action and accompanying system response.[3] The system's precise *behavior*—exactly *how* the system responds—typically is not part of a conventional or *concrete* use case; many

assumptions about the form and behavior of the system to be designed remain implicit.[4] Use cases permit a complete cataloging of user tasks for different classes of users but say little or nothing about how these tasks are presented to the user or how they should be prioritized in the interface. In our experience, the biggest shortcoming of traditional use cases as a basis for interaction design is their tendency to treat all possible user interactions as equally likely and important. This is indicative of their origin in software engineering rather than interaction design. They may be useful in identifying edge cases and for determining that a product is functionally complete, but they should be deployed only in the later stages of design validation.

User stories are used in agile programming methods, but typically they aren't actual stories or narratives. Rather, they are short sentences phrased like this: "As a user, I would like to log in to my online banking account." Typically this is followed by another couple of sentences briefly describing the necessary interface to accomplish the interaction. User stories are much more like informally phrased requirements than they are like scenarios; they don't describe the user's entire flow at a big-picture level or describe what the user's end goal is. Both of these are critical for stripping away unnecessary interactions and targeting what users really need (see Chapter 12 for more on this topic).

Scenarios are more akin to *epics* as described by agile methods. Like scenarios, epics do not describe task-level interactions, but rather broader and more far-reaching clusters of interactions that are intended to meet user goals. Epics focus more on function and presentation of user interfaces and interactions than they do on user behaviors. But in terms of scope and appropriate level of granularity, they have much more in common with scenarios than user stories do.

Scenario-based design

In the 1990s, substantial work was done by the HCI (human-computer interaction) community around the idea of use-oriented software design. From this work came the concept of the **scenario**, commonly used to describe a method of *design problem solving by concretization*: using a specific story to both construct and illustrate design solutions. These concepts are discussed by John M. Carroll in his book *Making Use*:

> *Scenarios are paradoxically concrete but rough, tangible but flexible ... they implicitly encourage "what-if?" thinking among all parties. They permit the articulation of design possibilities without undermining innovation ... Scenarios compel attention to the use that will be made of the design product. They can describe situations at many levels of detail, for many different purposes, helping to coordinate various aspects of the design project.[5]*

Carroll's use of *scenario-based design* describes how *users accomplish tasks*. It consists of an environmental *setting* and includes *agents* or *actors* who are abstracted stand-ins for users, with role-based names such as Accountant or Programmer.

Although Carroll certainly understands the power and importance of scenarios in the design process, we've found two shortcomings with scenarios as he approaches them:

- Carroll's concept of the actor as an abstracted, role-oriented model is insufficiently concrete to provide understanding of or empathy with users. It is impossible to design appropriate behaviors for a system without understanding its users in detail.

- Carroll's scenarios jump too quickly to the elaboration of tasks without considering the user's goals and motivations that drive and filter these tasks. Although Carroll does briefly discuss goals, he refers only to *goals of the scenario*. These goals are circularly defined as the completion of specific tasks. In our experience, user goals must be considered before user tasks can be identified and prioritized. Without addressing the motivation of human behavior, high-level product definition can be difficult and misguided.

The missing ingredient in Carroll's scenario-based design methods is the use of personas. A persona is a tangible representation of the user that acts as a believable agent in the setting of a scenario. In addition to reflecting current behavior patterns and motivations, personas let you explore how user motivations should influence and prioritize tasks in the future. Because personas model *goals* and not simply tasks, the scope of the problems addressed by scenarios can be broadened to include those related to product definition. They help answer the questions "What should this product do?" and "How should this product look and behave?"

Persona-based scenarios

Persona-based scenarios are concise narrative descriptions of one or more personas using a product or service to achieve specific goals. They allow us to start our designs from a story describing an ideal experience from the persona's perspective, focusing on people and how they think and behave, rather than on technology or business goals.

Scenarios can capture the *nonverbal dialog*[6] between the user and a product, environment, or system over time, as well as the structure and behavior of interactive functions. Goals serve as a filter for tasks and as a guide for structuring the display of information and controls during the iterative process of constructing the scenarios.

Scenario content and context are derived from information gathered during the Research phase and analyzed during the Modeling phase. Designers perform a type of role play in creating these scenarios, walking the personas through their future interactions with

the product or service,[7] almost similar to actors performing improvisation. This process leads to real-time synthesis of structure and behavior—typically at a whiteboard or on tablets—and later informs the detailed look and feel. Finally, personas and scenarios are used to test the validity of design ideas and assumptions throughout the process.

Three types of scenarios

The Goal-Directed Design method employs three types of persona-based scenarios at different points in the design process, each with a successively more interface-specific focus. The first—the *context scenario*—is used to explore, at a high level, how the product can best serve the needs of the personas. The context scenarios are created before any design sketching is performed. They are written from the persona's perspective, focusing on human activities, perceptions, and desires. It is when developing this kind of scenario that the designer has the most leverage to imagine an ideal user experience. More details about creating this type of scenario can be found in the section "Step 4: Construct context scenarios."

Once the design team has defined the product's functional and data elements and developed a Design Framework (as described in Chapter 5), a context scenario is revised. It becomes a *key path scenario* by more specifically describing user interactions with the product and by introducing the design's vocabulary. These scenarios focus on the most significant user interactions, always paying attention to how a persona uses the product to achieve its goals. Key path scenarios are iteratively refined along with the design as more and more detail is developed.

Throughout this process, the design team uses *validation scenarios* to test the design solution in a variety of situations. These scenarios tend to be less detailed and typically take the form of a number of what-if questions about the proposed solutions. Chapter 5 covers development and use of key path and validation scenarios.

Design Requirements: The "What" of Interaction

The Requirements Definition phase determines the *what* of the design: what information and capabilities our personas require to accomplish their goals. It is critical to define and agree on the *what* before we move on to the next question: *how* the product looks, behaves, operates, and feels. Conflating these two questions can be one of the biggest pitfalls in the design of an interactive product. Many designers are tempted to jump right into detailed design and render possible solutions. Regardless of how creative and skillful you are, we urge you not to do this. It runs the risk of leading to a never-ending cycle of iteration. Proposing a solution without clearly defining and agreeing on the problem

leaves you without a clear, objective method of evaluating the design's fitness. This in turn can lead to "I like" versus "you like" subjective differences within the product team and stakeholders, with no easy way to converge to consensus.

In lieu of such a method, you, your stakeholders, and your clients are likely to resort to taste and gut instinct, which have a notoriously low success rate with something as complex as an interactive product.

In other creative fields, the importance of defining the *what* first is well understood. Graphic novelists don't start with inking and coloring; they explore their characters and then outline and storyboard, roughly sketching both the narrative and form. That is exactly what we will do as we define our digital concepts.

DESIGN
PRINCIPLE

Define what the product will do before you design how the product will do it.

Design requirements aren't features

It's important to note that our concept of a "requirement" here is different from how the term is commonly used (and, we believe, *misused*) in the industry. In many product-development organizations, "requirement" has become synonymous with "feature" or "function." There is clearly a relationship between requirements and functions (which we leverage as a key part of our design process, as you will see in the next chapter). But we suggest that you think of *design requirements* as being synonymous with *needs*. Put another way, at this point, you want to rigorously define the human and business needs your product must satisfy.

Design requirements aren't specifications

Another industry use of the term "requirements" refers to a laundry list of capabilities generated, typically by product managers. These marketing requirements documents (MRDs) or product requirements documents (PRDs) are, when well executed, an attempt to describe the *what* of a product, but there are a few pitfalls. First, these lists are often only loosely connected to any kind of user research and quite frequently are generated without any serious exploration of user needs. Although the *what* described in these documents might (if you're lucky) reflect a coherent product, there's little guarantee that it will be a product the users find desirable.

Second, many MRDs and PRDs fall into the trap of confusing the *what* of the product with the *how*. Detailed descriptions of interfaces, such as "There should be a menu containing...," presuppose a solution that may be inappropriate for the user or his work flow.

Mandating solutions before the design process is a recipe for trouble, because it can easily lead to clunky and disjointed interactions and products.

For example, think about designing a data analytics tool to help an executive better understand the state of his business. If you jump right to the *how* without understanding the *what*, you might assume that the tool's output should be reports. It would be easy to come to this conclusion. If you performed user research, you probably would have noticed that reports are a widespread and accepted solution. However, if you imagine some scenarios and analyze your users' actual requirements, you might realize that your executive actually needs a way to recognize exceptional situations before opportunities are missed or problems arise. He or she also needs a way to understand emerging trends in the data. From here, it isn't difficult to see that static, flat reports are hardly the best way to meet these needs. With such a solution, your executive has to do the hard work of scrutinizing several of these reports to find the significant data underlying such exceptions and trends. Much better solutions might include data-driven exception reporting or real-time trend monitors.

The final problem with this kind of requirements document is that in itself it is of little use to either business stakeholders or developers. Without a way for them to visualize the contents of these lists—to see a *design* that reflects what the lists describe—neither stakeholders nor developers will have an easy time making decisions based on what is described.

Design requirements are strategic

In figuring out the best way to meet particular human needs by starting with requirements rather than solutions, interaction designers have an extraordinary amount of leverage to create powerful and compelling products. Separating problem and solution is an approach that provides maximum flexibility in the face of changing technological constraints and rising opportunities. By clearly defining user needs, designers can work with technologists to find the best viable and feasible solutions without compromising the product's ability to help people achieve their goals. Working in this manner, the product definition is not at risk when the implementation runs into problems. Also, it becomes possible to plan long-term technology development so that it can provide increasingly sophisticated ways of meeting user needs.

Design requirements come from multiple sources

We've already talked about personas and scenarios as a primary source of design requirements. While that may be the most important part of the requirements equation, other requirements also factor into the design. Business needs and constraints, as well as technical and legal constraints, typically are gathered during interviews with the

product's business and technical stakeholders. The next sections offer a more elaborate list of requirements.

The Requirements Definition Process

The translation of robust models into design solutions consists of two major phases. The *Requirements Definition*, shown in Figure 4-1, answers the broad questions about what a product is and what it should do. The *Framework Definition* answers questions about how a product behaves and how it is structured to meet user goals.

Figure 4-1: An overview of the Requirements Definition process

In this section, we'll discuss the Requirements Definition in detail. The Framework Definition is covered in Chapter 5. The methods described here are based on the persona-based scenario methodology developed by Robert Reimann, Kim Goodwin, Lane Halley, David Cronin, and Wayne Greenwood, and refined over the last decade by design practitioners at Cooper.

The Requirements Definition process consists of the following five steps (which are described in detail in the remainder of this chapter):

1 Create problem and vision statements
2 Explore/brainstorm
3 Identify persona expectations
4 Construct context scenarios
5 Identify design requirements

Although these steps proceed in roughly chronological order, they represent an iterative process. Designers can expect to cycle through Steps 3 through 5 several times until the requirements are stable. This is a necessary part of the process and shouldn't be short-circuited. A detailed description of each of these steps follows.

Step 1: Create problem and vision statements

Before beginning the process of ideation, it's important for designers to have a clear mandate for moving forward. While the Goal-Directed method aims to define products and services via personas, scenarios, and design requirements, it is often useful at this point to define in what direction these scenarios and requirements should be headed. We already have a sense of which users we're targeting and what their goals are, but lacking a clear product mandate, there is still considerable room for confusion. Problem and vision statements provide just such a mandate and are extremely helpful in building consensus among stakeholders before the design process moves forward.

At a high level, the *problem statement* defines the purpose of the design initiative.[8] A design problem statement should concisely reflect a situation that needs changing, for both the personas *and* the business providing the product to the personas. Often a cause-and-effect relationship exists between business concerns and persona concerns. For example:

> *Company X's customer satisfaction ratings are low. Market share has diminished by 10 percent over the past year because users have inadequate tools to perform tasks X, Y, and Z that would help them meet their goal of G.*

Connecting business issues to usability issues is critical to drive stakeholders' buy-in to design efforts and to frame the design effort in terms of both user and business goals.

The *vision statement* is an inversion of the problem statement that serves as a high-level design objective or mandate. In the vision statement, you lead with the user's needs, and you transition from those to how the design vision meets business goals. Here's a sample template for the preceding example's product redesign (similar wording works for new products as well):

> *The new design of Product X will help users achieve G by allowing them to do X, Y, and Z with greater [accuracy, efficiency, and so on], and without problems A, B, and C that they currently experience. This will dramatically improve Company X's customer satisfaction ratings and lead to increased market share.*

The content of both the problem and vision statements should come directly from research and user models. User goals and needs should be derived from the primary and secondary personas, and business goals should be extracted from stakeholder interviews.

Problem and vision statements are useful when you are redesigning an existing product. They also are useful for new-technology products or products being designed for unexplored market niches. With these tasks, formulating user goals and frustrations into

problem and vision statements helps establish team consensus and focus attention on the priorities of design activity to follow.

Step 2: Explore and brainstorm

At the early stages of the Requirements Definition, exploration, or brainstorming, assumes a somewhat ironic purpose. At this point in the project, we have been researching and modeling users and the domain for days or even months, and it is almost impossible to avoid having developed some preconceptions about what the solution looks like. However, ideally we'd like to create context scenarios without these prejudgments, and instead really focus on how our personas would likely want to engage with the product. We brainstorm at this stage to get these ideas out of our heads so that we can record them and thereby "let them go" for the time being.

The primary purpose here is to eliminate as much preconception as possible. Doing so allows designers to be open-minded and flexible as they use their imagination to construct scenarios and to use their analytical skills to derive requirements from these scenarios. A side benefit of brainstorming at this point in the process is that it switches your brain into "solution mode." Much of the work performed in the Research and Modeling phases is analytical in nature, and it takes a different mindset to come up with inventive designs.

Exploration, as the term suggests, should be unconstrained and uncritical. Air all the wacky ideas you've been considering (plus some you haven't), and be prepared to record them and file them for safekeeping until much later in the process. You don't know if any of them will be useful in the end, but you might find the germ of something wonderful that will fit into the design framework you later create.

It's also useful to cherry-pick some exploratory concepts to share with stakeholders or clients as a means to discover their true appetite for creative solutions and time horizons. If the stakeholders say they want "blue-sky thinking," you can use carefully selected exploratory concepts to test your blue-sky ideas and watch their reactions. If the discussion seems negative, you know to calibrate your thinking a bit more conservatively as you move forward with your scenarios.

Karen Holtzblatt and Hugh Beyer describe a facilitated method for brainstorming that can be useful for getting an exploration session started, especially if your team includes stakeholders, clients, or even developers who are eager to get started thinking about solutions.[9]

Don't spend too much time on the brainstorming step. A few hours for simple projects to a couple of days for a project of significant scope or complexity should be more than sufficient for you and your teammates to get all those crazy ideas out of your systems. If

you find your ideas getting repetitious, or the popcorn stops popping, that's a good time to stop.

Step 3: Identify persona expectations

As we discussed in Chapter 1, a person's *mental model* is her own internal representation of reality—how she thinks about or explains something to herself. Mental models are deeply ingrained, are almost subliminal in terms of a person's self-awareness of them, and are frequently the result of a lifetime of cumulative experiences. People's expectations about a product and how it works are highly informed by their mental model.

It is therefore important that the *represented model* of our interfaces—how the design behaves and presents itself—should match what we understand about users' mental models as much as possible. The represented model should not reflect the *implementation model*—how the product is actually constructed internally.

To accomplish this, we formally record these expectations. They are an important source of design requirements. For each primary and secondary persona, we identify the following:

- Attitudes, experiences, aspirations, and other social, cultural, environmental, and cognitive factors that influence the persona's expectations
- General expectations and desires the persona may have about the experience of using the product
- Behaviors the persona will expect or want from the product
- How that persona thinks about basic elements or units of data (For example, in an e-mail application, the basic elements of data might be messages and people.)

Your persona descriptions may contain enough information to answer these questions directly; however, your research data will remain a rich resource. Use it to analyze how interview subjects define and describe objects and actions that are part of their usage patterns, along with the language and grammar they use. Here are some things to look for:

- What do the interview subjects mention first?
- Which action words (verbs) do they use? What nouns?
- Which intermediate steps, tasks, or objects in a process *don't* they mention? (Hint: These might not be terribly important to how they think about things.)

Step 4: Construct context scenarios

All scenarios are stories about people and their activities, but *context scenarios* are the most storylike of the three types we employ.

A context scenario tells the story of a particular user persona, with various motivations, needs, and goals, using the future version of your product in the way that is most typical for that persona. It describes the broad context in which that persona's usage patterns are exhibited. It includes environmental and organizational (in the case of enterprise systems) considerations.[10] A successful context scenario captures all of these attributes and addresses them in the extrapolated work flow narrative you create.

As we've discussed, *this is where design begins.* As you develop context scenarios, you should focus on how the product you are designing can best help your personas achieve their goals. Context scenarios establish the primary touch points that each primary and secondary persona has with the system (and possibly with other personas) over the course of a day or some other meaningful length of time.

Context scenarios should be broad and relatively shallow in scope. They should not describe product or interaction detail but rather should focus on high-level actions from the user's perspective. It is important to map out the big picture first so that we can systematically identify design requirements. Only then can we design appropriate interactions and interfaces.

Context scenarios address questions such as the following:

- In what setting(s) will the product be used?
- Will it be used for extended amounts of time?
- Is the persona frequently interrupted?
- Do several people use a single workstation or device?
- With what other products will it be used?
- What primary activities does the persona need to perform to meet her goals?
- What is the expected end result of using the product?
- How much complexity is permissible, based on persona skill and frequency of use?

Context scenarios should *not* represent product behaviors as they currently are. These scenarios represent the brave new world of Goal-Directed products, so, especially in the initial phases, focus on addressing the personas' goals. Don't yet worry about exactly *how* things will get accomplished. Initially you should treat the design as a bit of a magic black box.

In most cases, more than one context scenario is necessary. This is true especially when there are multiple primary personas, but sometimes even a single primary persona may have two or more distinct contexts of use.

Context scenarios are also entirely *textual*. We are not yet discussing form, only the behaviors of the user and the system. This discussion is best accomplished as a textual narrative, saving the "how" for later refinement steps.

A sample context scenario

The following is the first iteration of a context scenario for a primary persona for a personal digital assistant (PDA) type phone, including both the device and its service. Our persona is Vivien Strong, a real-estate agent in Indianapolis, whose goals are to balance work and home life, close the deal, and make each client feel like he or she is her *only* client.

Here is Vivien's context scenario:

1 While getting ready in the morning, Vivien uses her phone to check her e-mail. Because it has a relatively large screen and quick connection time, it's more convenient than booting up a computer as she rushes to make her daughter, Alice, a sandwich for school.

2 Vivien sees an e-mail from her newest client, Frank, who wants to look at a house this afternoon. The device has his contact info, so she can call him with a simple action right from the e-mail.

3 While on the phone with Frank, Vivien switches to speakerphone so she can view the screen while talking. She looks at her appointments to see when she's free. When she creates a new appointment, the phone automatically makes it an appointment with Frank, because it knows with whom she is talking. She quickly enters the address of the property into the appointment as she finishes her conversation.

4 After sending Alice to school, Vivien heads into the real-estate office to gather some papers for another appointment. Her phone has already updated her Outlook appointments, so the rest of the office knows where she'll be in the afternoon.

5 The day goes by quickly, and eventually Vivien is running a bit late. As she heads toward the property she'll be showing Frank, the phone alerts her that her appointment is in 15 minutes. When she flips open the phone, she sees not only the appointment, but also a list of all documents related to Frank, including e-mails, memos, phone messages, and call logs to Frank's number. Vivien initiates a call, and the phone automatically connects to Frank because it knows her appointment with him is soon. She lets him know she'll be there in 20 minutes.

6 Vivien knows the address of the property but is unsure exactly where it is. She pulls over and taps the address she put into the appointment. The phone downloads directions along with a thumbnail map showing her location relative to the destination.

7 Vivien gets to the property on time and starts showing it to Frank. She hears the phone ring from her purse. Normally while she is in an appointment, the phone automatically goes to voicemail, but Alice has a code she can press to get through. The phone knows it's Alice calling, so it uses a distinctive ringtone.

8 Vivien takes the call. Alice missed the bus and needs to be picked up. Vivien calls her husband to see if he can do it. She gets his voicemail; he must be out of service range. She tells him she's with a client and asks if he can get Alice. Five minutes later the phone sounds a brief tone. Vivien recognizes it as her husband's; she sees he's sent her an instant message: "I'll get Alice; good luck on the deal!"

Notice how the scenario remains at a fairly high level, without getting too specific about interfaces or technologies. It's important to create scenarios that are within the realm of technical possibility, but at this stage the details of reality are unimportant. We want to leave the door open for truly novel solutions, and it's always possible to scale back; we are ultimately trying to describe an *optimal*, yet still feasible, experience. Also notice how the activities in the scenario tie back to Vivien's goals and try to eliminate as many tasks as possible.

Pretending it's magic

A powerful tool in the early stages of developing scenarios is to *pretend the interface is magic*. If your persona has goals and the product has magic powers to meet them, how simple could the interaction be? This kind of thinking is useful in helping designers think outside the box. Magic solutions obviously won't suffice, but figuring out creative ways to technically accomplish interactions that are as close to magic solutions as possible (from the personas' perspective) is the essence of great interaction design. Products that meet goals with a minimum of hassle and intrusion seem almost magical to users. Some of the interactions in the preceding scenario may seem a bit magical, but all are possible with technology available today. It's the goal-directed behavior, not the technology alone, that provides the magic.

> **DESIGN PRINCIPLE**   *In the early stages of design, pretend the interface is magic.*

Step 5: Identify design requirements

After you are satisfied with an initial draft of your context scenario, you can analyze it to extract the personas' needs or design requirements. These design requirements can be thought of as consisting of *objects*, *actions*, and *contexts*.[11] And remember, as we've discussed, we prefer not to think of requirements as identical to features or tasks. Thus, a requirement from the preceding scenario might read as follows:

> *Call (action) a person (object) directly from an appointment (context).*

If you are comfortable extracting needs in this format, it works quite well. Otherwise, you may find it helpful to separate them into data, functional, and contextual requirements, as described in the following sections.

Data requirements

Personas' data needs are the objects and information that must be represented in the system. Using the semantics just described, it is often useful to think of data requirements as the objects and adjectives related to those objects. Common examples include accounts, people, addresses, documents, messages, songs, and images, as well as attributes of those, such as status, dates, size, creator, and subject.

Functional requirements

Functional needs are the operations or actions that need to be performed on the system's objects and that typically are translated into interface controls. These can be thought of as the product's *actions*. Functional needs also define places or containers where objects or information in the interface must be displayed. (These clearly are not actions in and of themselves but usually are suggested by actions.)

Contextual requirements

Contextual requirements describe relationships or dependencies between sets of objects in the system. This can include which objects in the system need to be displayed together to make sense for work flow or to meet specific persona goals. (For example, when choosing items for purchase, a summed list of items already selected for purchase should probably be visible.) Other contextual requirements may include considerations regarding the physical environment the product will be used in (an office, on the go, in harsh conditions) and the skills and capabilities of the personas using the product.

Other requirements

After you've gone through the exercise of pretending it's magic, it's important to get a firm idea of the realistic requirements of the business and technology you are designing for. (But we hope that designers have some influence over technology choices when the choice directly affects user goals.)

- **Business requirements** can include stakeholder priorities, development timelines, budgetary and resource constraints, regulations and legal considerations, pricing structures, and business models.

- **Brand and experience requirements** reflect attributes of the experience you want users and customers to associate with your product, company, or organization.

- **Technical requirements** can include weight, size, form factor, display, power constraints, and software platform choices.

- **Customer and partner requirements** can include ease of installation, maintenance, configuration, support costs, and licensing agreements.

Having followed Steps 1 through 5, you should now have a rough, creative overview of how the product will address user goals in the form of context scenarios, as well as a reductive list of needs and requirements extracted from your research, user models, and the scenarios. These design requirements not only provide a design and development direction but also provide a scope of work to communicate to stakeholders. Any new design requirements after this point must necessarily change the scope of work.

Now you are ready to delve deeper into the details of your product's behaviors and begin to consider how the product and its functions will be represented. You are ready to define the framework of the interaction.

Notes

1. Laurel, 2013
2. Rheinfrank and Evenson, 1996
3. Wirfs-Brock, 1993
4. Constantine and Lockwood, 1999
5. Carroll, 2001
6. Buxton, 1990
7. Verplank, et al, 1993
8. Newman and Lamming, 1995
9. Holtzblatt and Beyer, 1998
10. Kuutti, 1995
11. Shneiderman, 1998

DESIGNING THE PRODUCT: FRAMEWORK AND REFINEMENT

In the preceding chapter, we talked about the first part of the design process: developing scenarios to imagine ideal user interactions and then defining requirements from these scenarios and other sources. Now we're finally ready to *design*.

Creating the Design Framework

Rather than jump into the nuts and bolts right away, we want to stay at a high level and concern ourselves with the overall structure of the user interface and associated behaviors. We call this phase of the Goal-Directed process the Design Framework. If we were designing a house, at this point we'd be concerned with what rooms the house should have, where they should be positioned, and roughly how big they should be. We would not be worried about the precise measurements of each room or things like the doorknobs, faucets, and countertops.

The *Design Framework* defines the overall structure of the users' experience. This includes the underlying organizing principles and the arrangement of functional elements on the screen, work flows, interactive behaviors and the visual and form languages used to express information, functionality, and brand identity. In our experience,

form and behavior must be designed in concert; the Design Framework is made up of an interaction framework, a visual design framework, and sometimes an industrial design framework. At this phase in a project, interaction designers use scenarios and requirements to create rough sketches of screens and behaviors that make up the *interaction framework*. Concurrently, visual designers use visual language studies to develop a *visual design framework* that is commonly expressed as a detailed rendering of a single screen archetype.

Other specialists on the team may be working on frameworks of their own. Industrial designers execute form language studies to work toward a rough physical model and *industrial design framework*. Service designers build models of the information exchange for each touch point in a *service framework*. Each of these processes is addressed in this chapter.

When it comes to the design of complex behaviors and interactions, we've found that focusing too early on pixel pushing, widget design, and specific interactions can get in the way of effectively designing a comprehensive framework in which all the product's behaviors can fit. Instead we should take a top-down approach, concerning ourselves first with the big picture and rendering our solutions without specific detail in a low-fidelity manner. Doing so can ensure that we and our stakeholders focus initially on the fundamentals: serving the personas' goals and requirements.

Revision is a fact of life in design. Typically, the process of representing and presenting design solutions helps designers and stakeholders refine their vision and understanding of how the product can best serve human needs. The trick, then, is to render the solution in only enough detail to provoke engaged consideration, without spending too much time or effort elaborating details that are certain to be modified or abandoned. We've found that sketch-like storyboards of context and screens, accompanied by narrative in the form of scenarios, are a highly effective way to explore and discuss design solutions without creating undue overhead and inertia.

Research about the usability of architectural renderings supports this notion. A study of people's reactions to different types of CAD images found that pencil-like sketches encouraged discourse about a proposed design and also increased understanding of the renderings as representing work in progress.[1] Carolyn Snyder covers this concept at length in *Paper Prototyping* (Morgan Kaufmann, 2003), where she discusses the value of such low-fidelity presentation techniques in gathering user feedback. While we believe that usability testing and user feedback are often most constructive during design refinement, sometimes they are useful as early as the Framework phase. (More discussion of usability testing can be found later in this chapter.)

Defining the interaction framework

The interaction framework defines not only the high-level structure of screen layouts but also the product's flow, behavior, and organization. The following six steps describe the process of defining the interaction framework (see Figure 5-1):

1. Define form factor, posture, and input methods.
2. Define functional and data elements.
3. Determine functional groups and hierarchy.
4. Sketch the interaction framework.
5. Construct key path scenarios.
6. Check designs with validation scenarios.
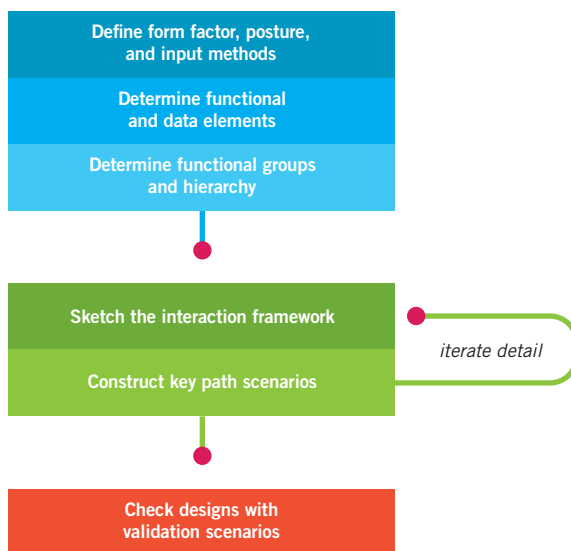
Figure 5-1: The Framework Definition process

Even though we've broken the process into numerically sequenced steps, typically this is not a linear effort. Rather, it occurs in iterative loops. In particular, Steps 3 through 5 may be switched, depending on the designer's thought process (more on this later). The six steps are described in the following sections.

Step 1: Define form factor, posture, and input methods

The first step in creating a framework is to define the *form factor* of the product you'll be designing. Is it a web application that will be viewed on a high-resolution computer screen? Is it a phone that must be small, light, low-resolution, and visible in both bright

sunlight and the dark? Is it a kiosk that must be rugged to withstand a public environment while accommodating thousands of distracted novice users? What constraints does each of these form factors imply for any design? Each has clear implications for the product's design, and answering this question sets the stage for all subsequent design efforts. If the answer isn't obvious, look to your personas and scenarios to better understand the ideal usage context and environment. Where a product requires the design of both hardware and software, these decisions also involve industrial design considerations. Later in the chapter we discuss how to coordinate interaction design with industrial design.

As you define the form, you should also define the product's basic *posture* and determine the system's input method(s). A product's posture is related to how much attention the user will devote to interacting with the product and how the product's behaviors respond to the kind of attention the user will devote to it. This decision should be based on usage contexts and environments, as described in your context scenarios (see Chapter 4). We discuss the concept of posture in greater depth in Chapter 9.

The input method is how users will interact with the product. This is driven by the form factor and posture, as well as by your personas' attitudes, aptitudes, and preferences. Choices include keyboard, mouse, keypad, thumb-board, touchscreen, voice, game controller, remote control, dedicated hardware buttons, and many other possibilities. Decide which combination is appropriate for your primary and secondary personas. If you need to use a combination of input methods (such as the common website or desktop application that relies on both mouse and keyboard input), decide on the product's *primary* input method.

Step 2: Define functional and data elements

Functional and data elements represent functionality and information that are revealed to the user in the interface. These are the concrete manifestations of the functional and data requirements identified during the Requirements Definition phase, as described in Chapter 4. Whereas the requirements were purposely described in general terms, from the personas' perspective, functional and data elements are described in the language of user-interface representations. It is important to note that each element must be defined in response to specific requirements defined earlier. This is how we ensure that every aspect of the product we are designing has a clear purpose that can be traced back to a usage scenario or business goal.

Data elements typically are the fundamental subjects of interactive products. These objects—such as photos, e-mail messages, customer records, or orders—are the basic units to be referred to, responded to, and acted on by the people using the product. Ideally they should fit with the personas' mental models. At this point, it is critical to comprehensively catalog the data objects, because the product's functionality is commonly defined in relation to them. We are also concerned with the objects' significant

attributes, such as the sender of an e-mail message or the date a photo was taken. But it is less important to be comprehensive about the attributes at this point, as long as you have an idea whether the personas care about a few attributes or a lot. It can be helpful at this point to involve your team's software architect, who can use this Goal-Directed data model to create a more formal data object model for later use by developers. We'll discuss development touch points more in Chapter 6.

It's useful to consider the relationships between data elements. Sometimes a data object may contain other data objects; other times there may be a more associative relationship between objects. Examples of such relationships include a photo within an album, a song within a playlist, or an individual bill within a customer record. These relationships can be documented as simply as creating indented bulleted lists. For more complex relationships, more elaborate "box and arrow" diagrams may be appropriate.

Functional elements are the operations that can be done to the data elements and their representations in the interface. Generally speaking, they include tools to act on and ways to visually and structurally manage data elements. The translation of functional requirements into functional elements is where we start making the design concrete. While the context scenario is the vehicle to imagine the overall experience we will create for our users, this is where we begin to make that experience real.

It is common for a single requirement to necessitate multiple interface elements. For example, Vivien, our persona for a smartphone from Chapter 4, needs to be able to call her contacts. The following functional elements meet that need:

- Voice activation (voice data associated with the contact)
- Assignable quick-dial buttons
- Selecting a contact from a list
- Selecting a contact from an e-mail header, appointment, or memo
- Auto-assigning a call button in the appropriate context (for example, for an upcoming appointment)

Again, it is imperative to return to context scenarios, persona goals, and mental models to ensure that your solutions are appropriate for the situation at hand. This is also where design principles and patterns begin to become a useful way to arrive at effective solutions without reinventing the wheel. You also must exercise your creativity and design judgment here. In response to any identified user requirement, typically a number of solutions are possible. Ask yourself which of the possible solutions is most likely to do the following:

- Accomplish user goals most efficiently.
- Best fit your design principles.

- Fit within technology or cost parameters.

- Possibly differentiate the interaction from the competition.

- Best fit other requirements.

Pretend the product is human

As you saw in Chapter 4, pretending that a tool, product, or system is *magic* is a powerful way to imagine the ideal user experience to be reflected in concept-level context scenarios. In the same way, pretending that the system is *human* is a powerful way to structure interaction-level details. This simple principle is discussed in detail in Chapter 8. In a nutshell, interactions with a digital system should be similar in tone and helpfulness to interactions with a polite, considerate human.[2] As you determine the interactions and behavior along with the functional elements and groupings, you should ask yourself these questions: What would a helpful human do? What would a thoughtful, considerate interaction feel like? Does the product treat the primary persona humanely? How can the software offer helpful information without getting in the way? How can it minimize the persona's effort in reaching his goals?

For example, a mobile phone that behaves like a considerate person knows that, after you've completed a call with a number that isn't in your contacts, you may want to save the number. Therefore, the phone provides an easy and obvious way to do so. An inconsiderate phone forces you to scribble the number on the back of your hand as you go into your contacts to create a new entry.

Apply principles and patterns

Critical to translating requirements into functional elements (as well as grouping these elements and exploring detailed behavior in scenarios and storyboards) is applying general interaction principles and specific interaction patterns. These tools leverage years of interaction design experience of designers working on similar problems. Neglecting to take advantage of such knowledge means wasting time on problems whose solutions are well known. Additionally, deviating from standard design patterns can create a product where the users must learn every interaction idiom from scratch, rather than recognizing behaviors from other products and leveraging their own experience. (We discuss the idea of design patterns in Chapter 7.) Of course, sometimes it is appropriate to invent new solutions to common problems. But as we discuss further in Chapter 17, you should obey standards unless you have a good reason not to.

Scenarios provide an inherently top-down approach to interaction design. They iterate through successively more detailed design structures, from main screens down to tiny subpanes or dialogs. Principles and patterns add a more bottom-up approach to balance the process. Principles and patterns can be used to organize elements at all levels of the

design. Chapter 7 discusses the uses and types of principles and patterns in detail. Part II of this book provides a wealth of useful interaction principles appropriate to this step in the process.

Step 3: Determine functional groups and hierarchy

After you have a good list of top-level functional and data elements, you can begin to group them into functional units and determine their hierarchy.[3] Because these elements facilitate specific tasks, the idea is to group elements to best facilitate the persona's flow (see Chapter 11) both within a task and between related tasks. Here are some issues to consider:

- Which elements need a large amount of screen real estate, and which do not?
- Which elements are *containers* for other elements?
- How should containers be arranged to optimize flow?
- Which elements are used together, and which are not?
- In what sequence will a set of related elements be used?
- What data elements would be useful for the persona to know or reference at each decision?
- What interaction patterns and principles apply?
- How do the personas' mental models affect organization?

At this point it's important to organize data and functions into top-level container elements, such as screens, frames, and panes. These groupings may change somewhat as the design evolves (particularly as you sketch the interface), but it's still useful to provisionally sort elements into groups. This will speed up the process of creating initial sketches. Again, indented lists or simple Venn diagrams are appropriate at this point for documenting these relationships.

Consider which primary screens or states (which we'll call *views*) the product requires. Initial context scenarios give you a feel for what these might be. If you know that the user has several end goals where data and functionality don't overlap, it might be reasonable to define separate views to address them. On the other hand, if you see a cluster of related needs (for example, to make an appointment, or to review nearby restaurants, or if your persona needs to see a calendar and contacts), you might consider defining a view that incorporates all these.

When grouping functional and data elements, consider how they should be arranged given the product's platform, posture, screen size, form factor, and input methods. Containers for objects that must be compared or used together should be adjacent. Objects representing steps in a process should, in general, be adjacent and ordered sequentially.

Using detailed interaction design principles and patterns is extremely helpful at this juncture. Part III of this book provides many principles that can be of assistance at this stage of organization.

Step 4: Sketch the interaction framework

Now we're ready to sketch the interface. This visualization of the interface should be simple at first. Around the studio, we often call this the rectangles phase. Our sketches start by subdividing each view into rough rectangular areas corresponding to panes, control components (such as toolbars), and other top-level containers, as shown in Figure 5-2. Label the rectangles, and illustrate and describe how one grouping or element affects others. Draw arrows from one set of rectangles to others to represent flows or state changes.
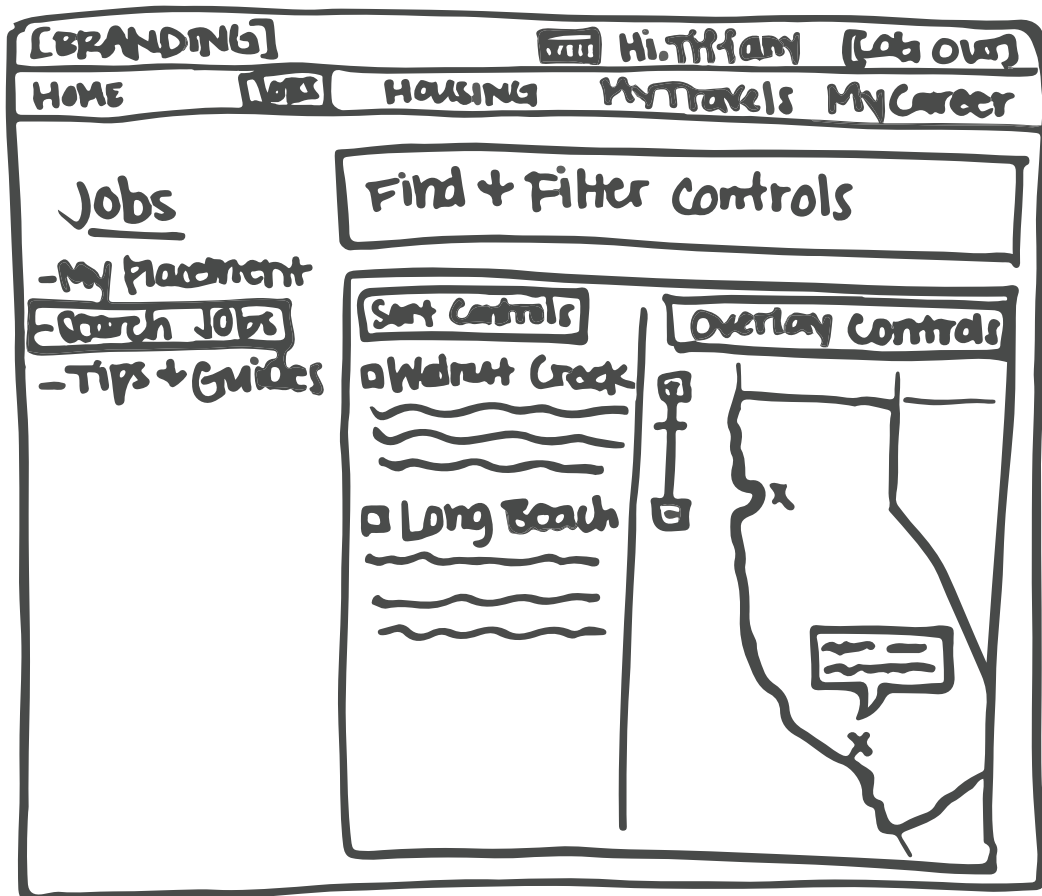
Figure 5-2: An early framework sketch from designs Cooper created for Cross Country TravCorps, an online portal for traveling nurses. Framework sketches should be simple, starting with rectangles, names, and brief descriptions of relationships between functional areas. Details can be visually hinted at to give an idea of the contents, but don't fall into the trap of designing detail at this stage.

You may want to sketch different ways of fitting together top-level containers in the interface. This visualization of the interface should be simple at first: boxes representing each functional group and/or container with names and descriptions of the relationships between the different areas (see Figure 5-2).

Be sure to look at the entire top-level framework first. Don't become distracted by the details of a particular area of the interface (although *imagining* what goes into each container will help you decide how to arrange elements and allocate real estate). You will have plenty of time to explore the design at the widget level later. Trying to do so too soon may risk creating a lack of coherence in the design as you move forward. At this high-level "rectangle phase," it's easy to explore a variety of ways to present information and functionality and to perform radical reorganizations if necessary. It's often useful to try several arrangements, running through validation scenarios (see the later section describing Step 6), before landing on the best solution. Spending too much time and effort on intricate details early in the design process discourages designers from changing course to what might be a superior solution. It's easier to discard your work and try another approach when you don't have a lot of effort invested.

Sketching the framework is an iterative process that is best performed with a small, collaborative group. This group includes one or two interaction designers (or ideally an interaction designer and a "design communicator"—someone who thinks in terms of the design narrative) and a visual or industrial designer.

We've found a few tool choices that work well during the sketching phase. Working at a whiteboard promotes collaboration and discussion—and, of course, everything is easy to erase and redraw. A digital camera provides a quick and easy means to capture ideas for later reference.

In recent years we've also grown fond of using tablet computers with OneNote connected to a shared monitor for our initial sketches. Whatever tool you use, it needs to be fast, collaborative, visible to everyone on the team, and easy to iterate and share.

Once the sketches reach a reasonable level of detail, it becomes useful to start rendering in a computer-based tool. Each tool has its strengths and weaknesses, but those commonly used to render high-level interface sketches currently include Adobe Fireworks, Adobe Illustrator, Microsoft Visio, Microsoft PowerPoint, Axure, and Omni Group's OmniGraffle. The key is to find the tool that is most comfortable for you so that you can work quickly, roughly, and at a high level. We've found it useful to render drawings in a visual style that suggests the sketchiness of the proposed solutions. (Recall that rough sketches tend to do a better job of promoting discourse about design.) It is also critical to be able to easily render several related, sequential screen states to depict the product's behavior in the key path scenario. (The "states" construct in Fireworks makes it a particularly good tool for doing this.)

Step 5: Construct key path scenarios

A *key path scenario* describes how the persona interacts with the product, using the vocabulary of the interaction framework. These scenarios depict the primary pathways through the interface that the persona takes with the greatest frequency, often on a daily basis. For example, in an e-mail application, key path activities include viewing and composing mail, not configuring a new mail server.

These scenarios typically evolve from the context scenarios, but here we specifically describe the persona's interaction with the various functional and data elements that make up the interaction framework. As we add more and more detail to the interaction framework, we iterate the key path scenarios to reflect this detail in greater specificity around user actions and product responses.

Unlike the goal-oriented context scenarios, key path scenarios are more task-oriented, focusing on task details broadly described and hinted at in the context scenarios. (In this way they are similar to Agile use cases.) This doesn't mean that we can ignore goals. Goals and persona needs are the constant measuring stick throughout the design process, used to trim unnecessary tasks and streamline necessary ones. However, *key path scenarios* must describe in detail the behavior of each major interaction and provide a walkthrough of each major pathway.

Storyboarding

By using a sequence of low-fidelity sketches accompanied by the narrative of the key path scenario, you can richly portray how a proposed design solution helps personas accomplish their goals, as shown in Figure 5-3. This technique of *storyboarding* is borrowed from filmmaking and cartooning, where a similar process is used to plan and evaluate ideas without having to deal with the cost and labor of shooting actual film. Each interaction between the user and the product can be portrayed on one or more frames or slides. Advancing through them provides a reality check of the interactions' coherence and flow.

Process variations and iteration

Because creative human activities are rarely a sequential, linear process, the steps in the Framework phase shouldn't be thought of as a simple sequence. It is common to move back and forth between steps and to iterate the whole process several times until you have a solid design solution. Depending on how you think, you have a couple of different ways to approach Steps 3 through 5. You may find that one works better for you than another.
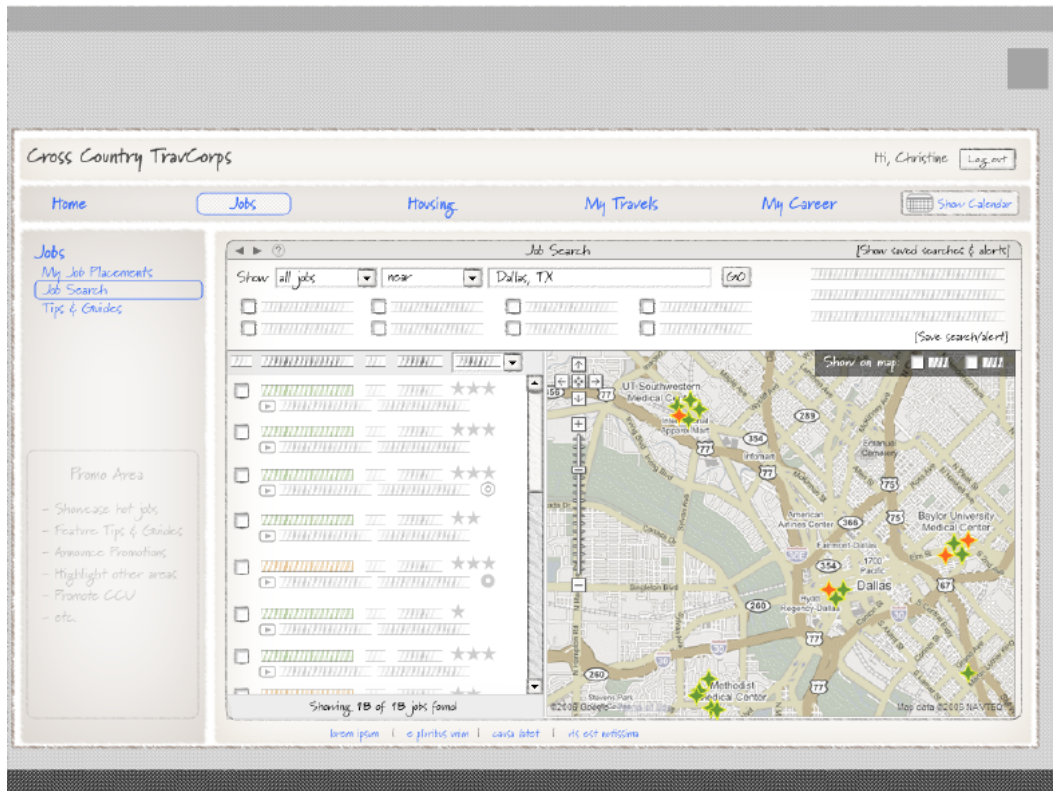
Figure 5-3: A more evolved Framework rendering from the Cross Country TravCorps job search web application

Verbal thinkers may want to use the scenario to drive the process and approach Steps 3 through 5 in the following sequence:

1. Key path scenarios
2. Work out the groupings verbally
3. Sketch

Visual thinkers may find that starting from the illustration will help them make sense of the other parts of the process. They may find this easier:

1. Sketch
2. Key path scenarios
3. See if your groupings work with the scenarios.

Step 6: Check designs with validation scenarios

After you have storyboarded your key path scenarios and adjusted the interaction framework until the scenario flows smoothly and you're confident that you're headed in the right direction, it is time to shift focus to less frequent or less important interactions. These *validation scenarios* typically are not developed in as much detail as key path scenarios. Rather, this phase consists of asking a series of what-if questions. The goal is to poke holes in the design and adjust it as needed (or throw it out and start over). You should address three major categories of validation scenarios, in the following order:

- **Alternative scenarios** are alternative or less-traveled interactions that split off from key pathways at some point along the persona's decision tree. These could include commonly encountered exceptions, less frequently used tools and views, and variations or additional scenarios based on the goals and needs of secondary personas. Returning to our smartphone scenario from Chapter 4, an example of a key path variant would be if Vivien decided to respond to Frank by e-mail in Step 2 instead of calling him.

- **Necessary-use scenarios** include actions that *must* be performed, but only infrequently. Purging databases, upgrading a device, configuring, and making other exceptional requests might fall into this category. Necessary-use interactions demand pedagogy because they are seldom encountered: Users may forget how to access the function or how to perform tasks related to it. However, this rare use means that users won't require parallel interaction idioms such as keyboard equivalents—nor do such functions need to be user-customizable. An example of a necessary-use scenario for the design of a smartphone is if the phone was sold secondhand, requiring the removal of all personal information associated with the original owner.

- **Edge-case use scenarios**, as the name implies, describe atypical situations that the product must nevertheless be able to handle, albeit infrequently. Developers focus on edge cases because they often represent sources of system instability and bugs and typically require significant attention and effort. Edge cases should never be the focus of the design effort. Designers can't ignore edge-case functions and situations, but the interaction needed for them is of much lower priority and usually is buried deep in the interface. Although the *code* may succeed or fail on its capability to successfully handle edge cases, the *product* will succeed or fail on its capability to successfully handle daily use and necessary cases. Returning again to Vivien's smartphone (in Chapter 4), an example of an edge-case scenario would be if Vivien tried to add two different contacts who have the same name. This is not something she is likely to do, but it is something the phone should handle if she does.

Defining the visual design framework

As the interaction framework establishes an overall structure for product behavior, and for *the form as it relates to behavior*, a parallel process focused on the visual and industrial design is also necessary to prepare for detailed design unless you're working with a

well-established visual style. This process follows a trajectory similar to the interaction framework, in that the solution is first considered at a high level and then narrows to an increasingly granular focus. Chapter 17 provides further details on the integration of visual design and interaction design.

The *visual design framework* typically follows this process:

1 Develop experience attributes.

2 Develop visual language studies.

3 Apply the chosen visual style to the screen archetype.

Step 1: Develop experience attributes

The first step in defining a visual design framework to choose a set of three to five adjectives that will be used to help define the tone, voice, and brand promise of the product. (There is a strategy discussion to have if these attributes don't fit the persona's goals and interests.) This set of descriptive keywords are collectively called *experience attributes.*

Visual designers usually lead the development of experience attributes, as interaction designers are more accustomed to thinking about product behavior than brand. It's often a good idea to involve stakeholders in this process, or at least to get their input at the onset. The process used at Cooper for generating experience attributes is as follows:

1 Gather any existing brand guidelines. Familiarize yourself with them. If the company has clear brand guidelines built around one product—the product you're designing— much of your work may have already been done for you.

2 Gather together examples of strongly branded products, interfaces, objects, and services. Including multiple examples from particular domains will help stakeholders think about their differences. If we include images of cars, for instance, we might include examples from BMW, Toyota, Ferrari, and Tesla.

3 Work with stakeholders to identify direct and indirect competition. Gather examples of these products and services' products and interfaces to include in your examples.

4 Pull relevant terms mentioned by interviewees in the course of your qualitative research. Pay particular attention to any pain points mentioned. For instance, if many mention that a competitor or the existing version of the product is hard to use or "unintuitive," you may want to discuss whether "friendly," "easy," or "understandable" should be an attribute.

5 With the brand guidelines, example products, competition, and user notes on display to reference, have a discussion with stakeholders about the sub-brand of the product you're designing. We often ask stakeholders to vote for and against examples by

placing red or green stickers on them, and then discuss any clear winners, losers, or controversial examples.

6 From the outcomes of this discussion, identify the minimum number of adjectives that define and distinguish the product.

7 If any of the words have multiple meanings, document the exact sense intended. "Sharp," for instance, could refer to precision and sleekness, or it could mean intelligence and wit.

8 Consider competitors. If your set of attributes does not distinguish the brand from competitors, refine them until they do. Also make sure that individual attributes are aspirational. "Smart" is good. "Brilliant" is better.

9 Check back with the stakeholders (and especially any marketers) on your proposed attribute set to discuss and finalize them before moving forward.

Step 2: Develop visual language studies

The next step is to explore a variety of visual treatments through *visual language studies*, as shown in Figure 5-4. These studies are based on the experience attributes and include color, type, and widget treatments. They also include the overall dimensionality and any "material" properties of the interface (for example, does it feel like glass or paper?).

These studies should show these aspects abstractly and independent of the interaction design, because our goal is to assess the overall tone and suitability for general interactions. We also want to avoid possibly distracting our stakeholders with highly rendered versions of rough interaction designs.

Visual language studies should relate to the personas' experience goals, as well as to any experience or brand keywords that were developed in the Requirements Definition phase (see Chapter 4). Commonly, a company's brand guidelines form a good starting point for this activity. But it should be noted that brand guidelines rarely consider the interactive experience and may not account for the differences in multiple software products. "Brand guidelines" commonly consist of a document explaining how a company's brand identity should be conveyed visually and textually.

Substantial work is often required to translate a style guide for marketing collateral into a meaningful look and feel for an interactive product or website. It's also important to consider environmental factors and persona aptitudes when devising visual styles. Screens that must be visible under bright lights or from a distance require high contrast and more saturated colors. The elderly and other sight-impaired users require larger and more readable typefaces.
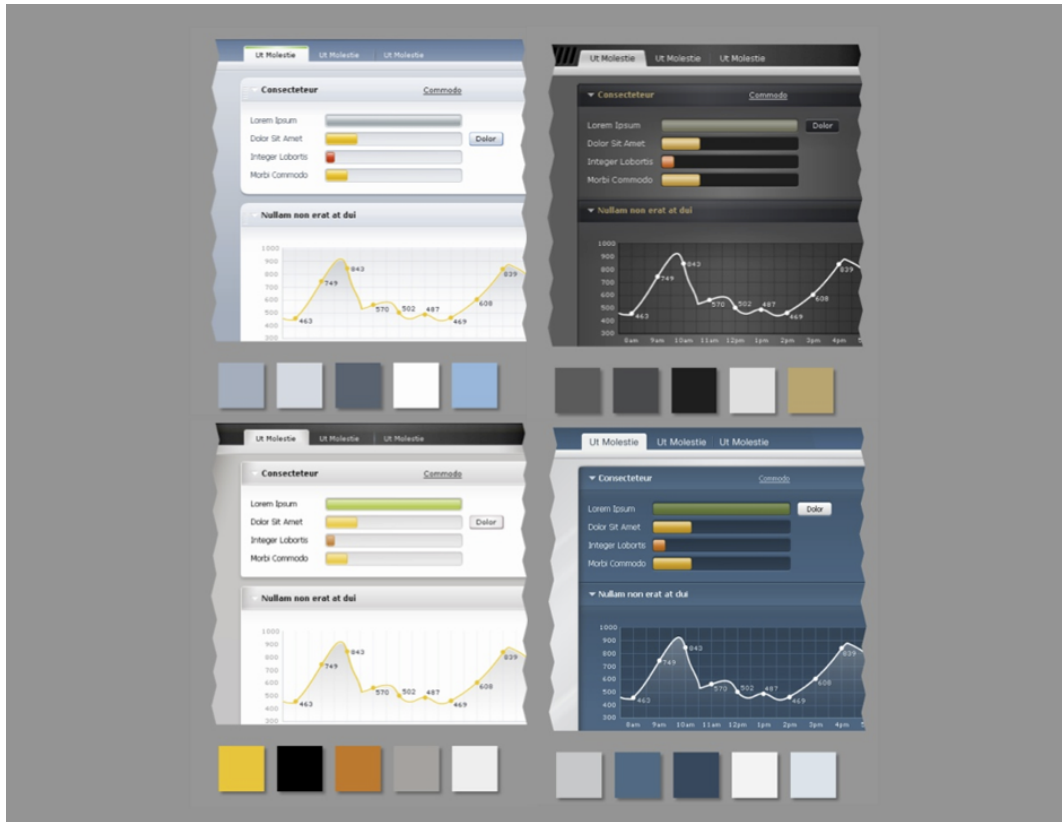
Figure 5-4: Visual language studies are used to explore a variety of visual styles abstractly and somewhat independent of the interaction design. This is useful because it allows us to have initial discussions about visual language without getting hung up on interaction design details. Of course, eventually visual design and interaction design must be conducted in lockstep.

We typically show between three and five different approaches during our initial review with stakeholders, most often using each one to optimize a particular experience attribute. This is a little different from our approach to interaction design, in which a product usually has one optimal behavioral framework. Visually, several different styles all can be consistent with experience keywords and goals. Using experience attributes to develop these approaches helps move stakeholders away from personal tastes and biases by providing a vocabulary for an experience that is in sync with the brand's meaning.

It is often useful to develop one or two extreme options that push the look and feel a bit too far in one direction. Doing this makes it easier to differentiate between the various approaches and helps stakeholders choose an appropriate direction. You will have ample opportunity later in the process to tame a particularly extreme visual style. That said, all the choices you present to your stakeholders should be reasonable and appropriate. It's almost an unwritten rule that if there's one direction you don't want your client or stakeholders to choose, that's the one they're guaranteed to like.

Once you've developed a good spectrum of visual language studies reflecting persona experience goals, brand guidelines, and experience keywords, it's time to present them to stakeholders for feedback. It's important to contextualize them in terms of these goals and keywords and to describe the rationale for each direction and its relative merits. We ask stakeholders to first give us their initial emotional reaction and then talk through things in a more rational fashion. By the end of this presentation, we usually have consensus to move forward with some aspects of several of the visual styles. It is common to iterate the visual language studies before moving on to the next step.

Step 3: Apply the chosen visual style to the screen archetype

The final step is to apply one or two selected visual styles to key screens. We typically coordinate our visual and interaction design efforts so that this step is performed close to the end of the interaction framework. At that point the design has begun to stabilize, and sufficient specific detail reflects the visual style. This further refines the visual style so that it reflects key behaviors and information. By making the design more concrete, you can better assess the feasibility of the proposed solution without the overhead of updating numerous screens for each minor change. Additionally, it's easier to elicit feedback from stakeholders.

Defining the industrial design framework

We develop the *industrial design framework* in much the same manner as the visual design framework. But because the form factor and input method have significant implications for both the industrial and interaction design, it's useful to collaborate early on to identify relevant issues.

The industrial design framework typically follows this process:

1. Collaborate with interaction designers about form factor and input methods.
2. Develop rough prototypes.
3. Develop form language studies.

Step 1: Collaborate with interaction designers about form factor and input methods

If the product you are designing relies on custom hardware (as with a cell phone or medical device), it is important for interaction designers and industrial designers to agree on a general physical form and input methods. While the course of the design framework will certainly help refine the design, decisions need to be made at this point. These decisions include the product's general size and shape; the screen size (if any); the number and general orientation of hard and soft buttons; and whether the product has a touch or multitouch screen, keyboard, voice recognition, motion/position tracking, and so on. This collaboration typically starts with a couple of days at the whiteboard and a condensed set of scenarios.

Important things to consider when making these decisions include persona experience goals (see Chapter 3), attitudes, aptitudes, and environmental factors, as well as brand and experience keywords, market research, manufacturing costs, and pricing targets. Because the cost of a hinge can make or break the margin on hardware, and because internal components (such as a battery) can have a tremendous impact on form, an early sanity check with mechanical and electrical engineers is critical.

There is only one user experience, and it comes from the combination of the physical form and the product's interactive behavior. The two must be designed in concert and, according to the adage of Modern architecture, form should follow function. The demands of interaction must guide the industrial design, but concerns about fabrication and cost will also impact the possibilities available to interaction design.

DESIGN PRINCIPLE	*There is only one user experience: Form and behavior must be designed in concert.*

Step 2: Develop rough prototypes

It is often the case that even after the overall form and input methods are defined, the industrial designer still can take a variety of approaches. For example, when we've designed office phones and medical devices, it's often been asked whether the screen angle should be fixed or if it should be adjustable and, if so, how that will be accomplished. Industrial designers sketch and create rough prototypes from foam board and other materials. In many cases, we'll show several to stakeholders because each one has different cost and ergonomic considerations.

Step 3: Develop form language studies

In a fashion similar to the visual language studies described earlier, the next step is to explore a variety of physical styles. Unlike the visual language studies, these are not abstract composites. Instead, they represent various looks applied to the specific form factors and input mechanisms determined in Steps 1 and 2. These studies include shape, dimensionality, materials, color, and finish.

As with visual style studies, form language studies should be informed by persona goals, attitudes, aptitudes, experience keywords, environmental factors, and manufacturing and pricing constraints. Typically these studies require several rounds of iteration to find a feasible and desirable solution.

Defining the service design framework

Because service design often affects organizations' business models, the *service design framework* may be conducted before other areas of design.

The service design framework typically follows this process:

1. Describe customer journeys.
2. Create a service blueprint.
3. Create experience prototypes.

The book *Service Design* by Polane, Løvlie, and Reason (Rosenfeld Media, 2013) contains a much more thorough treatment of this subject, with examples.

Step 1: Describe customer journeys

Similar to the context scenarios of interaction design, customer journeys describe an individual persona's use of a service as a descriptive narrative, from first exposure to final transaction. Different journeys stress different aspects of the service, accounting for different personas' goals. Each customer journey also provides an opportunity for the designer to take personas through secondary paths where the service helps them recover from a nuanced problem.

Step 2: Create a service blueprint

A service blueprint is the service's "big picture." It describes the collection of touch points by which the persona uses the service, such as a mobile site or a storefront. It also

describes the "backstage" processes by which service is delivered, such as the interface used by a customer service representative handling a phone call.

Early blueprints were flowcharts that described the connections between touch points. More recent trends draw these as swimlane diagrams that place the user at the top, the service organization at the bottom, and its channels—like marketing, sales, and customer service—across the page.

A horizontal "line of visibility" on the blueprint often distinguishes onstage and backstage touch points.

Some designers may prefer to begin with the service blueprint instead of the customer journeys. While each influences the other and is iterated across the project, the authors believe it is usually best (unless this is an update to an existing, mature service) to start with the customers via their design proxies—personas. Starting with the customer experience may help identify unexpected touch points in the service map that may otherwise be ignored.

Step 3: Create experience prototypes

Although the exhaustive design of a particular channel may most properly belong to interaction or visual designers, service designers illustrate a persona's individual experience (and the continuity between touch points) through experience prototypes. They almost certainly include mock-ups of key touch points like mobile apps and websites, but they can include much more. Often these are created as short video scenes that illustrate the experience cinematically.

These prototypes take many forms at many different degrees of fidelity, from simple interviews with potential customers focused on mock-ups to full-scale pilots of the prospective service.

Refining the Form and Behavior

When a solid, stable framework definition is reached, designers see the remaining pieces of the design begin to fall smoothly into place: Each iteration of the key path scenarios adds detail that strengthens the product's overall coherence and flow. At this stage, a transition is made into the *Refinement* phase, where the design is translated into a final, concrete form.

In this phase, principles and patterns remain important in giving the design a fine formal and behavioral finish. Parts II and III of this book provide useful principles for the

Refinement phase. It is also critical for the programming team to be intimately involved throughout the Refinement phase. Now that the design has a solid conceptual and behavioral basis, developer input is critical to creating a finished design that can and will be built, while remaining true to concept.

The Refinement phase is marked by the translation of the sketched storyboards into full-resolution screens that depict the user interface at the pixel level, as shown in Figure 5-5.
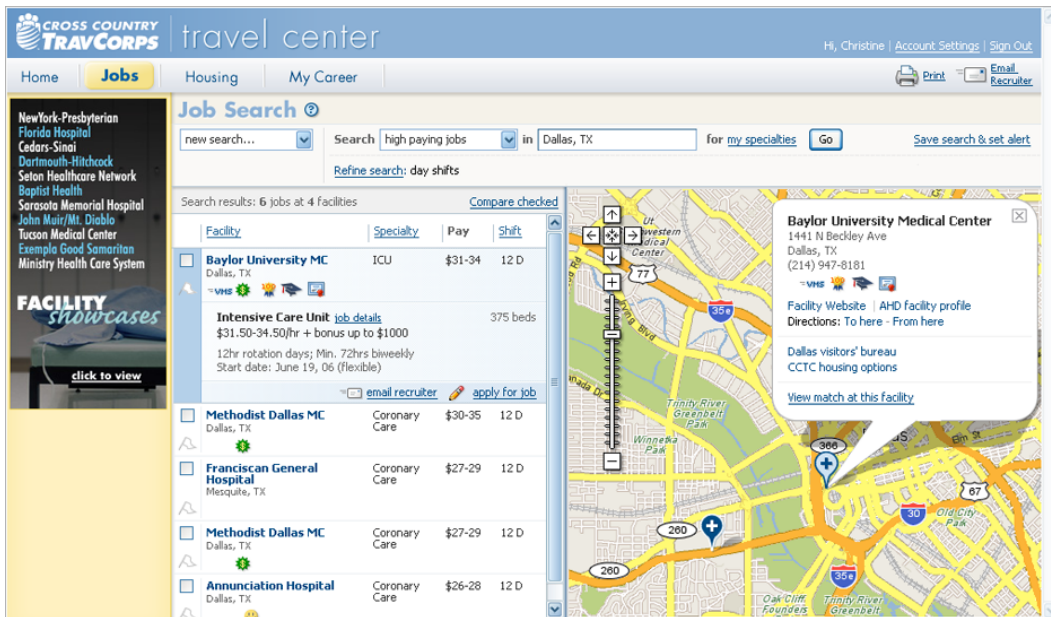
Figure 5-5: Full-resolution bitmap screens for Cross Country TravCorps based on the Framework illustration from Figure 5-3. Note that minor changes to the layout naturally result from the realities of pixels and screen resolution. Visual and interaction designers need to work together closely at this stage to ensure that visual changes to the design continue to reinforce appropriate product behaviors and meet the goals of the primary personas.

The basic process of design refinement follows the same steps we used to develop the design framework, this time at deeper and deeper levels of detail. (Of course, it isn't necessary to revisit the form factor and input methods unless an unexpected cost or manufacturing issue crops up with the hardware.) After following Steps 2 through 6 at the view and pane levels, while incorporating the increasingly refined visual and industrial designs, use scenarios to motivate and address the product's more granular components.

During this phase, you should address every primary view and dialog possible. Throughout the refinement phase, visual designers should develop and maintain a visual style guide. Developers use this guide to apply visual design elements consistently when they create low-priority parts of the interface that the designers typically don't have the time

and resources to complete themselves. At the same time, industrial designers work with mechanical engineers to finalize components and assembly.

While the end product of the design process can be any one of a variety of outputs, we often create a printable Form and Behavior Specification. This document includes screen renderings with callouts sufficiently detailed for a developer to code from, as well as detailed storyboards to illustrate behaviors over time. It can also be valuable to produce an interactive prototype in HTML or Flash that can augment your documentation to better illustrate complex interactions. However, keep in mind that prototypes alone are rarely sufficient to communicate underlying patterns, principles, and rationale, which are vital concepts to communicate to developers. Regardless of your choice of design deliverable, your team should continue to work closely with the construction team throughout implementation. Vigilance is required to ensure that the design vision is faithfully and accurately translated from the design document into a final product.

Validating and Testing the Design

In the course of an interaction design project, it's often desirable to evaluate how well you've hit the mark by going beyond your personas and validation scenarios to put your solutions in front of actual users. You should do this after the solution is detailed enough to give users something concrete to respond to, and with enough time allotted to make alterations to the design based on your findings.

In our experience, user feedback sessions and usability tests are good at identifying major problems with the interaction framework and at refining things like button labels and activity order and priority. They're also essential for fine-tuning such behaviors as how quickly a screen scrolls in response to turning a hardware knob. Unfortunately, it's difficult to craft a test that assesses anything beyond first-time ease of learning. There are a number of techniques for evaluating a product's usability for intermediate or expert users, but this can be quite time-consuming and is imprecise at best.

You have a variety of ways to validate your design with users. You can hold informal feedback sessions where you explain your ideas and drawings and see what the user thinks. Or you can give a more rigorous *usability test*, in which users are asked to complete a predetermined set of tasks. Each approach has advantages. The more informal style can be done spontaneously and requires less preparation. The downside to this approach is that the designer can unintentionally "lead the witness" by explaining things in a persuasive manner. In general, we've found this approach to be acceptable for a technical audience that can imagine how a few drawings might represent a product interface. It can be a useful alternative to usability testing when the design team doesn't have time to prepare for formal usability testing.

Given sufficient time, more formal usability testing has some advantages. Usability tests determine how well a design allows users to accomplish their tasks. If the test's scope is sufficiently broad, it can also tell you how well the design helps users reach their end goals.

To be clear, usability testing is, at its core, a means to *evaluate*, not *create*. It is not an alternative to interaction design, and it will never be the source of that great idea that makes a compelling product. Rather, it is a method to assess the effectiveness of ideas you've already had and to smooth over the rough edges.

Usability testing is also not the same as user research. For some practitioners, "tests" can include research activities such as interviews, task analyses, and even creative "participatory design" exercises. This conflates a variety of needs and steps in the design process into a single activity.

User research must occur *before* ideation; user feedback and usability testing must *follow* it. In fact, when project constraints force us to choose between ethnographic research and usability testing, we find that time spent on research gives us much more leverage to create a compelling product. Likewise, given limited days and dollars, we've found that spending time on design provides more value to the product design process than testing. It's much more important to spend time making considered design decisions based on a solid research foundation than to test a half-baked design created without the benefit of clear, compelling models of the target users and their goals and needs.

What to test

Because the findings of usability testing are often quantitative, usability research is especially useful in comparing specific design variants to choose the most effective solution. Customer feedback gathered from usability testing is most useful when you need to validate or refine particular interaction mechanisms or the form and expression of specific design elements.

Usability testing is especially effective at validating the following:

- **Naming**—Do section/button labels make sense? Do certain words resonate better than others?

- **Organization**—Is information grouped into meaningful categories? Are items located in the places customers might look for them?

- **First-time use and discoverability**—Are common items easy for new users to find? Are instructions clear? Are instructions necessary?

- **Effectiveness**—Can customers efficiently complete specific tasks? Are they making missteps? Where? How often?

It is also worth noting that usability testing, by its nature, focuses on assessing a product's first-time use. It is often quite difficult (and always laborious) to measure how effective a solution is on its 50th use—in other words, for the most common target: the perpetual intermediate user. This is quite a conundrum when you are optimizing a design for intermediate or expert users. One technique for accomplishing this is using a diary study, in which subjects keep diaries detailing their interactions with the product. Elizabeth Goodman, et al, provide a good explanation of this technique in *Observing the User Experience* (Morgan Kaufmann, 2012).

When performing usability testing, be sure that what you are testing can actually be measured, that the test is administered correctly, that the results will be useful in correcting design issues, and that the resources necessary to fix the problems observed in a usability study are available.

When to test: Summative and formative evaluations

In his 1993 book *Usability Engineering* (Morgan Kaufmann, 2012), Jakob Nielsen distinguishes between *summative evaluations*, which are tests of completed products, and *formative evaluations*, conducted during design as part of an iterative process. This is an important distinction.

Summative evaluations are used in product comparisons, to identify problems prior to a redesign, and to investigate the causes of product returns and requests for training and support. Summative studies generally are conducted and thoroughly documented by professional third-party evaluators. In some cases, particularly in competitive product comparisons, summative studies are designed to yield quantitative data that can be tested for statistical significance.

Unfortunately, summative evaluations are often used as part of the quality assurance process near the end of the development process. At this point, it's usually too late to make meaningful design changes. Design should be evaluated before the coding begins (or at least early enough that you have time to change the implementation as designs are adjusted). However, if you need to convince stakeholders or developers that the current product *does* have a usability problem, nothing beats watching real users struggle through basic tasks.

Formative evaluations do just this. These quick, qualitative tests are conducted during the design process, generally during the Refinement phase. When effectively devised and moderated, a formative evaluation opens a window to the user's mind, allowing the designers to see how (and, with interviews, why) their target audience responds to the information and tools they've provided to help them accomplish their tasks.

Although summative evaluations have their uses, they are product- and application-management activities conducted to inform product life cycle planning. They can be useful "disaster checks" during development, but the costs of changes at this point—in time, money, and morale—can be high.

Conducting formative usability tests

There are a wide variety of perspectives on how to conduct and interpret usability tests. Unfortunately, we've found that many of these approaches either presume to replace active design decision making or are overly quantitative, resulting in nonactionable data about things like "time to task." A good reference for usability testing methods that we've found to be compatible with Goal-Directed interaction design methods is Carolyn Snyder's *Paper Prototyping* (Morgan Kaufmann, 2003). It doesn't discuss every testing method or the relationship between testing and design, but it covers the fundamentals well and provides some relatively easy techniques for usability testing.

In brief, we've found the following to be essential components of successful formative usability tests:

- Test late enough in the process that there is a substantially concrete design to test, and early enough to allow adjustments in the design and implementation.
- Test tasks and aspects of the user experience appropriate to the product at hand.
- Recruit participants from the target population, using your personas as a guide.
- Ask participants to perform explicitly defined tasks while thinking aloud.
- Have participants interact directly with a low-tech prototype (except when testing specialized hardware where a paper prototype can't reflect nuanced interactions).
- Moderate the sessions to identify issues and explore their causes.
- Minimize bias by using a moderator who has not previously been involved in the project.
- Focus on participant behaviors and their rationale.
- Debrief observers after tests are conducted to identify the reasons behind observed issues.
- Involve designers throughout the study process.

Designer involvement in usability studies

Misunderstanding between an uninformed designer and the user is a common cause of usability problems. Personas help designers understand their users' goals, needs, and points of view, creating a foundation for effective communication. A usability study, by

opening another window on the user's mind, allows designers to see how their verbal, visual, and behavioral messages are received. They also learn what users intend when interacting with the designed affordances and constraints.

Designers (or, more broadly, design decision makers) are the primary consumers of usability study findings. Although few designers can moderate a session with sufficient neutrality, their involvement in the study planning, direct observation of study sessions, and participation in the analysis and problem-solving sessions are critical to a study's success. We've found it important to involve designers in the following ways:

- Planning the study to focus on important questions about the design
- Using personas and their attributes to define recruiting criteria
- Using scenarios to develop user tasks
- Observing the test sessions
- Collaboratively analyzing study findings

Notes
1. Schumann et al., 1996
2. Cooper, 1999
3. Shneiderman, 1998

CREATIVE TEAMWORK

In the Introduction to this book, we described the Goal-Directed method as consisting of three p's: principles, patterns, and processes. However, there's a fourth "p" worth mentioning— practices. This book mostly concerns itself with the first three, but in this chapter we'd like to share a few thoughts about the *practice* of Goal-Directed design and how design teams integrate into the larger product team.

In design and business, teams are common, but rarely are they successful or productive. The finer points of teamwork are not commonly taught or communicated. Teams require meeting time and discussion, and they add layers of communication. These are not inherently disadvantages, but when teamwork is not undertaken with care, its outcomes can be mere compromises. Team members can be too polite to dismiss others' ideas or too stubborn to let go of their own.

If you've ever worked on a highly productive team, you know that working together can produce outcomes that would be very hard, and sometimes impossible, for a single individual. In software and service development, we need teammates to ensure that the appropriate problems are addressed, ideas flow, solutions are effectively evaluated, and dead ends are quickly recognized. Well-functioning teams can actually make the product development process more efficient and its outcomes more effective for users.

This chapter discusses strategies for working together, complementary approaches to product development, and tactics for assembling teams across an organization. Some of the most interesting and important design problems are too big to solve alone—and too often, going it alone also can be rather lonely.

Small, Focused Teams

The following discussion addresses the team concept at two levels:

- **Core teams** are small and focused. Often they are arranged around a specific kind of expertise, such as engineering know-how, creativity, marketing savvy, or business leadership. But they could also be small, cross-functional teams in a startup or a small-scale product organization.

- **Extended teams** are large and sometimes geographically distributed. These can include stakeholders, whose work depends on the results of the project but who aren't responsible for the design itself. In almost any product development effort, the extended product team includes separate core teams for (at least) marketing, design, and engineering.

Even in large-scale product organizations, most of the actual work is done within the context of the core teams; therefore, this chapter covers techniques that can improve work within small teams. Whether that team is cross-functional or focused on a specific aspect of product development, the strategies in this chapter can help organize team-work around a set of simple practices. These practices can help ensure that ideas flow, work is efficient, and critiques are timely.

Thinking Better, Together

Small teams continually prioritize and address items on their work list. Some are minor; some merely seem minor; all must be prioritized and considered at their proper depth. Effective functional or cross-functional teams do more than "divide and conquer" when confronting the never-ending queue; they harness the sparky, sometimes chaotic, energies of their members in a collaborative effort that we'll call "thought partnership."

You can think of a thought partner as an intellectual complement—a collaborator who shares your goals but who comes at problems from a different angle, with a different set of skills. In *Thinking, Fast and Slow*, author Daniel Kahneman describes his partnership in his academic work with Amos Tversky in a way that looks to us like good thought partnership:

> *The pleasure we found in working together made us exceptionally patient; it is much easier to strive for perfection when you are never bored. Both Amos and I were critical and argumentative ... but during the years of the collaboration, neither of us ever rejected out of hand anything the other said.*[1]

How did thought partnership evolve? In the early days of Cooper's consulting practice, one of the meeting rooms came to be known as The Yelling Room. (The name was evidence that simply hiring like-minded people didn't always guarantee harmony around the whiteboard.) In those days, it was common for designers to campaign, loudly, for their design ideas. The spirit was collaborative and feisty, but the output was often unclear. "What did we decide, again?" "Whose idea won?" "What do we need to tackle now?"

Over time, a new collaboration strategy emerged. One designer was given responsibility for marshaling the conversation, and the other focused on ideating and exploring. In our practice today, we've applied a much deeper definition and distinction to each of the two approaches:

- **Generation**—In a generation-based approach, invention is undertaken boundlessly, and outcomes are most successful when people have space to generate freely, think laterally, and explore openly.
- **Synthesis**—In a synthesis-based approach, invention is most fruitful when it is guided and focused, and outcomes are assured only when they've addressed the user need.

Teams that blend these approaches make quick progress through complicated problems. Whatever role you play in the product development process, you can use the following practices to find a teammate who will help you do your best thinking.

Generators and synthesizers

In our hiring process at Cooper, we seek to identify individuals who will make powerful thought partners, and we specifically try to find designers with complementary skills and dispositions. We call these roles "Generator" and "Synthesizer." We've found that these two styles of creativity strike a powerful balance when we tackle complicated design problems. Generators and Synthesizers share responsibility for delighting users with simple solutions, and they have different responsibilities during the process. In the best cases, they allow their partners to fully inhabit the creative roles they're most comfortable playing.

Generation and synthesis live on the same spectrum of creativity, as shown in Figure 6-1. Most designers sit on one side of the spectrum, favoring either synthetic or generative skills. A strong Generator typically needs a partner of equal strength in synthesis to strike a healthy balance. When working alone, strong Generators may zoom in too quickly on an incomplete solution or spend too much time casting about in the muddiness of a brainstorm. They need a Synthesizer to help make decisions at the right level at the right times and to keep the design moving forward.
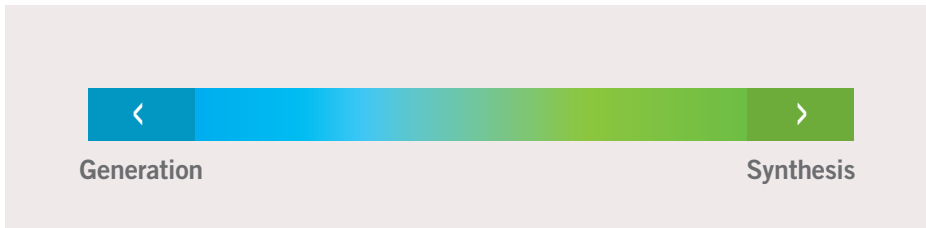
Figure 6-1: Generation and synthesis form a creative spectrum.

Synthesizers initiate dialog within their teams. Instead of proposing new ideas and solutions, Synthesizers ask questions to locate the intent and value within ideas that have already been proposed. As the conversations proceed, Synthesizers seek clarity, expose gaps, and draw connections. When working alone, strong Synthesizers can struggle to move beyond list-making and scenario thinking. They need a Generator to pull ideas into concrete, connected form.

The dialog between these two balances inspires ideation with evaluation and sense-making. It lays the foundation for a powerful partnership in confronting complex interaction problems. A generated idea can emerge as a vague or inspired idea, and capable synthesis can quickly reveal its inherent value or ensure that dead ends and darlings are quickly recognized and dispatched.

The following sections outline the qualities, characteristics, and interplay between the roles. Use the comparisons as a way to find your own creative disposition and to describe the kind of person who could bring out the best in you.

In a typical design meeting, the distinction between roles often manifests itself from the first moment, as shown in Figure 6-2. Who instinctively reaches for the marker and heads to the whiteboard when confronted with a design problem?

	Generator	Synthesizer
Most frequently found…	At the whiteboard, marker in hand	Formulating questions, scratching chin
When addressing a design problem…	Draws a picture	Makes lists
When considering a design solution…	Imagines new directions	Asks questions

Figure 6-2: Generators and synthesizers complement each other.

Each role tends to inhabit a specific physical and mental space during design meetings. Successful Generators tend to grab the marker to visualize ideas, as shown in Figure 6-3. Synthesizers tend to organize, listing the qualities of a good solution or refreshing their understanding of the problem, the user's goals, and the usage context.

	Generator	Synthesizer
Articulates the experience as…	A set of concrete ideas	A story with concrete plot points
Offers input in the form of…	Invention	Clarification
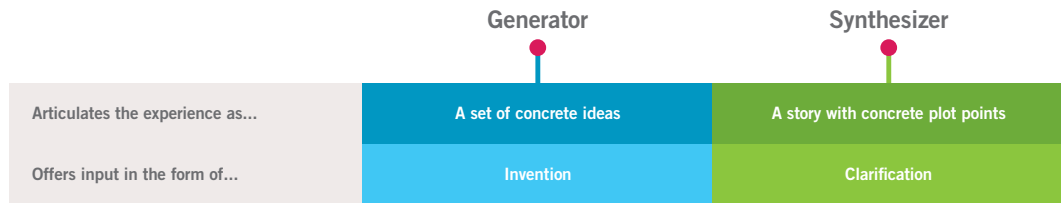
Figure 6-3: Generators and synthesizers approach design problems from different angles.

Generators tend to be relentlessly concrete thinkers, whereas the best Synthesizers lead with storytelling and prompting.

Here's a sample discussion from an imaginary project:

Generator: *I have an idea for adding something new to the list.* (Begins drawing the screen on the whiteboard.) *It's a gesture. When you're in the main list, you pull down, and you start to see the control for adding a new thing. It's just an empty field.*

Synthesizer: *Great. It might feel sort of hidden, though. In Jeff's scenario, he only adds a new thing every few weeks, right?*

Generator: *Yeah, true. He might forget it's there.*

Synthesizer: *I like the idea of keeping the focus on information rather than UI widgets, though.*

Generator: *Hmmm. How about we show the field with the help text "Add a new thing" when this main screen loads, and then the screen snaps up to cover it? So you see it, but then you focus on the list.*

Synthesizer: *I like it. It might get annoying if it happens a bunch of times in the same session, but I think we can come up with some rules.*

As a design meeting progresses, the Generator and Synthesizer work together to explore solutions and develop ideas. The Synthesizer should help shape the discussion, gently tightening the depth of field. The conversation usually begins with a wide-angle discussion of the problem and from there drills ever tighter into the details of a solution.

As shown in Figure 6-4, creative minds often disagree, especially when they're coming at the problem from different directions. Early on in a creative partnership, trust must be established. Generators must take the lead in concept direction. This means that Synthesizers must cede both the literal and figurative whiteboard marker. At the same time, Generators must trust Synthesizers to steer the discussion, to keep from getting mired in details, and to reframe the problem when necessary. This means that Generators must feel comfortable with introducing newly formed ideas and not feel threatened when their partner evaluates and critiques these ideas. Most ideas are bad, after all, and a primary goal in a thought partnership is to identify an idea's quality or promise early on and dispense with the cruft.

	Generator	Synthesizer
Takes responsibility for…	Driving the concept direction	Ensuring that concept is coherent and consistent
Can easily slip into…	Excessive detail	Pre-emptive analysis
When the creative balance is off…	Falls in love with a solution and defends it to the death	Blocks an idea or direction before it has a chance to develop
When the creative balance is perfect…	Freely experiments with ideas and explores varieties of solutions	Is open to ideas that might seem jarringly contrary to convention, probes for detail and nurtures their development

Figure 6-4: Generators and synthesizers have different responsibilities, strengths, and pitfalls.

In the *detailed design* phase, the team often spends the morning working together and then separates to document details, as shown in Figure 6-5. The Generator typically opens a drawing tool and begins capturing decisions in the form of drawings. The Synthesizer captures decisions, either in schematics that help explain the flow or in text that explains the rationale. The details that the team documents can be communicated in a variety of ways, depending on the team's size and needs. In our practice, we favor frequent, lightweight, informal notes over formal documentation milestones.

	Generator	Synthesizer
After the meeting…	Draws the interactive structure and flow	Captures the rationale for decisions in rough schematics and text
Focuses on understanding…	Flow, and the implications of design decisions	Development of a pattern language

Figure 6-5: Generators and synthesizers perform different tasks away from the whiteboard.

Getting started with thought partnership

If you want to establish thought partnership in your practice, you could begin by creating Generator or Synthesizer roles and then finding people to fill them. You could also start leaner, finding small ways to apply elements of the practice that are appropriate to your work. It can take a while to determine what kind of partnership you need, but you can start with simple steps.

Finding a partner who generates

- Before you begin, clarify the problem you want to solve.
- Appeal to the generative capabilities of a colleague or friend: "I need someone to help kick-start some ideas."
- If things don't work at first in the meeting, walk up to the board and draw a bad idea. If your partner is a good Generator, he or she will leap up and begin building on the bad idea or offer a counterproposal.

Finding a partner who synthesizes

- Before you begin, make sure you're designing for a higher-level "story" or scenario. This will help give your partner a rubric from which to work during the meeting.
- Appeal to your partner's evaluative capabilities: "Can you help me figure out whether this idea is any good?"
- If things don't immediately go well in the meeting, work on the story. Make sure you're both clear on what the user is doing and why.

Switching roles on the fly

Establish ground rules at the beginning of your partnership. Synthesizers should probe and guide; Generators should explore and ideate. You can swap roles within a meeting, but it's a good idea to make this switch explicit. When a Synthesizer comes up with a great idea, he must resist the urge to wrest the marker away from the Generator. Instead, he can simply offer to switch roles: "Mind if I generate for a bit?"

Getting unstuck (the 15-minute rule)

Small groups of creative people sometimes hit a wall. Ideas won't flow; conversations swirl or get mired in detail. In our practice, we encourage teams to bring in another designer if the discussion doesn't advance for 15 minutes. The core team should brief this outsider on simple contextual details: the user, the scenario, and the ideas under

consideration. The outsider should, in turn, probe the designers for rationale: Why is this good? How does this help? Almost invariably, this simple conversation helps lift the designers out of the details and move beyond the impasse in a matter of minutes.

Rightsizing core teams

If two are good, three must be better, four would be amazing, and ten could bend space and time. Right? Organizations large and small fall victim to an additive urge when building teams. In our experience, teams are most effective when the members have clear roles and when the teams are small and nimble. This is why we have applied the moniker "core team" to the unit responsible for getting work done.

In *Scaling Up Excellence*, Stanford business professors Robert Sutton and Huggy Rao cite numerous examples in which bigger teams actually produce worse outcomes; they call this phenomenon "The Problem of More":

> *Employees in bigger teams gave others less support and help because it was harder to maintain so many social relationships and to coordinate with more people ... The key challenges are how to add rules, tools and people without creating [bloat].[2]*

In our practice at Cooper, we steer clear of bloat by insisting on four things: small teams, clear roles, tight decision-making loops, and minimal "work about work." That last phrase describes any activities that are not directly tied to making progress on the core objectives of product development: coming up with great ideas and shipping great products. Status e-mails and quick, noncritical check-ins are examples of "small" tasks that can accumulate and end up requiring significant effort and coordination. If you're reading this book while you're in a nonessential meeting, you're sinking in the quicksand of "work about work."

By localizing decision-making and being proactive about planning milestones and check-ins, we carve out space for a small set of creative teammates to flourish. In specific numeric terms, our rule of thumb for the rightsized core team to focus on an individual problem is no more than four, and no fewer than two. A minimum of two enables rapid evaluation and iteration. Teams with more than four members have too much overhead, too many people to please, and too much opportunity to lose momentum.

Finally, remember that a small team can be nimble only when roles are clear, responsibility for success is shared, and abilities are complementary. The following sections discuss the various participants in teams large and small and provide some tips for getting the most out of them.

Working Across Design Disciplines

The Synthesis-Generation model can be applied to any professionals engaged in creative discussion. In our practice, it provides structure when any two individual designers are collaborating—visual designer and interaction designer, two interaction designers, designer and developer, interaction designer and creative lead. While the practice of user experience design has become more established, practitioners often find themselves in new collaborative situations with creative professionals from many backgrounds. This model can be usefully applied in many core team situations, with participants from various disciplines.

Over a product's lifetime, designers across disciplines must do more than merely communicate. Interaction designers must coordinate and cooperate with visual and industrial designers, staging decision-making so that each discipline is equipped with the material it needs to work most effectively. The following sections offer a framework for understanding the distinct responsibilities of each discipline, along with a summary of how the disciplines can work together effectively.

Interaction design

Interaction designers are responsible for understanding and specifying how the product should behave. This work overlaps with the work of both visual and industrial designers in a couple of important ways. When designing physical products, interaction designers must work with industrial designers early on to specify the requirements for physical inputs and to understand the behavioral impacts of the mechanisms behind them. Interaction designers cross paths with visual designers throughout the project. In our practice, their collaboration begins early, as visual designers guide discussions of the brand and emotive aspects of the experience in discussions with users and the extended team.

Visual interface design

In our practice, we've come to recognize that *visual interface design* is a critical and distinct discipline that must be conducted in concert with interaction design and—when appropriate—industrial design. It has great power to influence a product's effectiveness and appeal. But for this potential to be fully realized, visual design must not be an afterthought—it isn't a "coat of paint." It should be regarded as one of the essential tools for satisfying user and business needs.

Visual interface designers' work emphasizes the organizational aspects of the design and how visual cues and affordances communicate behavior to users. These designers must work closely with interaction designers to understand the priority of information, flow, and functionality in the interface and to identify and strike the right emotive tones.

Visual interface designers focus on how to match the interface's visual structure to the logical structure of both the users' mental models and the application's behaviors. They are also concerned with communicating application states to users (such as read-only versus editable) and with cognitive issues surrounding user perception of functions (layout, visual hierarchy, figure-ground issues, and so on).

Visual interface designers must have command of basic visual properties—color, typography, form, and composition—and must know how these can be used to effectively convey affordances, information hierarchy, and mood. Visual interface designers should be aware of brand psychology, know the history of graphic design, and be familiar with current trends. They must know accessibility principles and the science of human perception. Visual interface designers also need a fundamental understanding of interface conventions, standards, and common idioms. (Read more about Goal-Directed visual interface design in Chapter 17.)

Graphic design

Until the last 20 years or so, the discipline of graphic design was dominated by the medium of printed ink, as applied to packaging, advertising, environmental graphics, and document design. These traditional practices were not developed to address the demands of pixel-based output. However, the discipline has evolved considerably in the last two decades, and graphic design has become rigorous and eloquent in its approach to digital, screen-based media.

Talented, digitally fluent graphic designers excel at providing rich, aesthetically pleasing, exciting interfaces. They can create beautiful and appropriate surfaces for the interface that establish a mood or connection to a corporate brand. For them, design is often first about the tone, style, and framework that communicate a brand experience, secondarily about legibility and readability of information, and finally about communicating behavior through affordances (see Chapter 13).

Visual information design

Visual information design is concerned with the visualization of data, content, and navigation, rather than interactive functions. It is distinguished from information design in that it is less concerned with the content's editorial and information architecture issues, focusing instead on the graphical presentation. This skill set is particularly important in designing data-intensive applications, where users spend much of their time with complex content.

The primary goal of visual information design is to present data in a way that promotes understanding. This is accomplished by controlling the information hierarchy through

the use of visual properties such as typography, color, shape, position, and scale, as well as how these attributes change over time. It can also include microinteractions with information displays that expose details or connections to additional information. Common applications of visual information design include charts, graphs, sparklines, and other means of displaying quantitative information. Edward Tufte has written several seminal books that cover this topic in detail, including *The Visual Display of Quantitative Information* (Graphic Press, 1993).

Industrial design

When designing a convergent product, industrial designers must define the form of the physical product, embodying the brand with shape and material. For the purposes of interaction design, they specify physical input mechanisms. Interaction designers can perform research into the users' overall needs and the device's goals as a whole. Industrial designers lay the groundwork for their concepts with competitive analysis and materials research. The input mechanisms should not be specified before the desired interaction paradigms are known. Interaction iterations of the software's primary elements should be conducted in parallel with industrial design concepts so that the outcomes of each can inform subsequent phases. Throughout, interaction designers benefit from the materials expertise and ergonomic acumen of industrial designers, and the industrial designers benefit from the holistic experience vision created by the interaction designers.

Much like the difference in skills between graphic designers and visual interface and information designers, a similar split occurs among the ranks of industrial designers. Some are more adept at creating arresting and appropriate forms for objects, and others emphasize logical and ergonomic mapping of physical controls in a manner that matches user goals and communicates device behaviors. The increase in software-enabled devices that use rich visual displays demands a concerted effort on the part of interaction designers, visual interface designers, and industrial designers to produce complete and effective solutions.

The interplay of these roles involves much more subtlety, and many great resources explore this topic in detail. Kim Goodwin's book, *Designing for the Digital Age* (Wiley, 2011), offers practical techniques and tips for defining roles and establishing thought partnership in the context of a design project.

The Extended Team

The preceding discussion covered practices that allow great design to happen among a small core team. But for design to transform into a shipping product, the design must

enter the minds and hearts of many more people. Will the rest of the extended team understand its value? Will they indulge it long enough to give it the kind of critique that makes it better? Even if it survives early whiteboard sessions, will it survive a presentation to the product owner? If the product owner gets behind it, will it be understood, implemented, and extended effectively by the production engineers?

The following discussion is intended to outline a few simple strategies for integrating design in large-scale product-focused teams. It's not intended to be a comprehensive accounting of the best practices in product or service design and development. Very few ideas are truly great. This is why you need timely, candid feedback from capable collaborators, both among your core team and across the extended product team. This section discusses the frequent major participants within product teams. Emphasis is placed on how and when to collaborate with them so that your ideas get the kinds of critiques they need, at the appropriate time.

Areas of responsibility and authority

Designers are never the sole participants in creating a great interactive experience. Within digital product organizations, the expertise of engineers, marketers, and business stakeholders must be interwoven with the work of design in a product's creation and evolution. The following list describes a division of responsibilities, balanced by an equal division of authority among those different disciplines:

- **Design** is responsible for users' goals for the product. Many organizations currently do not hold any specific person, or team, responsible for goals. To carry out this responsibility, designers must have the authority to decide how the product will look, feel, and behave. They also need access to information. They must observe and speak to potential users about their needs, to engineers about technological opportunities and constraints, to marketing about opportunities and requirements, and to management about the kind of product to which the organization will commit and the results they expect.

- **Usability** is responsible for validating that users respond to the design as intended and that the overall experience and detailed interactions have the desired effect—being useful, usable, and desirable. To be effective, usability should be independent from but collaborative with the design group, and both usability and design should report to a decision-maker who can weigh the results from an informed and objective standpoint, and has the authority to have any necessary corrections to the design or implementation enforced. Usability's strength is in identifying problems, while design's strength is in identifying solutions. Collaboration works best when this division of labor is maintained.

- **Engineering** is responsible for construction. This means that they must have authority over the raw materials and construction processes—development platforms

and libraries, for example—along with the relative difficulty ratings and cost of the items in their backlog. The engineering and design teams must stay in touch when those items are being prioritized. Designers must review and react to the implementation of form and behaviors, and both teams must evaluate the success or failure of the overall experience as it is developed and tested. Designers should rely on engineers to provide guidance on technical constraints and opportunities, as well as the feasibility of proposed design solutions.

- **Marketing** is responsible for defining the market opportunity as a set of customer needs, preferences, and motivations. This team also eventually must convince customers to purchase the product. To do this, the marketing team must have the authority to advocate for the most lucrative, or beneficial, customer segments. Its members must provide guidance in targeting design research with the appropriate users, and the team must have access to the outcomes of that design research. (It's worth noting that, as we discuss in Chapter 3, customers and users are often different people with different needs.)

- **Business leads** are responsible for defining the business opportunity, because they're also accountable for the product's profitability. This group must drive decision-making around where opportunities for differentiation exist, and what features and functions must be prioritized across the extended team. To make those decisions, business leads need to receive clear information from the other groups: design's research and product definition, marketing's research and sales projections, and engineering's estimations of the time and cost to create the product.

Collaboration among these teams best happens in two venues: at frequent, informal working meetings where new ideas are explored, evaluated, and elaborated on; and at checkpoints that correspond to the end of each phase in established processes. Working meetings are particularly important for engineers once the design has begun to gel, and they are critical for marketing in the project's early and late stages.

As the product vision evolves, each team must continually seek to address its core concern:

- **Designers**—What are the simplest, most coherent, most delight-inspiring mechanisms for crafting the experience?

- **Usability professionals**—Does the design deliver on its promise of usefulness, usability, and desirability? Do users really use the product as assumed in the design?

- **Engineers**—How do we deliver the experience in a way that is fast, robust, scalable, and extendable?

- **Marketers**—How can we inspire adoption?

- **Business leaders**—Where is the most evident overlap of product function and market need?

When team members focus on those questions and ensure that they're prioritized appropriately, extended team interactions are clear and direct.

Collaborating with agile developers

Designers imagine and specify the right experience: a form that delights users, an experience that feels right, behaviors that fit use. Developers offer a corollary: The right experience should be built *in the right way*. We once believed that all design work should be completed before coding begins, but we've come to learn that this is neither desirable nor practical. Clear benefits arise from methodically testing and proving the feasibility of design hypotheses along the way.

Agile development methods arose as a response to the very real drawbacks of "waterfall" methods, which are characterized by years-long development timelines, hundred-page requirements documents, processes with various "gates" to ensure "quality," and dozens of participants. Agile methods seek to optimize time and energy, reduce waste, and ensure that concepts actually deliver value to users. They encourage many of the same principles discussed early in this chapter—small teams, focused work, and frequent discussion. Yet although they have evolved the practice of software development, agile methods also complicate—and, in some cases, short-circuit—design work.

This section discusses ways to ensure that the work of designers informs and shapes the development of products developed with agile methods. Let's first understand a couple of fundamental perspectives on agile methods:

- Business stakeholders tend to love the idea of agile development because it sounds economical and efficient. Yet business stakeholders often don't realize until much later that building fast means thinking fast, and fast thinking requires a solid foundation of assumptions and expected outcomes. If no underlying foundation or shared vision exists for what should be built and tested, agile work is aimless and, contrary to promises, wastes time.

- Developers tend to love agile methods because such methods support more of what they love (coding) and less of what they don't (sitting in meetings, interpreting requirements documents). When misapplied, however, this way of working can result in heads-down sprinting toward hazily specified destinations, resulting in just as many frustrating dead ends as classic waterfall methods.

In our experience, agile development is highly successful when core product elements are clearly rendered, widely understood, and implemented so that they can be tested well. Thus, we have found that *the primary elements of an experience should be planned, visualized, and discussed before they are built.* Even in a highly iterative process, informed planning should precede construction. You must think before you do. This is a much

messier reality than the neat sequential compartmentalization of design and construction, but it can initiate a productive dialog between designers, developers, and business decision-makers.

The remainder of this section covers a couple of simple questions at a high level:

- What does design mean in an agile context?
- How should designers adapt their practice in fast-paced, agile ways?

The work of designers on an agile team

Interaction designers seek simplicity and cohesion, knowing that cohesive solutions won't appear fully formed on day one. They emerge over time, hewing to the spirit of Antoine de Saint-Exupéry's oft-cited design aphorism: "...Perfection is attained not when there is no longer anything to add, but when there is no longer anything to take away."[5] Waterfall methods are friendlier to this way of working, because they provide time for ideas to develop and for their flaws to be revealed and removed. Agile methods value speed and focus on small iterations, but they reward designers with a chance to understand how users interpret design in progress.

In agile contexts, designers must prioritize and visualize; they can bring clarity and concreteness to desired outcomes and facilitate conversations around what elements must be developed to achieve them. This sounds similar to the work of design in many other contexts, but there are two important differences:

- There can be tension, or outright disagreement, in how the experience is defined when people work in agile teams. Designers must be able to define the critical elements of the user experience while working in concert with developers who may reveal constraints on or obstacles to that definition.
- Designers must think differently about the inputs to, and outcomes of, their practice. Agile development allows for feedback from users early on and often. This is an opportunity, and designers should appreciate it as such.

Designers rarely get a chance to fully visualize and perfect a user experience vision in an agile environment. But they must advocate for goal-directedness in the definition of the product and in the prioritizing of items in development.

Defining the user experience on agile teams

At the beginning of a collaborative effort with agile developers, designers will want to quickly assess *how much of the experience has already been defined, specified, or implied.*

This definition could be explicit—in the form of a wireframe created by a lead architect or business stakeholder—or implicit, in the form of shared assumptions about how it will work, the implementation technologies used, and so on. Designers should expect to shape the primary elements of the user experience—layout and navigational metaphors, the information architecture, transitions and animations that establish a sense of polish. Therefore, it's important to reckon with predetermined implications and constraints.

On the best agile teams, the designers specify the foundation of the experience while the developers lay the non-user-facing technical foundation. In less-optimal cases, designers need to move quickly to confront assumed specifications, especially those that prematurely determine an important aspect of the experience. These conversations can be difficult, but the whole point of user experience design is to express the product hypothesis through the user interface.

In our practice, agile developers can be capable thought partners at this stage. Talented developers think deeply about the infrastructure and connective tissue of interactive products, and they bring a healthy perspective on where the true engineering complications lie. In the best cases, they provide useful direction toward the right level to apply design and ensure that the work of design is useful and readily implemented. In the worst cases, neither developers nor designers understand the value of the others' expertise.

Doing valuable work on an agile team is much like doing it in other contexts. It always comes down to articulating users' goals, context, and flows; visualizing and iterating the desired experience; and prompting, gathering, and interpreting frequent user feedback. On agile teams, designers must be able to quickly determine where the biggest user experience challenges lie and to ensure that related elements flows are defined before development begins.

For case studies in the collaboration of user experience designers and agile developers, read the Smashing Magazine article *Lean UX: Getting Out of the Deliverables Business* by Jeff Gothelf and Josh Seiden.[3] It's a fine primer on the ways in which user experience design can be applied in specific agile development contexts.

Incorporating user feedback

For designers, the most valuable by-product of agile processes is feedback on the user experience—and lots of it. This feedback can be highly useful, as long as it is prompted, gathered, and interpreted appropriately. When you're a designer, it's critical to determine what you want to learn about the user experience during each release of new features or functions.

Feedback in an agile context is fast-paced but familiar. Early on, the prompts should be directed toward understanding whether the fundamental user hypothesis is correct. Do

the expected users derive the expected value? How well do the primary elements serve them? It's also important to look for what is working, independent of what you hoped would work. What are people using? Why?

As development continues, feedback can be more focused: Do the primary elements flow well? How are they accessed? What isn't used? Where do users stumble? What surprises users?

Establishing a Creative Culture

Establishing teams is essential, but a team full of capable people doesn't guarantee good output. Great teams emerge and develop because they're placed in environments—physical and virtual—that nourish them. In our experience, the team's cultural and social dynamics are as important as clarity around roles. Does the team enjoy working together? Do they have fun together? Do they have compatible schedules? Interests? Ways of working? Senses of humor?

Renowned record producer Steve Albini once laid out his expectations in a letter to a band before they entered the recording studio:

> *I have worked on hundreds of records, and I have seen a direct correlation between the quality of the end result and the mood of the band throughout the process. If the record takes a long time, and everyone gets bummed and scrutinizes every step ... the end result is seldom flattering.*[4]

A famous recording studio doesn't guarantee a good record, nor does a great producer. These ingredients are important, but those investments are wasted if the session is unproductive. Process can provide structure, but the ideas themselves need light and life to emerge and develop. Creating a positive, productive organizational culture is a topic that is too big to fully explore in this space, but it's too important to omit. Here is some of the raw material of an organizational culture; consider how you can use these elements to promote the cultural oxygen required to fuel the spark of creativity:

- **Environmental serendipity**—Creative organizations can go overboard with environmental design, but there's a method in what may seem like madness: Casual interactions with people, artifacts, or architectural moments set the tone for creativity. Small environmental surprises—a bold color, a new texture, an interesting surface—offer external encouragement for sparky ideation.

- **Small workspaces**—An ideal workroom announces itself as a place that is friendly to creativity, providing surfaces and tools for sketching, along with room to move around. Nothing kills momentum more often than distraction, so a workroom should block out

as much of the outside world as possible. A door is the typical architectural method of achieving this goal, but doors can be rare in open floor plan workplaces. Find a corner, erect barriers, and keep out the rest of the world as you dive deep into a problem.

- **A collaborative code of conduct**—The team should agree on how they'll work together. Clarity and trust in one another's roles, responsibilities, and work habits are essential. Small rituals are the foundation of good teamwork. Meeting start times, durations, breaks, and agendas can seem inconsequential, but small problems can accumulate. Focus must be established together, so turn off your phone and close your laptop. Like a growing pile of dirty dishes in the kitchen sink, minor inattentions can easily overwhelm good intentions. Do your dishes, hold your teammates accountable, and get to work.

- **Space for positive digression**—Digressions can seem egregious until they deliver you to unexpected insight. When a team is pressed for time, a digression can feel wasteful and sap energy. Teams who are open to digression tend to have more wide-ranging discussions and arrive at more interesting and more carefully considered solutions. The moral: Leave some time to stray.

Paying attention to the social dynamic is critical throughout the lifetime of the partnership. When there's conflict within the team, progress slows, quality decreases, and the end result flatters no one. For a design team, a solution of poor quality should be the ultimate bummer.

Identifying Skill Levels in Designers

Designers' skills must meet the scale or depth of a design problem. Craftspeople get bored with simple problems; apprentices may end up with subtle, sophisticated problems that their skills can't match. In both cases, the product suffers, and the reputation of the design organization can take a hit. Design leaders must ensure that problems are rightsized to the abilities of the designers in their practices. This requires a keen sense of a problem's location, size, and impact before the design work even begins.

Here's a quick guide to how we think about the levels of experience and the skills required at each level:

- **Apprentices** are early career designers who must be matched with mentors to guide their skill development. It can take considerable time and effort to develop the design judgment that allows apprentices to reliably provide solid, sound ideas that truly fit the task of the design problem. In the process of developing these skills, apprentices must be allowed to stretch to solve problems beyond their skill levels, but they need strong support and guidance from senior teammates while doing so.

- **Craftsmen**—With time, designers achieve more and more independence as they master their craft. This allows them to take a leadership role within a core team and drive a creative vision every day. Many designers spend their entire career at this level, especially if they're less inclined to take on the responsibilities of organizational leadership.

- **Leaders** possess high levels of design craft, along with the willingness and aptitude to provide organizational leadership. In large-scale product companies, a design leader is essential in providing guidance and structure for the design team, advocating for budget and authority, scoping and prioritizing projects, promoting the cause of design, and hiring designers.

Progressing up the ladder of design skill comes down to developing design judgment. How quickly can a designer recognize the fitness of a solution to its task? And how reliably can she guide a team toward a better solution or a deeper understanding of the problem? Leaders must supplement hard skills in design judgment with mentorship skills and organizational savvy. Not every designer aspires to develop these capabilities, and "leadership" is by no means the only, or most optimal, endpoint for highly talented craftsmen.

Collaboration Is the Key

There is no recipe for creative vision or good design judgment. Even when you believe you have the right concept, it takes considerable hard work, diligence, and skill to execute it well. One of the most challenging and chaotic but ultimately rewarding aspects of this hard work is collaborating with the rest of the product and business team. The vertigo caused by these struggles and challenges has motivated us to take a methodical approach.

We have found that designers should be able to collaborate with all the other people on the project team whose work impacts the overall user experience. Depending on the project, this may include design strategists, user and market researchers, user documentation writers, packaging designers, and possibly even store and point-of-sale designers. The point of this collaboration is to ensure that all aspects of the user experience are in harmony. They shouldn't work at cross purposes or use different design languages that could ultimately confuse the user or muddy the product's message.

Ultimately, the successful delivery of a product that meets people's needs requires the careful coordination of the efforts of a large number of people. We've found that to be effective, designers must ultimately assume considerable responsibility for orchestrating a fine balance between the numerous forces pushing and pulling on a product. We

hope that the tools described in this chapter will help you create great digital products that truly satisfy your users and customers.

Notes

1. Kahneman, 2011, p 5
2. Sutton and Rao, 2014
3. http://www.smashingmagazine.com/2011/03/07/lean-ux-getting-out-of-the-deliverables-business/
4. http://dontpaniconline.com/magazine/music/steve-albinis-letter-to-nirvana-before-he-produced-in-utero
5. Saint-Exupéry, 2002

Designing Behavior and Form

A BASIS FOR GOOD PRODUCT BEHAVIOR

In Part I, we discussed how to appropriately sequence the decisions to define and design a desirable and effective product. But how do we make these decisions? What makes a design solution good? As we've already discussed, a product's ability to meet the goals and needs of users while also accommodating business goals and technical constraints is one measure of design quality. But does a product solution have recognizable attributes that enable it to accomplish this successfully? Can we generalize common solutions to apply to similar problems? Must a design possess universally applicable features to make it a "good" design?

The answers to these questions lie in the use of interaction design values, principles, and patterns. *Design values* are guidelines for the successful and appropriate practice of design. *Design principles* are guidelines for the design of useful and desirable products, systems, and services. *Design patterns* are exemplary, generalizable solutions to specific classes of design problems.

Design Values

Design values describe imperatives for the effective and ethical practice of design. They inform and motivate the principles and patterns that are discussed later in this chapter.

Values are rules that govern action, and typically they are based at their core on a set of beliefs. The following set of design values was developed by Robert Reimann, Hugh Dubberly, Kim Goodwin, David Fore, and Jonathan Korman to apply to interaction design in particular, but they also readily apply to any design discipline that aims to serve people's needs.

Designers should create design solutions that are:

- **Ethical** (*considerate, helpful*)
  - Do no harm
  - Improve human situations
- **Purposeful** (*useful, usable*)
  - Help users achieve their goals and aspirations
  - Accommodate user contexts and capacities
- **Pragmatic** (*viable, feasible*)
  - Help commissioning organizations achieve their goals
  - Accommodate business and technical requirements
- **Elegant** (*efficient, artful, affective*)
  - Represent the simplest complete solution
  - Possess internal (self-revealing, understandable) coherence
  - Appropriately accommodate and stimulate cognition and emotion

Let's explore each of these values in a bit more detail.

Ethical interaction design

Interaction designers face ethical questions when they are asked to design a system that has fundamental effects on people's lives. These may be direct effects on users of a product, or second-order effects on other people whose lives the product touches in some way. This can become a particular issue for interaction designers because, unlike graphic designers, the product of their design work is not simply the persuasive communication of a policy or the marketing of a product. It is, in fact, the means of executing policy or the creation of a product itself. In a nutshell, interactive products *do things*, and as designers, we must be sure that the results of our labor do *good* things. It is relatively straightforward to design a product that does well by its users, but the effect that product has on others is sometimes more difficult to calculate.

Do no harm

Products shouldn't harm anyone. Or, given the complexities of life in the real world, they should, at the very least, *minimize harm*. Interactive systems could be a party to several possible types of harm:

- **Interpersonal** harm (loss of dignity, insult, humiliation)
- **Psychological** harm (confusion, discomfort, frustration, coercion, boredom)
- **Physical** harm (pain, injury, deprivation, death, compromised safety)
- **Economic** harm (loss of profits, loss of productivity, loss of wealth or savings)
- **Social and societal** harm (exploitation, creation, or perpetuation of injustice)
- **Environmental** harm (pollution, elimination of biodiversity)

Avoiding the first two types of harm requires a deep understanding of the user audience, as well as buy-in from stakeholders that these issues are within a scope that can be addressed by the project. Many of the concepts discussed in Parts II and III can help designers craft solutions that support human intelligence and emotions. Avoiding physical harm requires a solid understanding of ergonomic principles and appropriate use of interface elements. Physical harm can result from something as simple as repetitive stress injury due to excessive mouse usage. Far more serious issues of poor design can result in death, as is possible with overly complex and distracting in-car navigation systems.

Outside the realm of consumer products, it is easier to imagine examples that are relevant to the last three types of harm, such as a stock-trading application, an electronic voting system, or the control system for an offshore oil-drilling platform or nuclear power plant.

Design of military or gambling applications, or other applications that are in some sense *deliberately* harmful—or applications that, by virtue of their success at increasing workforce efficiency, allow employers to reduce their workforce—pose a different challenge, one for the designer's *conscience*. Such ethical gray areas have no easy answers.

Although on the surface environmental and societal harm may not seem like issues related to most consumer products, a deeper consideration exposes important issues of *sustainability*. Designers should be mindful of the full life cycle of the products they design, even after they are disposed of. They should also be aware of how the behaviors of people using products may affect the broader environment. For example, the iPhone and its associated ecosystem have resulted in dramatic increases in data usage on cellular and other networks. This in turn has resulted in the building of more cellular towers, vast expansions of server farms, and a dramatic increase in the demand for power.

Environmental impacts may be second- or third-order effects of consumer products, and may be difficult to predict, but the ultimate consequences may nonetheless be very significant.

In her book, *User Experience in the Age of Sustainability* (Morgan Kaufmann, 2012), Kem-Laurin Kramer identifies the following key phases of a product's life cycle that have a bearing on sustainability:

- **Manufacturing**, wherein the source, type, extraction, and refinement processes, and use of materials, begin to define the product's environmental footprint

- **Transportation**, wherein the modes of transport used to get products to market, and their associated power sources, are added to the environmental footprint

- **Usage and energy consumption**, wherein energy used to produce and power products as well as maintain services is added to the environmental footprint

- **Recyclability**, wherein reuse of materials, ease of repair/servicing, upgrade paths, and availability of replacement parts are factored into the environmental footprint

- **Facilities**, wherein the environmental requirements of manufacturing, R&D, sales, warehousing, server farms, and other physical support locations round out the remainder of the product's environmental footprint

These five phases may seem like a lot to consider when thinking about designing a new product, much less a digital product. Only a few of them really apply to software products, such as energy consumption and facilities for a typical web service. But we can look to companies like Apple, which are taking such factors into account. Many of Apple's recent products have been specifically designed to minimize materials, maximize recyclability, and reduce power consumption. Whether Apple has been able to match this forward-thinking sustainable hardware approach on the software side (think of the giant, power-hungry server farms that support the iCloud and iTunes services, for example) is perhaps an open question. Google, Facebook, and the entire web-service industry need to address this issue.

Improve human situations

Not doing harm is, of course, insufficient for a truly ethical design; improving things should be a goal as well. Here are some types of situations that interactive systems might improve:

- **Increasing understanding** (individual, social, cultural)
- **Increasing the efficiency and effectiveness** of individuals and groups
- **Improving communication** between individuals and groups

- **Reducing sociocultural tensions** between individuals and groups
- **Improving equity** (financial, social, legal)
- **Balancing cultural diversity with social cohesion**

Designers should always keep such broad issues at the back of their minds as they engage in new design projects. Opportunities to do good should always be considered, even if they are slightly outside the box.

Purposeful interaction design

The primary theme of this book is *purposeful* design based on an understanding of user goals and motivations. If nothing else, the Goal-Directed process described in Part I should help you achieve purposeful design. Part of purposefulness, however, is not only understanding users' goals but also understanding their limitations. User research and personas serve well in this regard. The behavior patterns you observe and communicate should describe your users' strengths *as well as* their weaknesses and blind spots. Goal-Directed Design helps designers create products that support users where they are weak and empower them where they are strong.

Pragmatic interaction design

Design specifications that gather dust on a shelf are of no use to anyone: A design must get built to be of value. After it is built, it needs to be deployed in the world. And after it is deployed, it needs to provide benefits to its owners. It is critical that business goals and technical issues and requirements be taken into account in the course of design. This doesn't imply that designers necessarily need to take at face value everything their stakeholders and developers tell them. An active dialog must occur among the business, engineering, and design groups about where firm boundaries exist and what areas of the product definition are flexible. Developers often state that a proposed design is *impossible* when what they mean is that it is *impossible given the current schedule*. Marketing organizations may create business plans based on aggregated and statistical data without fully understanding how individual users and customers are likely to behave. Designers, who have gathered detailed, qualitative research on users, may have insight into the business model from a unique perspective. Design works best when a relationship of mutual trust and respect exists among design, business, and engineering.

Elegant interaction design

Elegance is defined in the dictionary as both "gracefulness and restrained beauty of style" and "scientific precision, neatness, and simplicity." We believe that elegance in design, or at least interaction design, incorporates both of these ideals.

Represent the simplest complete solution

One of the classic elements of good design is *economy of form*: using less to accomplish more. In interface design, this means using only the screens and widgets necessary to accomplish the task. This economy extends to behavior: a simple set of tools for the user that allows him to accomplish great things. In visual design, this means using the smallest number of visual distinctions that clearly conveys the desired meaning. Less is more in good design, and designers should endeavor to solve design problems with the fewest additions of form and behavior, in conformance to the mental models of your personas. This concept is well known to developers, who recognize that better algorithms are clearer and shorter.

Yvon Chounard, famed outdoorsman and founder of outdoor clothing company Patagonia, puts it best when he quotes French writer and aviator Antoine de St. Exupéry, who said, "in anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away."

Possess internal coherence

Good design has the feeling of a unified whole, in which all parts are in balance and harmony. Products that are poorly designed, or not designed at all, often look and feel like they are cobbled together from disparate pieces haphazardly knit together. Often this is the result of implementation model construction, where different development teams work on different interface modules without communicating with each other, or where hardware and software are designed independently of each other. This is the antithesis of what we want to achieve. The Goal-Directed Design process, in which product concepts are conceived of as a whole at the top level and then iteratively refined to detail, provides an ideal environment for creating internally coherent designs. Specifically, the use of scenarios to motivate and test designs ensures that solutions are unified by a single narrative thread.

Appropriately accommodate and stimulate cognition and emotion

Many traditionally trained designers speak frequently of *desire* and its importance in the design of communications and products. They're not wrong, but we feel that in placing such emphasis on a single (albeit complex) emotion, they may sometimes be seeing only part of the picture.

Desire is a narrow emotion to appeal to when designing a product that serves a purpose, especially when that product is located in an enterprise, or its purpose is highly technical or specialized. One would hardly want to make a technician operating a radiation therapy system feel desire for the system. We instead want her to feel cautious and perhaps reverent of the rather dangerous energies the system controls. Therefore, we do everything we can as designers to keep her focus on the patient and his treatment. Thus,

in place of what we might call *desire*, the authors believe that elegance (in the sense of gracefulness) means that the user is stimulated and supported both cognitively and emotionally in whatever context she is in.

The remaining chapters of this book enumerate what we view as the most critical interaction and visual interface design principles. No doubt you will discover many more, but this set will more than get you started. The chapters in Part I provided the process and concepts behind the practice of Goal-Directed interaction design. The chapters to come provide a healthy dose of design insight that will help you transform this knowledge into excellent design, whatever your domain.

Interaction Design Principles

Interaction design principles are generally applicable guidelines that address issues of behavior, form, and content. They encourage the design of product behaviors that support the needs and goals of users and create positive experiences with the products we design. These principles are, in effect, a set of rules based on our values as designers and our experiences in trying to live up to those values. At the core of these values is the notion that technology should serve human intelligence and imagination, rather than the opposite. Also, people's experiences with technology should be structured in accordance with their abilities of perception, cognition, and movement.

Principles are applied throughout the design process, helping us translate tasks and requirements that arise from scenarios into formalized structures and behaviors in the interface.

Principles operate at different levels of detail

Design principles operate at several levels of granularity, ranging from the general practice of interaction design down to the specifics of interface design. The lines between these categories are fuzzy, to say the least, but interaction design principles can be generally thought of as falling into the following categories:

- **Conceptual principles** help define *what digital products should be like* and *how they fit structurally* into the broad context of use required by their users. Chapters 8 through 13 discuss conceptual-level design principles.

- **Behavioral principles** describe *how a product should behave*—in general and in specific contexts. Chapters 14 through 17 discuss general behavior-level principles.

- **Interface-level principles** describe effective strategies for the *organization, navigation, and communication of behavior and information*. Chapters 18 through 21 discuss interface-level principles (and patterns) of interaction design.

Most interaction and visual design principles are cross-platform. But some platforms, such as mobile devices and embedded systems, require special consideration because of constraints imposed by factors like screen size, input method, and use context.

Behavioral and interface-level principles minimize work

One of the primary purposes that design principles serve is to optimize the experience of the user when she engages with a product. In the case of productivity tools and most non-entertainment-oriented products, this optimization usually means *minimizing work*.

Most of the design principles described in this book attempt to minimize work one way or another while giving the user greater levels of feedback and contextually useful information. The different types of work to be minimized, as well as some specific strategies for accomplishing this, are the subject of Chapter 12.

Games and other similar kinds of entertainment products require a somewhat different approach than simple minimization of work. They can engage as a result of *requiring* users to do just the right amount of a *certain* kind of work and then rewarding them for doing so. Social games, such as Farmville, have been immensely popular because their players became addicted to the work required to manage and grow their virtual farm (and the pride of sharing their successes with others). Of course, too much work or too little reward turns a game into a chore, as does a clunky interface that puts up roadblocks to easily performing the "fun" work of the game. This kind of gaming interaction design requires a fine touch.

Interaction Design Patterns

Design patterns are a means of capturing useful design solutions and generalizing them to address similar problems. This effort to formalize design knowledge and record best practices can serve several vital purposes:

- Reduce design time and effort on new projects
- Improve the quality of design solutions
- Facilitate communication between designers and developers
- Educate designers

Although the application of patterns in design pedagogy and efficiency is certainly important, we find the development of interaction design patterns to be particularly exciting. They can represent optimal interactions for the user and the class of activity that the pattern addresses.

Architectural patterns and interaction design

The idea of capturing interaction design patterns has its roots in the work of Christopher Alexander, who first described architectural design patterns in his seminal books *A Pattern Language* (Oxford University Press, 1977) and *The Timeless Way of Building* (Oxford University Press, 1979). By defining a rigorous set of architectural features, Alexander sought to describe the essence of architectural design that creates a feeling of well-being on the part of the inhabitants of structures.

This last aim of Alexander's project resonates so closely with the needs of interaction designers. The focus on the human aspects of each pattern differentiates architectural and interaction design patterns from engineering patterns, which are primarily intended as a way to reuse and standardize programming code.

There is one important difference between interaction design patterns and architectural design patterns. Interaction design patterns are concerned not only with structure and organization of elements but also with dynamic behaviors and changes in elements in response to user activity. It is tempting to view the distinction simply as one of change over time, but these changes are interesting because they occur in response to both application state and human activity. This differentiates them from preordained temporal transitions that can be found in mechanical products and TV and film media (which each have their own distinct set of design patterns).

Recording and using interaction design patterns

Patterns are always context-specific: They are defined to be applicable to common design situations that share similar contexts, constraints, tensions, and forces. When capturing a pattern, it is important to record the context to which the solution applies, one or more specific examples of the solution, the abstracted features common to all the examples, and the rationale behind the solution (why it is a good solution).

For a set of patterns to be useful, they must be meaningfully organized in terms of the contexts in which they are applicable. Such a set is commonly called a *pattern library* or *catalog*. If this set is rigorously defined and specified, and sufficiently complete to describe all the solutions in a domain, it is called a *pattern language*. (But considering the pace of innovation in all types of digital products, it seems unlikely that such a language will stabilize anytime soon.)

Design patterns are *not* recipes or plug-and-play solutions. In her book, *Designing Interfaces*, a broad and useful collection of interaction design patterns, Jenifer Tidwell provides the following caveat: "[Patterns] aren't off-the-shelf components; each implementation of a pattern differs a little from every other."[1]

It can be tempting in the world of software design to imagine that a comprehensive catalog of patterns could, given a clear idea of user needs, permit even novice designers to assemble coherent design solutions rapidly and easily. Although we have observed that there is some truth to this notion in the case of seasoned interaction designers, it is simply never the case that patterns can be mechanically assembled in cookie-cutter fashion, without knowledge of the context in which they will be used. As Christopher Alexander is swift to point out, architectural patterns are the antithesis of prefab building, because context is of absolute importance in defining the manifest form of the pattern in the world. The environment where the pattern is deployed is critical, as are the other patterns that compose it, contain it, and abut it. The same is true for interaction design patterns. The core of each pattern lies in the relationships between represented objects and between those objects and the user's goals. (This is one reason why a general style guide can never be a substitute for a context-specific design solution.) The pattern's precise form is certain to be somewhat different for each instance, and the objects that define it will naturally vary from domain to domain, but the relationships between objects remain essentially the same.

Types of interaction design patterns

Like most other design patterns, interaction design patterns can be hierarchically organized from the system level down to the level of individual interface widgets. Like principles, they can be applied at different levels of organization (and, as with design principles, the distinctions between these different levels are sometimes quite fuzzy):

- **Postural** patterns can be applied at the conceptual level and help determine the overall product stance in relation to the user. An example of a postural pattern is "transient," which means that a person uses it for only brief periods of time so that a larger goal can be achieved elsewhere. The concept of product posture and its most significant patterns are discussed at length in Chapter 9.

- **Structural** patterns solve problems that relate to the arrangement of information and functional elements on the screen. Structural patterns have become increasingly documented, especially with the rise in popularity of mobile user interfaces and platforms such as iOS and Android. They consist of views, panes, and other element groupings, discussed in Part III.

- **Behavioral** patterns solve wide-ranging problems relating to specific interactions with functional or data elements. What most people think of as widget behaviors fall into this category, and many such lower-level patterns are discussed in Part III.

Building a mental catalog of patterns is one of the most critical aspects of the education of an interaction designer. As we all become aware of the best parts of each other's work, we can collectively advance the interaction idioms we provide to our users. By leveraging existing work, we can focus on solving new problems, rather than reinventing the wheel.

Sample interaction design patterns

The following sections describe a couple of interaction design patterns; others are discussed in more detail in Part III of this book.

Desktop: Organizer-Workspace

One of the most commonly used high-level structural patterns in the desktop application universe is apparent in Microsoft Outlook. Its navigational (organizer) pane is on the left. Its workspace is on the right; it consists of an overview pane at the top and a detail pane at the bottom (see Figure 7-1).

This pattern is optimal for full-screen applications that require user access to many different kinds of objects, manipulation of those objects in groups, and display of detailed content or attributes of individual objects or documents. The pattern permits all this to be done smoothly in a single screen without the need for additional windows. Many e-mail clients use this pattern, and variations of it appear in many authoring and information management tools where rapid access to and manipulation of various types of objects is common.
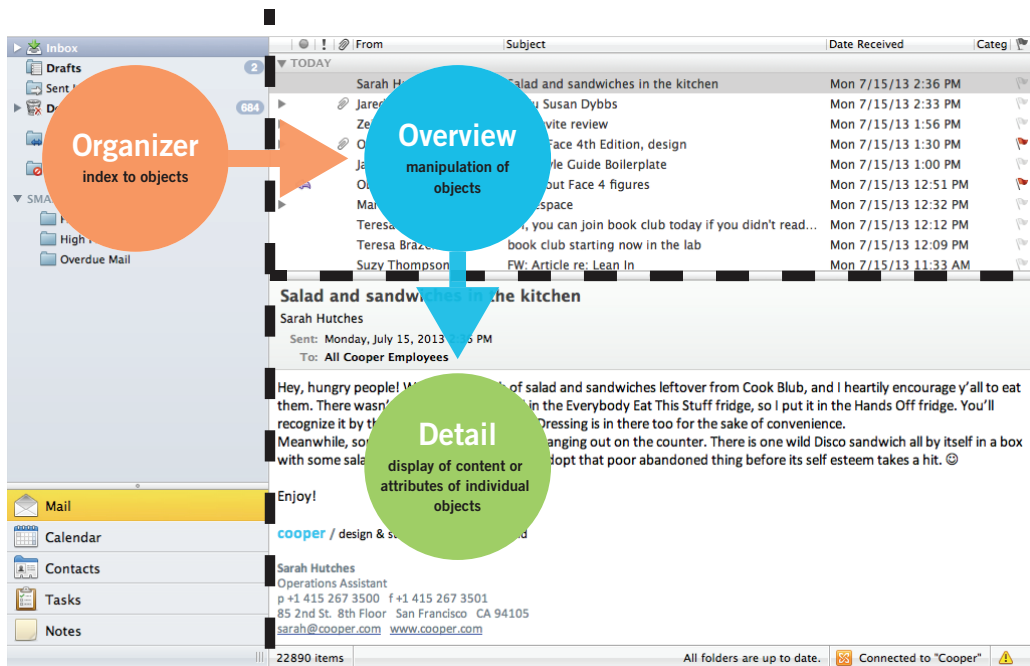
Figure 7-1: The primary structural pattern used by Microsoft Outlook is widely used throughout the industry, across many diverse product domains. The left-vertical pane provides navigation and drives the content of the overview pane in the upper right. A selection in this pane populates the lower-right pane with detail or document content.

Mobile: Double Drawer

Pioneered by Facebook and Path mobile apps, drawers that are exposed by swiping the main content view to the right (to expose the left drawer) or left (to expose the right drawer) are now common in many mobile apps on iOS and Android (see Figure 7-2). The left drawer typically contains primary navigation for the mobile app. The right drawer, when it exists, typically is used for access to an ancillary list of objects (such as friends in the case of Facebook).

Figure 7-2: Facebook's Double Drawer is another structural pattern that has become nearly as ubiquitous for mobile apps as the Organizer-Workspace pattern has for desktop apps. The drawer on the left provides navigation and drives the app's content view. The pane on the right is used for rapid global access to a list of ancillary objects—in this case, your Facebook friends.

Like the Organizer-Workspace pattern, the Double Drawer is almost perfectly optimized for its context—in this case, mobile rather than desktop. Swiping across the main content pane to reveal lists of navigation or communications options is a simple gross motor action, ideal for one-handed use. Selecting from the open drawers is similarly easy to do with the thumb, and the animated opening and shutting of the drawers is aesthetically satisfying as well. It's no wonder this pattern has been so quickly and widely adopted.

Notes

1. Tidwell, 2006

DIGITAL ETIQUETTE

In *The Media Equation* (Cambridge University Press, 1996), Stanford sociologists Clifford Nass and Byron Reeves make a compelling case that humans treat and respond to computers and other interactive products as if they were people. We should thus pay real attention to the "personality" projected by our digital products. Are they quietly competent and helpful, or do they whine, nag, badger, and make excuses?

Nass and Reeves' research suggests that humans have instincts that tell them how to behave around other sentient beings. As soon as an object exhibits sufficient levels of interactivity—such as that found in your average software application—these instincts are activated. Our reaction to software as sentient is both unconscious and unavoidable.

The implication of this research is profound: If we want users to like our products, we should design them to behave in the same manner as a likeable person. If we want users to be productive with our software, we should design it to behave like a supportive human colleague. To this end, it's useful to consider the appropriate working relationship between human beings and computers.

> **DESIGN PRINCIPLE**
>
> *The computer does the work, and the person does the thinking.*

The ideal division of labor in the digital age is very clear: The computer should do the work, and the person should do the thinking. Science fiction writers and computer

scientists tantalize us with visions of artificial intelligence—computers that think for themselves. However, humans don't really need that much help in the thinking department. Our ability to identify patterns and solve complex problems creatively is currently unmatched in the world of silicon. We *do* need a lot of help with the work of information *management*—activities like accessing, analyzing, organizing, and visualizing information. But the actual decisions made from that information are best made by us—the "wetware."

Designing Considerate Products

Nass and Reeves suggest that software-enabled products should be *polite*, but we prefer the term *considerate*. Although politeness could be construed as a matter of manners and protocol—saying "please" and "thank you," but doing little else that is helpful—being truly considerate means being concerned with the needs of others. Above and beyond performing basic functions, considerate software has the goals and needs of its users as a concern.

If an interactive product is stingy with information, obscures its processes, forces users to hunt around for common functions, and is quick to blame people for its own failings, users are sure to have an unpleasant and unproductive experience. This will happen regardless of how polite, cute, visually metaphoric, anthropomorphic, or full of interesting content the software is.

On the other hand, interactions that are respectful, generous, and helpful will go a long way toward creating a positive experience for people using your product.

> **DESIGN PRINCIPLE**
>
> *Software should behave like a considerate human being.*

Building considerate products is not necessarily more difficult than building rude or inconsiderate products, but it requires that you envision interactions that emulate the qualities of a sensitive and caring person. Humans have many wonderful characteristics that make them considerate, and some of these can be emulated to a greater or lesser degree by interactive products. We think the following are some of the most important characteristics of considerate interactive products (and humans):

- Take an interest
- Are deferential
- Are forthcoming

- Use common sense
- Use discretion
- Anticipate people's needs
- Are conscientious
- Don't burden you with their personal problems
- Keep you informed
- Are perceptive
- Are self-confident
- Don't ask a lot of questions
- Fail gracefully
- Know when to bend the rules
- Take responsibility
- Help you avoid awkward mistakes

None of these characteristics, by the way, is at odds with more pragmatic goals of functional data processing (which lies at the core of all silicon-enabled products). Let's discuss them in more detail.

Considerate products take an interest

A considerate friend wants to know more about you. He remembers your likes and dislikes so that he can please you in the future. Everyone appreciates being treated according to his or her personal tastes.

Most software, on the other hand, doesn't know or care who is using it. Little, if any, of the *personal* software on our *personal* computers seems to remember anything *personal* about us, in spite of the fact that we use it constantly, repetitively, and exclusively. A good example of this behavior is how browsers such as Firefox and Microsoft Internet Explorer remember information that users routinely enter into forms on websites, such as a shipping address or username. Google Chrome even remembers these details across devices and sessions.

Software should work hard to remember our habits and, particularly, everything we tell it. From the perspective of a developer writing an application, it can be tempting to think about gathering a bit of information from a person as being similar to gathering a bit of information from a database. Every time the information is needed, the product asks the user for it. The application then discards that tidbit, assuming that it might change and that it can merely ask for it again if necessary. Not only are digital products better suited to recording things in memory than humans are, but our products also show they are

inconsiderate when they forget. Remembering humans' actions and preferences is one of the best ways to create a positive experience with a software-enabled product. We'll discuss the topic of memory in detail later in this chapter.

Considerate products are deferential

A good service provider defers to her client. She understands that the person she is serving is the boss. When a restaurant host shows us to a table in a restaurant, we consider his choice of table to be a suggestion, not an order. If we politely request another table in an otherwise empty restaurant, we expect to be accommodated. If the host refuses, we are likely to choose a different restaurant where *our* desires take precedence over the host's.

Inconsiderate products supervise and pass judgment on human actions. Software is within its rights to express its *opinion* that we are making a mistake, but it is presumptuous for it to judge or limit our actions. Software can *suggest* that we not "submit" our entry until we've typed in our telephone number, and it should explain the consequences if we do so, but if we want to "submit" without the number, we expect the software to do as it is told. The very word *submit* is a reversal of the deferential relationship we should expect from interactive products. Software should submit to users. Any application that proffers a "submit" button is being rude, as well as oblique and confusing.

Considerate products are forthcoming

If you ask a good retail sales associate for help locating an item, he will not only answer your question, but also volunteer useful collateral information. For example, he might tell you that a more expensive, higher-quality item than the one you requested is currently on sale for a similar price.

Most software doesn't attempt to provide related information. Instead, it narrowly answers the precise questions we ask it and typically is not forthcoming about other information, even if it is clearly related to our goals. When we tell our word processor to print a document, it doesn't tell us when the paper supply is low, or when 40 other documents are queued before us, or when another nearby printer is free. A helpful human would.

Figuring out the right way to offer potentially useful information can require a delicate touch. Microsoft's Office Assistant "Clippy" was almost universally despised for his smarty-pants comments like "It looks like you're writing a letter. Would you like help?" While we applauded his sentiment, we wished he weren't so obtrusive and could take a hint when it was clear we didn't want his help. After all, a good waiter doesn't interrupt your conversation to ask you if you want more water. He just refills your glass when it's empty, and he knows better than to linger when it's clear that you're in the middle of an intimate moment.

Considerate products use common sense

Offering inappropriate functions in inappropriate places is a hallmark of poorly designed interactive products. Many interactive products put controls for constantly used functions right next to never-used controls. You can easily find menus offering simple, harmless functions adjacent to irreversible ejector-seat-lever expert functions. It's like being seated at a dining table right next to an open grill.

Horror stories also abound of customers offended by computer systems that repeatedly sent them checks for $0.00 or bills for $957,142,039.58. One would think that the system might alert a human in the Accounts Receivable or Payable departments when an event like this happens, especially more than once, but common sense remains a rarity in most information systems.

Considerate products use discretion

Generally speaking, we want our software to remember what we do and what we tell it. But there are some things our that software probably shouldn't remember unless we specifically direct it, such as credit card numbers, tax IDs, bank accounts, and passwords. Furthermore, it should help us protect this kind of private data by helping us choose secure passwords, and reporting any possible improprieties, such as accounts accessed from an unrecognized computer or location.

Considerate products anticipate people's needs

A human assistant knows that you will require a hotel room when you travel to another city, even when you don't ask explicitly. She knows the kind of room you like and reserves one without any request on your part. She anticipates your needs.

A web browser spends most of its time idling while we peruse web pages. It could easily anticipate our needs and prepare for them while we are reading. It could use that idle time to preload all the links that are visible. Chances are good that we will soon ask the browser to examine one or more of those links. It is easy to abort an unwanted request, but it is always time-consuming to wait for a request to be filled. We'll discuss more ways for software to use idle time to our advantage later in this chapter.

Considerate products are conscientious

A conscientious person has a larger perspective on what it means to perform a task. Instead of just washing the dishes, for example, a conscientious person also wipes down the counters and empties the trash, because those tasks are also related to the larger

goal: cleaning up the kitchen. A conscientious person, when drafting a report, also puts a handsome cover page on it and makes enough photocopies for the entire department.

Here's an example: If we hand our imaginary assistant, Rodney, a manila folder and tell him to file it, he checks the writing on the folder's tab—let's say it reads MicroBlitz Contract—and proceeds to find the correct place for it in the filing cabinet. Under M, he finds, to his surprise, a manila folder already there with the same name. Rodney notices the discrepancy and finds that the existing folder contains a contract for 17 widgets delivered four months ago. The new folder, on the other hand, is for 34 sprockets slated for production and delivery in the next quarter. Conscientious Rodney changes the name on the old folder to read MicroBlitz Widget Contract, 7/13 and then changes the name of the new folder to read MicroBlitz Sprocket Contract, 11/13. This type of initiative is why we think Rodney is conscientious.

Our former imaginary assistant, Elliot, was a complete idiot. He was not conscientious at all. If he were placed in the same situation, he would have dumped the new MicroBlitz Contract folder next to the old MicroBlitz Contract folder without a second thought. Sure, he got it safely filed, but he could have done a better job that would have improved our ability to find the right contract in the future. That's why Elliot isn't our imaginary assistant anymore.

If we rely on a typical word processor to draft the new sprocket contract and then try to save it in the MicroBlitz folder, the application offers the choice of either overwriting and destroying the old widget contract or not saving it at all. The application not only isn't as capable as Rodney, it isn't even as capable as Elliot. The software is dumb enough to assume that because two folders have the same name, we meant to throw away the old one.

The application should, at the very least, mark the two files with different dates and save them. Even if the application refuses to take this "drastic" action unilaterally, it could at least show us the old file (letting us rename *that* one) before saving the new one. The application could take numerous actions that would be more conscientious.

Considerate products don't burden you with their personal problems

At a service desk, the agent is expected to keep mum about her problems and to show a reasonable interest in yours. It might not be fair to be so one-sided, but that's the nature of the service business. An interactive product, too, should keep quiet about its problems and show interest in the people who use it. Because computers don't have egos or tender sensibilities, they should be perfect in this role, but they typically behave the opposite way.

Software whines at us with error messages, interrupts us with confirmation dialog boxes, and brags to us with unnecessary notifiers ("Document Successfully Saved!" How nice for you, Mr. App: Do you ever *unsuccessfully* save?). We aren't interested in the application's crisis of confidence about whether to purge its Recycle Bin. We don't want to hear its whining about being unsure where to put a file on disk. We don't need to see information about the computer's data transfer rates and its loading sequence, any more than we need information about the customer service agent's unhappy love affair. Not only should software keep quiet about its problems, but it should also have the intelligence, confidence, and authority to fix its problems on its own. We discuss this subject in more detail in Chapter 15.

Considerate products keep you informed

Although we don't want our software pestering us incessantly with its little fears and triumphs, we do want to be kept informed about the things that matter to *us*. We don't want our local bartender to grouse to us about his recent divorce, but we appreciate it when he posts his prices in plain sight and when he writes what time the pregame party begins on his chalkboard, along with who's playing and the current Vegas spread. Nobody interrupts us to tell us this information: It's there in plain view whenever we need it. Software, similarly, can provide us with this kind of rich modeless feedback about what is going on. Again, we discuss how in Chapter 15.

Considerate products are perceptive

Most of our existing software is not very perceptive. It has a very narrow understanding of the scope of most problems. It may willingly perform difficult work, but only when given the precise command at precisely the correct time. For example, if you ask the inventory query system how many widgets are in stock, it will dutifully ask the database and report the number as of the time you ask. But what if, 20 minutes later, someone in the Dallas office cleans out the entire stock of widgets? You are now operating under a potentially embarrassing misconception, while your computer sits there, idling away billions of wasted instructions. It is not being perceptive. If you want to know about widgets once, isn't that a good clue that you probably will want to know about widgets again? You may not want to hear widget status reports every day for the rest of your life, but maybe you'll want to get them for the rest of the week. Perceptive software observes what users are doing and uses those observations to offer relevant information.

Products should also watch our preferences and remember them without being asked explicitly to do so. If we always maximize an application to use the entire available screen, the application should get the idea after a few sessions and always launch in that configuration. The same goes for placement of palettes, default tools, frequently used templates, and other useful settings.

Considerate products are self-confident

Interactive products should stand by their convictions. If we tell the computer to discard a file, it shouldn't ask, "Are you sure?" Of course we're sure; otherwise, we wouldn't have asked. It shouldn't second-guess us or itself.

On the other hand, if the computer has any suspicion that we might be wrong (which is always), it should anticipate our changing our minds by being prepared to undelete the file upon our request.

How often have you clicked the Print button and then gone to get a cup of coffee, only to return to find a fearful dialog box quivering in the middle of the screen, asking, "Are you sure you want to print?" This insecurity is infuriating and the antithesis of considerate human behavior.

Considerate products don't ask a lot of questions

Inconsiderate products ask lots of annoying questions. Excessive choices, especially in the form of questions, quickly stop being a benefit and instead become an ordeal.

Asking questions is quite different from providing choices. When browsing on your own in a store, you are presented with *choices*. When going to a job interview, you are asked *questions*. Which is the more pleasurable experience? Part of the reason why is that individual asking the questions is understood to be in a position superior to the individual being asked. Those with authority ask questions; subordinates respond. When software asks questions rather than offering choices, users feel disempowered.

Beyond the power dynamics issues, questions also tend to make people feel badgered and harassed. *Would you like soup or salad?* Salad. *Would you like cabbage or spinach?* Spinach. *Would you like French, Thousand Island, or Italian?* French. *Would you like lite or regular?* Stop! Please, just bring me the soup instead! *Would you like corn chowder or chicken noodle?*

Users *really* don't like to be asked questions by products, especially since most of the questions are stupid or unnecessary. Asking questions tells users that products are:

- Ignorant
- Forgetful
- Weak
- Fretful
- Lacking initiative
- Overly demanding

These are all qualities that we typically dislike in people. Why should we desire them in our products? The application is not asking us our opinion out of intellectual curiosity or a desire to make conversation, the way a friend might over dinner. Rather, it is behaving ignorantly while also representing itself as having false authority. The application isn't interested in our opinions; it requires information—often information it didn't really need to ask us in the first place.

Many ATMs continually ask users what language they prefer: "Spanish, English, or Chinese?" This answer is unlikely to change after a person's first use. Interactive products that ask fewer questions, provide choices without asking questions, and remember information they have already learned appear smarter to users, as well as more polite and considerate.

Considerate products fail gracefully

When your friend commits a serious faux pas, he tries to make amends and undo the damage. When an application discovers a fatal problem, it can take the time and effort to prepare for its failure without hurting the user, or it can simply crash and burn.

Many applications are filled with data and settings. When they crash, that information is still often discarded. The user is left holding the bag. For example, say an application is computing merrily along, downloading your e-mail from a server, when it runs out of memory at some procedure buried deep in the application's internals. The application, like most desktop software, issues a message that says, in effect, "You are hosed," and it terminates immediately after you click OK. You restart the application, or sometimes the whole computer, only to find that the application lost your e-mail. When you interrogate the server, you find that it has also erased your e-mail because the e-mail was already handed over to your application. This is not what we should expect of good software.

In this example, the application accepted e-mail from the server—which then erased its copy—but it didn't ensure that the e-mail was properly recorded locally. If the e-mail application had ensured that those messages were promptly written to the local disk, even before it informed the server that the messages were successfully downloaded, the problem would never have arisen.

Some well-designed software products, such as Ableton Live, a brilliant music performance tool, rely on the Undo cache to recover from crashes. This is a great example of how products can easily keep track of user behavior, so if some situation causes problems, it is easy to extricate yourself.

Even when applications don't crash, inconsiderate behavior is rife, particularly on the web. Users often need to enter information into a set of forms on a page. After filling in 10 or 11 fields, a user might click the Submit button, and, due to some mistake or omission

on his part, have the site reject his input and tell him to correct it. The user then clicks the back arrow to return to the page, and lo, the 10 valid entries were inconsiderately discarded along with the single invalid one. Remember your incredibly mean junior high geography teacher who ripped up your report on South America because you wrote it in pencil instead of ink? And as a result, you hate geography to this day? Don't create products that act like that!

Considerate products know when to bend the rules

When manual information-processing systems are translated into computerized systems, something is lost in the process. Although an automated order-entry system can handle millions more orders than a human clerk can, the human clerk can *work the system* in a way that most automated systems ignore. There is almost never a way to jigger the functioning to give or take slight advantages in an automated system.

In a manual system, when the clerk's friend from the sales force tells him that getting a particular order processed speedily means additional business, the clerk can expedite that one order. When another order comes in with some critical information missing, the clerk still can process it, remembering to acquire and record the information later. This flexibility usually is absent from automated systems.

Most computerized systems have only two states: nonexistence and full compliance. No intermediate states are recognized or accepted. Every manual system has an important but paradoxical state—unspoken, undocumented, but widely relied upon—**suspense**. In this state a transaction can be accepted even though it is not fully processed. The human operator creates that state in his head or on his desk or in his back pocket.

For example, suppose a digital system needs both customer and order information before it can post an invoice. Whereas the human clerk can go ahead and post an order before having detailed customer information, the computerized system rejects the transaction, unwilling to allow the invoice to be entered without it.

The characteristic of manual systems that lets humans perform actions out of sequence or before prerequisites are satisfied is called *fudgeability*. It is one of the first casualties when systems are computerized, and its absence is a key contributor to the inhumanity of digital systems. It is a natural result of the implementation model. Developers don't always see a reason to create intermediate states, because the computer has no need for them. Yet there are strong human needs to be able to bend the system slightly.

One of the benefits of fudgeable systems is reducing the number of mistakes. By allowing many small, temporary mistakes into the system and entrusting humans to correct them before they cause problems downstream, we can avoid much bigger, more permanent mistakes. Paradoxically, most of the hard-edged rules enforced by computer systems are

imposed to prevent just such mistakes. These inflexible rules make the human and the software adversaries. Because the human is prevented from fudging to prevent big mistakes, he soon stops caring about protecting the software from colossal problems. When inflexible rules are imposed on flexible humans, both sides lose. It is bad for business to prevent humans from doing things the way they want, and the computer system usually ends up having to digest invalid data anyway.

In the real world, both missing information and extra information that doesn't fit into a standard field are important tools for success. For example, suppose a transaction can be completed only if the termination date is extended two weeks beyond the official limit. Most companies would rather fudge on the termination date than see a million-dollar deal go up in smoke. In the real world, limits are fudged all the time. Considerate products need to realize and embrace this fact.

Considerate products take responsibility

Too many interactive products take the attitude that "It isn't my responsibility." When they pass along a job to some hardware device, they wash their hands of the action, leaving the stupid hardware to finish. Any user can see that the software isn't being considerate or conscientious, that the software isn't shouldering its part of the burden of helping the user become more effective.

In a typical print operation, for example, an application begins sending a 20-page report to the printer and simultaneously displays a print process dialog box with a Cancel button. If the user quickly realizes that he forgot to make an important change, he clicks the Cancel button just as the first page emerges from the printer. The application immediately cancels the print operation. But unbeknownst to the user, while the printer was beginning to work on page 1, the computer had already sent 15 pages to the printer's buffer. The application cancels the last five pages, but the printer doesn't know about the cancellation; it just knows that it was sent 15 pages, so it goes ahead and prints them. Meanwhile, the application smugly tells the user that the function was canceled. The application lies, as the user can plainly see.

The user is unsympathetic to the communication problems between the application and the printer. He doesn't care that the communications are one-way. All he knows is that he decided not to print the document before the first page appeared in the printer's output tray, he clicked the Cancel button, and then the stupid application continued printing for 15 pages even after it acknowledged his Cancel command.

Imagine what his experience would be if the application could properly communicate with the print driver. If the software were smart enough, the print job could easily have been abandoned before the second sheet of paper was wasted. The printer has a Cancel function—it's just that the software was built to be too lazy to use it.

Considerate products help you avoid awkward mistakes

If a helpful human companion saw you about to do something that you would almost certainly regret afterwards—like shouting about your personal life in a room full of strangers, or sending an empty envelope in the mail to your boss—they might take you quietly aside and gently alert you to your mistake.

Digital products should similarly help you realize when you are, for example, about to inadvertently send a text to your entire list of contacts instead of the one friend you were intending to confide in, or are about to send an e-mail to the director of your department without the quarterly report you mentioned you were enclosing in the text of your message.

The intervention should not, however, be in the form of a standard modal error message box that stops the action and adds insult to injury, but rather through careful visual and textual feedback that lets you know that you are messaging a group rather than a person, or that you haven't enclosed any attachments even though you mentioned that you were.

For this latter situation, your e-mail app might even modelessly highlight the drop area for you to drag your attachment to, while at the same time giving you the option of just going ahead and sending the message sans attachments, in case the software got your intentions wrong.

Products that go the extra mile in looking out for users by helping them prevent embarrassing mistakes—and not berating them for it—will quickly earn their trust and devotion. All other things equal, considerate product design is one of the things, and perhaps even *the* thing, that distinguishes an only passable app from a truly great one.

Designing Smart Products

In addition to being considerate, helpful products and people must be *smart*. Thanks to science fiction writers and futurists, there is some confusion about what it means for an interactive product to be smart. Some naive observers think that smart software can actually behave intelligently.

While that might be nice, our silicon-enabled tools are still a ways away from delivering on that dream. A more useful understanding of the term (if you're trying to ship a product this decade) is that these products can work hard even when conditions are difficult and even when users aren't busy. Regardless of our dreams of thinking computers, there is a much greater and more immediate opportunity to get our computers to work harder.

The remainder of this chapter discusses some of the most important ways that software can work a bit harder to serve humans better.

Smart products put idle cycles to work

Because every instruction in every application must pass single-file through a CPU, we tend to optimize our code for this needle's eye. Developers work hard to keep the number of instructions to a minimum, ensuring snappy performance for users. What we often forget, however, is that as soon as the CPU has hurriedly finished all its work, it waits idle, doing nothing, until the user issues another command. We invest enormous effort in reducing the computer's reaction time, but we invest little or no effort in putting it to work proactively when it is not busy reacting to the user. Our software commands the CPU as though it were in the army, alternately telling it to hurry up and wait. The hurry up part is great, but the waiting needs to stop.

In our current computing systems, users need to remember too many things, such as the names they give to files and the precise location of those files in the file system. If a user wants to find that spreadsheet with the quarterly projections on it, he must either remember its name or go browsing. Meanwhile, the processor just sits there, wasting billions of cycles.

Most current software also takes no notice of context. When a user struggles with a particularly difficult spreadsheet on a tight deadline, for example, the application offers precisely as much help as it offers when he is noodling with numbers in his spare time. Software can no longer, in good conscience, waste so much idle time while users work. It is time for our computers to begin shouldering more of the burden of work in our day-to-day activities.

Most users in normal situations can't do anything in less than a few seconds. That is enough time for a typical desktop computer to execute at least a *billion* instructions. Almost without fail, those interim cycles are dedicated to idling. The processor does *nothing* except wait. The argument against putting those cycles to work has always been "We can't make assumptions; those assumptions might be wrong." Our computers today are so powerful that, although the argument is still true, it is frequently irrelevant. Simply put, it doesn't matter if the application's assumptions are wrong; it has enough spare power to make several assumptions and discard the results of the bad ones when the user finally makes his choice.

With Windows and Apple's OS X's preemptive, threaded multitasking and multicore, multichip computers, the computer can perform extra work in the background without significantly affecting the performance most users see. The application can launch a search for a file and, if the user begins typing, merely abandon the search until the next hiatus. Eventually, the user stops to think, and the application has time to scan the

whole disk. The user won't even notice. This is precisely the kind of behavior that makes OS X's Spotlight search capabilities so good. Search results are almost instantaneous, because the operating system takes advantage of downtime to index the hard drive.

Every time an application puts up a modal dialog box, it goes into an idle waiting state, doing no work while the user struggles with the dialog. This should never happen. It would not be hard for the dialog box to hunt around and find ways to help. What did the user do last time? The application could, for example, offer the previous choice as a suggestion for this time.

We need a new, more proactive way of thinking about how software can help people reach their goals and complete their tasks.

Smart products have a memory

When you think about it, it's pretty obvious that for a person to perceive an interactive product as considerate and smart, that product must have some knowledge of the person and be capable of learning from his or her behavior. Looking through the characteristics of considerate products presented earlier reinforces this fact: For a product to be truly helpful and considerate, it must *remember* important things about the people interacting with it.

Developers and designers often assume that user behavior is random and unpredictable and that users must be continually interrogated to determine the proper course of action. Although human behavior certainly isn't deterministic, like that of a digital computer, it is rarely random, so being asked silly questions is predictably frustrating for users.

Because of assumptions like this, most software is forgetful, remembering little or nothing from execution to execution. If our applications *are* smart enough to retain any information during and between uses, it is usually information that makes the job easier for the *developer*, not the user. The application willingly discards information about how it was used, how it was changed, where it was used, what data it processed, who used it, and whether and how frequently the application's various facilities were used. Meanwhile, the application fills initialization files with driver names, port assignments, and other details that ease the developer's burden. It is possible to use the exact same facilities to dramatically increase the software's smarts from the user's perspective.

If your application, website, or device could predict what a user will do next, couldn't it provide a better interaction? If your application could know which selections the user will make in a particular dialog box or form, couldn't that part of the interface be skipped? Wouldn't you consider advance knowledge of what actions your users take to be an awesome secret weapon of interface design?

Well, you *can* predict what your users will do. You *can* build a sixth sense into your application that will tell it with uncanny accuracy exactly what the user will do next! All those billions of wasted processor cycles can be put to great use: All you need to do is give your interface a **memory**.

When we use the term *memory* in this context, we don't mean RAM, but rather a facility for tracking and responding to user actions over multiple sessions. If your application simply remembers what the user did the last several times (and how), it can use that as a guide to how it should behave the next time.

If we enable our products with an awareness of user behavior, a memory, and the flexibility to present information and functionality based on previous user actions, we can realize great advantages in user efficiency and satisfaction. We would all like to have an intelligent and self-motivated assistant who shows initiative, drive, good judgment, and a keen memory. A product that makes effective use of its memory is more like that self-motivated assistant, remembering helpful information and personal preferences without needing to ask. Simple things can make a big difference—the difference between a product your users tolerate and one they *love*. The next time you find your application asking your users a question, make it ask itself one instead.

Smart products anticipate needs

Predicting what a user will do by remembering what he did last is based on the principle of *task coherence*. This is the idea that our goals and how we achieve them (via tasks) is generally similar from day to day. This is true not only for tasks such as brushing our teeth and eating breakfast, but it also describes how we use our word processors, e-mail applications, mobile devices, and enterprise software.

When a consumer uses your product, there is a good chance that the functions he uses and how he uses them will be very similar to what he did in previous uses of the product. He may even be working on the same documents, or at least the same types of documents, located in similar places. Sure, he won't be doing the exact same thing each time, but his tasks will likely be variants of a limited number of repeated patterns. With significant reliability, you can predict your users' behavior by the simple expedient of remembering what they did the last several times they used the application. This allows you to greatly reduce the number of questions your application must ask the user.

For example, even though Sally may use Excel in dramatically different ways than she does PowerPoint or Word, she tends to use Excel the same way each time. Sally prefers 12-point Helvetica and uses that font and size with dependable regularity. It isn't really necessary for the application to ask Sally which font to use or to revert to the default. A reliable starting point would be 12-point Helvetica, every time.

Applications can also pay closer attention to clusters of actions. For example, in your word processor, you might often reverse-out text, making it white on black. To do this, you select some text and change the font color to white. Without altering the selection, you then set the background color to black. If the application paid enough attention, it would notice that you requested two formatting steps without an intervening selection option. As far as you're concerned, this is effectively a single operation. Wouldn't it be nice if the application, upon seeing this unique pattern repeated several times, automatically created a new format style of this type—or, better yet, created a new Reverse-Out toolbar control?

Most mainstream applications allow their users to set defaults, but this doesn't fit the bill as a smart behavior. Configuration of this kind is an onerous process for all but power users. Many users will never understand how to customize defaults to their liking.

Smart products remember details

You can determine what information the application should remember with a simple rule: If it's worth it to the user to do it, it's worth it to the application to remember it.

Everything that users do should be remembered. Our hard drives have plenty of storage, and a memory for your application is a good investment of storage space. We tend to think that applications are wasteful, because a big application might consume 200 MB of disk space. That might be typical usage for an application, but not for most user data. If your word processor saved 1 KB of execution notes every time you ran it, it still wouldn't amount to much. Suppose you use your word processor 10 times every business day. There are approximately 250 workdays per year, so you run the application 2,500 times a year. The net consumption is still only 2 MB, and that gives an exhaustive recounting of the entire year! This is probably not much more than the background image you put on your desktop.

> **DESIGN PRINCIPLE**
>
> *If it's worth it to the user to do it, it's worth it to the application to remember it.*

Any time your application finds itself with a choice, and especially when that choice is being offered to a user, the application should remember the information from run to run. Instead of choosing a hardwired default, the application can use the previous setting as the default, and it will have a much better chance of giving the user what he wants. Instead of asking the user to make a determination, the application should go ahead and make the same determination the user made last time, and let him change it if it is wrong. Any options users set should be remembered, so that the options remain in effect until manually changed. If a user ignores aspects of an application or turns

them off, they should not be offered again. The user will seek them out when he is ready for them.

One of the most annoying characteristics of applications without memories is that they are so parsimonious with their assistance regarding files and disks. If there is one place where users need help, it's with files and disks. An application like Word remembers the last place a person looked for a file. Unfortunately, if she always puts her files in a folder called Letters, and then she edits a document template stored in the Templates folder just once, all her subsequent letters are saved in the Templates folder rather than in the Letters folder. So, the application must remember more than just the last place the files were accessed. It must remember the last place files *of each type* were accessed.

The position of windows should also be remembered; if you maximized the document last time, it should be maximized next time. If you positioned it next to another window, it is positioned the same way the next time without any instruction from the user. Microsoft Office applications do a good job of this.

Remembering file locations

All file-open facilities should remember where the user gets his files. Most users access files from only a few folders for each given application. The application should remember these source folders and offer them in the Open dialog box. The user should never have to step through the tree to a given folder more than once.

Deducing information

Software should not simply remember these kinds of explicit facts; it should also remember useful information that can be deduced from these facts. For example, if the application remembers how many bytes changed in the file each time it is opened, it can help the user with some checks of reasonableness. Imagine that a file's changed-byte count is 126, 94, 43, 74, 81, 70, 110, and 92. If the user calls up the file and changes 100 bytes, nothing would be out of the ordinary. But if the number of changed bytes suddenly shoots up to 5,000 (as a result, perhaps, of inadvertently deleting a page of content), the application might suspect that something is amiss. Although there is a chance that the user has inadvertently done something that will make him sorry, the probability of that is low, so it isn't right to bother him with a confirmation dialog. It is, however, very reasonable for the application to keep a milestone copy of the file before the 5,000 bytes were changed, just in case. The application probably won't need to keep it beyond the next time the user accesses that file, because the user will likely spot any glaring mistake and will then perform an Undo.

Multi-session Undo

Most applications discard their stack of Undo actions when the user closes the document or application. This is very shortsighted on the application's part. Instead, the application could write the Undo stack to a file. When the user reopens the file, the application could reload its Undo stack with the actions the user performed the last time the application was run—even if that was a week ago!

Past data entries

An application with a better memory can reduce the number of errors users make. This is simply because users have to enter less information. More of it will be entered automatically from the application's memory. Many browsers offer this function, though few mobile or desktop applications do. In an invoicing application, for example, if the software enters the date, department number, and other standard fields from memory, the invoicing clerk has fewer opportunities to make typing errors in those fields.

If the application remembers what the user enters and uses that information for future reasonableness checks, the application can work to keep erroneous data from being entered. Contemporary web browsers such as Internet Explorer and Firefox provide this facility: Named data entry fields remember what has been entered into them before and allow users to pick those values from a combo box. For security-minded individuals, this feature can be turned off, but for the rest of us, it saves time and prevents errors.

Foreign application activities on application files

Applications might also leave a small thread running between invocations. This little application can keep an eye on the files it has worked on. It can track where the files go and who reads and writes to them. This information might be helpful to a user when he next runs the application. When he tries to open a particular file, the application can help him find it, even if it has been moved. The application can keep the user informed about what other functions were performed on his file, such as whether it was printed or e-mailed. Sure, this information might be unneeded, but the computer can easily spare the time, and only bits have to be thrown away, after all.

Making smart products work

A remarkable thing happens to the software design process when the power of task coherence is recognized. Designers find that their thinking takes on a whole new quality. The normally unquestioned recourse of popping up a dialog box gets replaced with a more studied process in which the designer asks much more subtle questions. How much should the application remember? Which aspects should it remember? Should

the application remember more than just the last setting? What constitutes a change in pattern? Designers start to imagine situations like this: The user accepts the same date format 50 times in a row and then manually enters a different format once. The next time the user enters a date, which format should the application use? The format used 50 times, or the more-recent one-time format? How many times must the new format be specified before it becomes the default? Even though ambiguity exists, the application still shouldn't ask the user. It must use its initiative to make a reasonable decision. The user is free to override the application's decision if it is the wrong one.

The following sections explain some characteristic patterns in how people make choices. These can help us resolve these more-complex questions about task coherence.

Decision-set reduction

People tend to reduce an infinite set of choices to a small, finite set of choices. Even when you don't do the exact same thing each time, you tend to choose your actions from a small, repetitive set of options. People perform this *decision-set reduction* unconsciously, but software can take notice and act on it.

For example, just because you went shopping at Safeway yesterday doesn't necessarily mean you shop at Safeway exclusively. However, the next time you need groceries, you probably will shop at Safeway again. Similarly, even though your favorite Chinese restaurant has 250 items on the menu, chances are you usually choose from your personal subset of five or six favorites. When people drive to and from work, they usually choose from a small number of favorite routes, depending on traffic conditions. Computers, of course, can remember four or five things without breaking a sweat.

Although simply remembering the last action is better than not remembering anything, it can lead to a peculiar pathology if the decision set consists of precisely two elements. For example, if you alternately read files from one folder and store them in another, each time the application offers you the last folder, it is guaranteed to be wrong. The solution is to remember more than just one previous choice.

Decision-set reduction guides us to the idea that pieces of information the application must remember about the user's choices tend to come in groups. Instead of one right way, several options are all correct. The application should look for more subtle clues to differentiate which one of the small set is correct. For example, if you use a check-writing application to pay your bills, the application may very quickly learn that only two or three accounts are used regularly. But how can it determine from a given check which of the three accounts is the most likely to be appropriate? If the application remembered the payees and amounts on an account-by-account basis, that decision would be easy. Every time you pay the rent, it is the exact same amount! It's the same with a car payment. The amount paid to the electric company might vary from check to check, but it

probably stays within 10 or 20 percent of the last check written. All this information can be used to help the application recognize what is going on and use that information to help the user.

Preference thresholds

The decisions people make tend to fall into two primary categories: important and unimportant. Any given activity may involve hundreds of decisions, but only a few of them are important. All the rest are insignificant. Software interfaces can use this idea of *preference thresholds* to simplify tasks for users.

After you decide to buy that car, you don't really care who finances it as long as the terms are competitive. After you decide to buy groceries, the particular checkout aisle you select may be unimportant. After you decide to ride the Matterhorn at Disneyland, you don't really care which bobsled they seat you in.

Preference thresholds guide us in our user interface design by demonstrating that asking users for successively detailed decisions about a procedure is unnecessary. After a user chooses to print, we don't have to ask him how many copies he wants or whether the image is landscape or portrait. We can make an assumption about these things the first time out and then remember them for all subsequent invocations. If the user wants to change them, he can always request the Printer Options dialog box.

Using preference thresholds, we can easily track which facilities of the application each user likes to adjust and which are set once and ignored. With this knowledge, the application can offer choices where it has an expectation that the user will want to take control, not bothering him with decisions he won't care about.

Mostly right, most of the time

Task coherence predicts what users will do in the future with reasonable, but not absolute, certainty. If our application relies on this principle, it's natural to wonder about the uncertainty of our predictions. If we can reliably predict what the user will do 80 percent of the time, this means that 20 percent of the time we will be wrong. It might seem that the proper step to take here is to offer users a choice, but this means that they will be bothered by an unnecessary dialog box 80 percent of the time. Rather than offering a choice, the application should go ahead and do what it thinks is most appropriate and allow users to override or undo it. If the Undo facility is sufficiently easy to use and understand, users won't be bothered by it. After all, they will have to use Undo only two times out of ten instead of having to deal with a redundant dialog box eight times out of ten. This is a much better deal for humans.

Designing Social Products

Most of the content created in modern software isn't made for the sole use of its author (task lists, reminders, and personal journals are perhaps the exceptions). Most content is passed on to other humans: Presentations to be heard, documents to be read, images to be seen, status updates to be liked; as well as content to be reviewed, application, and used in the next step of some business process. Even lines of application code that are "meant for machines," are still written in structures and using language that is meant to be readable by other developers.

Traditional, non-connected software left the sharing of this content up to the user, via documents and through software that was specifically built for the task, such as an e-mail client. But as software has grown more complex—and more respectful of the goals of its users—those sharing and collaboration features are built right in. Software is becoming less like a private office into which one disappears to produce work and later emerges, work-in-hand, but more like an open-plan office where collaborators and even consumers are just a shout away.

So far, this chapter has discussed how good software should be considerate to humans. When we communicate with other users via software-mediated mechanisms, the principle is the same, but must additionally take into account social norms and the expectations of the other users.

Social products know the difference between social and market norms

Every group has its own set of rules to which its members adhere, but the two largest categories are *social norms* and *market norms.*

Social norms are the unspoken rules of reciprocity afforded to friends and family. They include things like lending a hand when in need, and expressing gratitude.

Market norms are a different set of unspoken rules afforded to people with whom one is doing business. They include things like assurances of fair price for quality goods and honesty in dealings.

These two concepts are very distinct from each other. Mistakenly using market norms in a social setting can be perceived as extremely rude. Imagine leaving money on the table after enjoying a good meal at a friend's house and then leaving. Mistakenly using social norms in a market setting can get you arrested. Imagine shaking the hand of the waiter at a restaurant and saying, "Thank you, that was delicious!" and leaving without paying.

If your software is to avoid being either rude or illegal, it must know the norms in which its users operate. This is quite often apparent given the domain, but more all-encompassing systems may toy with those boundaries. You are connected to friends through LinkedIn. There are companies with pages on Facebook. Software used internally at an organization operates on semi-social rules, since its members are helping each other do the organization's business. Dating software are actually markets where the transaction is agreeing to meet.

Software adhering to market norms should assure both parties in a transaction that the deal will be fair, often acting as the "matchmaker" and trustworthy escrow agent. Software adhering to social norms should help its users adhere to the subculture's rules and hierarchy in reciprocal, rewarding ways.

Social software lets users present their best side

Representing users in a social interface poses challenges to the designer, that is, how to make those representations unique, recognizable, descriptive, and useful. The following strategies can help users represent themselves online.

User identity

It's tempting to want to use names to represent a user's identity, but names aren't always unique enough, and they don't always lend themselves to compact representations.

It's also tempting to assign semi-random avatars, as Google Docs does with its colored animal icons (while typing this, the author has been joined by an adorable magenta panda). The use of semi-random but canned visuals lets designers keep user avatars in line with the software's look and feel, but this adds cognitive load, as the user must then remember the person behind the icon. To keep this load to a minimum, let the user on the other end provide a visual that works best for them, either by selecting an unclaimed icon-color combination themselves, or by uploading their own image.

Uploading images does introduce the risk that inappropriate content might be chosen, but within opt-in networks and accountable (non-anonymous) social software, social pressures should keep this in check. In any case, adding a ToolTip for a full name offers users a quick reminder about who is who, should they need it.

Dynamic vs static user profiles

User profiles are a traditional way that a user can create an online presence for themselves, but not all users have the time or desire to fill out a bunch of static fields describing themselves. However, a user's social contributions to the network can be dynamically collected

and presented in summary: music listened to, people followed or friended, updates added or liked, books or movies mentioned, links posted or shared, etc. Facebook's Timeline is an example of one way of presenting this kind of information. In social networks, actions speak louder than self-descriptions, and summarized top-rated or best-liked contributions can form a great supplement or even an alternative to static bios.

As with any part of a user's profile, a user should have complete control of who gets to see this information, and should be able to curate it and organize it as they see fit—though the default should also look good for those users who don't want to fuss with it.

Social software permits easy collaboration

Collaboration is one of the most common reasons to make software social. A user may desire a second opinion, another pair of hands, or signoff by a colleague. Collaboration-centric social software can bake this type of functionality deeply into the interface.

For example, Microsoft Word lets people add review comments to documents, and others can add comments referencing earlier ones. It works after a fashion, but comments can unfortunately get jumbled and dissociated from their source.

Google Docs does Word one better by allowing collaborators to reply directly to comments, treating them almost as a threaded discussion, lets any collaborator "resolve" an issue with the click of a button, and supplies a place where old, resolved issues can be re-opened. The Google Docs model better fits the ways people interact to discuss and resolve questions. Designers should make sure that collaboration tools are apparent, usable, and fit their personas' collaboration needs and communication behaviors.

Social products know when to shut the door

In productivity software that is only peripherally social, such as Google Docs, the socialness of the software should not overwhelm or distract from the primary task. Of course social software must manifest other users, but it should do so carefully, treating the attention of the user with the utmost respect. A blinking, beeping announcement that an already-invited user has come into my document is a surefire way to get me to find another alternative. Additionally, users should have access to a polite but firm way to "shut the door," suspending social interruption for a time that the user needs to focus to get a task done.

Social products help networks grow organically

Networks of people grow and change over time as members join, connect, interact, gain standing, and leave. Social software must have ways for new members to discover the

network, join, build a presence, learn the rules, begin to participate, and receive gentle reinforcements when the member transgresses against the subculture's norms. Intermediate members should have tools that let them build mastery, nurture newer members, and seek help from senior members. Senior members should have tools to manage the network. All members need ways to suspend participation for a time, or to elegantly bow out. And, as morbid as the thought may be, networks must elegantly handle when their members die.

A particularly difficult social problem is how to handle when one user wishes to connect to another user who does not agree to the connection. Per social norms, the reluctant user can come across as aloof or standoffish. When the new mailroom intern tries to connect via LinkedIn to the CEO of a large organization, the CEO doesn't want to connect, but also doesn't want to send a signal that that employee isn't valued. To let the eager user save face, the reluctant user needs to pass responsibility to a set of explicit rules for the community, to another user who is expressly charged as a gatekeeper, or to some system constraint.

Social products respect the complexity of social circles

There is a psychosocial threshold to the size of networks that designers must keep in mind. Research from primatology has revealed that the size of the neocortex in a primate's brain has a direct correlation to the average size of its tribe. Tribes with a number of members smaller than this value risk having less security. Tribes with a number of members larger than this value become unstable.

This limit is called *Dunbar's number*, named for the anthropologist who first proposed it, Robin Dunbar. The obvious follow-up question to the primate observation is, of course, what is that value for humans? Given the size of our neocortex, the value is probably around 150. This is a generalized maximum number of social relationships that any one person can maintain fully. If your software permits networks larger than this value, there is a risk of instability unless there are either explicit rules and mechanisms that proscribe behavior, or a set of tools to manage the subsequent complexity.

Some software is slightly social, such as Google Docs, which allows one user to invite others to collaborate on individual documents. The social group here is strictly invite-only and strictly opt-in. Users may have a few levels of access, determined by the owner of the original document. These types of software leave complexity as an exercise for the user.

Other software may have more deliberate networks and rules around participation. Dropbox, for instance, is a file sharing software that allows users to define a network

of who has access to a shared group of files. Users may invite others, even outside of the defined organization, to individual folders or files.

Explicitly social software can involve massively disparate circles of people. Facebook, with its estimated 1 billion users, can find its users connected to coworkers, nuclear family, extended family, friends, frenemies, neighbors, ex-neighbors, school mates, ex-school mates, acquaintances, hobbyist circles, employers, potential employers, employees, clients, potential clients, sweethearts, paramours, and suspicious rivals, to name a few. The rules and norms for each of these subgroups can be of critical importance to the user, as can keeping groups separate.

For example, your college friends may delight in photos of your night of irresponsible carousing, but only some of your family would, and your employer or clients would almost certainly not. Unfortunately the mechanism for managing these circles and specifying which content goes to which circle is hard to find and cumbersome to use, occasionally resulting in some very embarrassed Facebook users. The ability to change permissions on the fly and retract actions is critical in such contexts. Facebook's rival social network, Google Plus, provides its users with a much clearer mechanism for understanding and managing social circles, with animations and modern layouts that make it fun rather than a chore.

Other more explicit *communities of practice* may afford certain permissions to members depending on role and seniority. Wikipedia has a huge amount of readers, a smaller group that writes and edits pages, and an even smaller group that has the administrative ability to grant permissions.

The larger and more complex your software is, the more design attention you will need to pay to the complexity of the social network.

Social products respect other users' privacy

Advertising-driven social media, like Facebook and Google Plus, have strong financial incentives to disrespect their users' privacy. The more that is known and shared about a particular user, the more targeted advertising becomes, and the higher the fees that can be charged to advertisers.

On the other hand, users want to feel that their privacy desires are being respected and can be driven to quit a service that goes too far. Facebook in particular has run into problems time and again where it changes policy to expose more user data; does not explain the change to users' satisfaction; and makes controls for the change difficult to find, understand, and use. Thus a user commenting on a risqué picture is suddenly shocked to find her Aunt Bessie scolding her for it. Facebook's repeated, inconsiderate missteps risk it slowly turning its users against its brand.

Considerate social software makes additional sharing a carefully explained, opt-in affair. And in the case of business norms, it can be an intellectual property rights violation, so take great care.

Social products deal appropriately with the anti-social

Griefers are users who abuse social systems by interfering with transactions, adding noise to conversations, and even sabotaging work. Amazon has had to deal with an entire subculture of griefers who delight in providing sarcastic reviews to goods on sale. In large-network systems with easy and anonymous account creation, griefers have little accountability for their actions and can be a major problem. Social software provides tools for users to silence griefers from ruining transactions they own, tools to categorically exclude griefers, and tools to report griefers to community caretakers. The trick in these tools is to help caretakers distinguish the genuinely anti-social user from those who are earnest but unpopular.

PLATFORM AND POSTURE

As you might recall from Chapter 5, the first question to answer as you begin to design the interaction framework of a digital product is "What *platform* and *posture* are appropriate?"

A product's *platform* can be thought of as the combination of hardware and software that enables the product to function—in terms of both user interactions and the product's internal operations.

A product's *posture* is its behavioral stance—how it presents itself to users. Posture is a way of talking about how much attention the user devotes to interacting with the product, and how the product's behaviors respond to the kind of attention the user devotes to it. This decision must be based on an understanding of likely usage contexts and environments.

Product Platforms

You're no doubt familiar with many of the most common platforms for interactive products:

- Desktop software
- Websites and web applications
- Mobile devices such as phones, tablets, and digital cameras

- Kiosks

- In-vehicle systems

- Home entertainment systems such as game consoles, TV set-top boxes, and stereo/home theater systems

- Professional devices such as medical and scientific instruments

Looking at this list, you may notice that "platform" is not a precisely defined concept. Rather, it is shorthand used to describe a number of important product features, such as the physical form, display size and resolution, input methods, network connectivity, operating system, and database capabilities.

Each of these factors has a significant impact on how the product can be designed, built, and used. Choosing the right platform is a balancing act. You must find the sweet spot that best supports the needs and context of your personas but also fits within the business constraints, objectives, and technological capabilities of your company or client.

In many organizations, platform decisions, particularly those regarding hardware, unfortunately are still made well in advance of the interaction designer's involvement. It is important to inform management that platform choices will be much more effective if they are made after interaction designers complete their work.

DESIGN PRINCIPLE | *Decisions about technical platforms are best made in concert with interaction design efforts.*

Product Postures

Most people have a predominant behavioral stance that fits their working role on the job. The soldier is wary and alert; the toll collector is bored and disinterested; the actor is flamboyant and larger than life; the service representative is upbeat and helpful. Products, too, have a predominant manner of presenting themselves to users.

Platform and posture are closely related: Different hardware platforms are conducive to different behavioral stances. A social networking application running on a smartphone clearly must accommodate a different kind of user attention and level of interaction than, say, a page layout application running on a large-display desktop computer.

Software applications may be bold or timid, colorful or drab, but they should be that way for a specific set of goal-directed reasons. The presentation of an application affects how users relate to it, and this relationship influences the product's usability (and

desirability). Apps whose appearance and behavior conflict with their purpose will seem jarring or inappropriate.

A product's look and behavior should also reflect how it is used, rather than the personal taste of designers, developers, or stakeholders. From the perspective of posture, look and feel is *not* solely a brand or an aesthetic choice; it is a behavioral choice. Your application's posture is part of its behavioral foundation, and whatever aesthetic choices you make should be in harmony with this posture.

The posture of your interface dictates many important guidelines for the rest of the design, but posture is not monolithic: Just as you present yourself to others in somewhat different ways depending on the social context, some products may exhibit characteristics of a number of different postures in different usage contexts. For instance, when reading e-mail on a smartphone during a train ride, the user may devote concentrated attention to interactions with the device (and expect a commensurate experience). However, the same user will have significantly less attention to devote if she is using it to look up an address while running to a meeting.

Similarly, while a word processor is optimized for concentrated, devoted, and frequent user attention, some of its tools, like the table construction tool, are used only briefly and infrequently. In cases like this, it is worthwhile both to define the predominant posture for the product as a whole and to consider the posture of individual features and usage contexts.

In the remainder of this chapter we discuss appropriate postures and other design considerations for several key platforms, including desktop software, websites and web applications, mobile handheld and tablet devices, and other embedded devices.

Postures for the Desktop

We use the term "the desktop" as a catchall phrase referring to applications that run on modern desktop or laptop computers. Generally speaking, interaction design has its roots in desktop software. While historically, designers have grappled with issues related to complex behaviors on a variety of technical platforms, desktop and laptop computers have brought these complex behaviors into the mainstream. In more recent history, web applications, mobile devices, and digital appliances of many kinds have expanded both the repertoire of behaviors and their mainstream reach; we'll talk more about these later in this chapter.

Desktop applications express themselves in three categories of posture: sovereign, transient, and daemonic. Because each describes a different set of behavioral attributes, each also describes a different type of user interaction. More importantly, these categories

give the designer a point of departure for designing an interface. A sovereign-posture application, for example, won't feel right unless it behaves in a "sovereign" way.

Sovereign posture

Applications that monopolize users' attention for long, continuous periods of time are called *sovereign-posture* applications. Sovereign applications offer a large set of related functions and features, and users tend to keep them up and running continuously, occupying the full screen. Good examples of this type of application are word processors, spreadsheets, and e-mail applications. Many job-specific applications are also sovereign applications because they are often deployed on the screen for long periods of time, and interaction with them can be deep and complex.

Users working with sovereign applications often find themselves in a state of flow. Sovereign applications are usually used *maximized* (we'll talk more about window states in Chapter 18). For example, it is hard to imagine using Microsoft Outlook in a 3-by-4-inch window. That size is inconvenient for Outlook's main job: allowing you to create and view e-mail and appointments (see Figure 9-1). A sovereign product dominates the user's workflow as his primary tool.
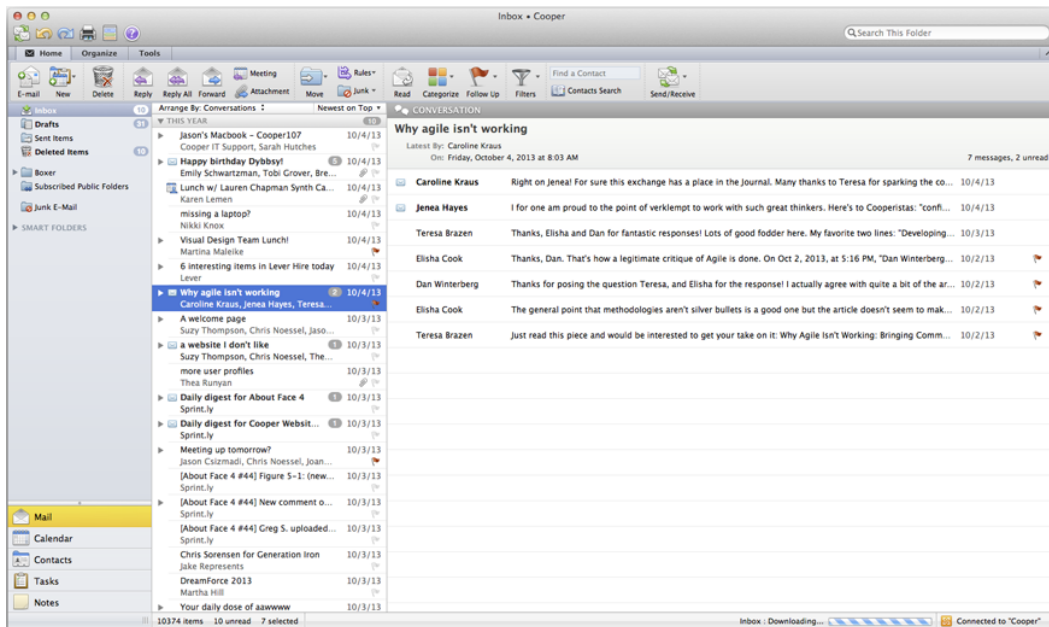
Figure 9-1: Microsoft Outlook is a classic example of a sovereign-posture application. It stays onscreen, interacting with the user for long, uninterrupted periods, and with its multiple adjacent panes for navigation and supporting information, it begs to take up the full screen.

Target intermediate users

Because people typically devote time and attention to using sovereign applications, they often have a vested interest in progressing up the learning curve to become intermediate users, which we'll discuss in detail in Chapter 11. Each user spends time as a novice, but only a short period of time *relative to the amount of time he will eventually spend* using the product. Certainly a new user has to get over the initial learning curve. But seen from the perspective of the entire relationship between the user and the product, the time he spends getting acquainted with the application is small.

From the designer's point of view, this often means that the application should be optimized for use by intermediates and not be aimed primarily at beginners (or experts). Sacrificing speed and power in favor of a clumsier but easier-to-learn idiom is out of place here, as is providing only sophisticated power tools. Of course, if you can offer easier or more powerful idioms without compromising the interaction for intermediate users, that is often best. In any case, the sort of user you're optimizing for is determined by your choice of primary persona and your understanding of his or her attitudes, aptitudes, and use contexts.

Between first-time users and intermediate users, many people use sovereign applications only occasionally. These infrequent users cannot be ignored. However, the success of a sovereign application is still dependent on its intermediate, frequent users until someone else satisfies both them *and* inexperienced users. WordStar, an early word processing application, is a good example. It dominated the word processing marketplace in the late '70s and early '80s because it served its intermediate users exceedingly well, even though it was extremely difficult for infrequent and first-time users. WordStar Corporation thrived until its competition offered the same power for intermediate users while simultaneously making the application much less painful for infrequent users. WordStar, unable to keep up with the competition, rapidly dwindled to insignificance.

Be generous with screen real estate

Because the user's interaction with a sovereign application dominates his session at the computer, the application shouldn't be afraid to take up as much screen real estate as possible. No other application will be competing with yours (beyond the occasional transient notification and communication apps), so don't waste space, but also don't be shy about taking what you need to do the job. If you need four toolbars to cover the bases, use four toolbars. In an application of a different posture, four toolbars may be overly complex, but the sovereign posture has a defensible claim on the pixels.

In most instances, sovereign applications run maximized. In the absence of explicit instructions from the user, your sovereign application should default to maximized or full-screen presentation. The application needs to be fully resizable and must work well in other screen configurations, but the interface should be optimized for full-screen use, instead of the less common cases.

> **DESIGN PRINCIPLE**
>
> *Optimize sovereign applications for full-screen use.*

Use a minimal visual style

Because users will stare at a sovereign application for long periods, you should take care to mute the colors and texture of the visual presentation. Keep the color palette narrow and conservative. Big colorful controls may look really cool to newcomers, but they seem garish after a couple of weeks of daily use. Tiny dots or accents of color will have more effect in the long run than big splashes, and they enable you to pack controls and information more tightly than you otherwise could.

> **DESIGN PRINCIPLE**
>
> *Sovereign interfaces should feature a conservative visual style.*

The user will stare at the same palettes, menus, and toolbars for many hours, gaining an innate sense of where things are from sheer familiarity. This gives you, the designer, freedom to do more with fewer pixels. Toolbars and their controls can be smaller than normal. Auxiliary controls such as screen splitters, rulers, and scrollbars can be smaller and more closely spaced.

Provide rich visual feedback

Sovereign applications are a great platform for creating an environment rich in visual feedback for users. You can productively add extra bits of information to the interface. A status bar at the bottom of the screen, the ends of the space normally occupied by scrollbars, the title bar, and other dusty corners of the product's visible components can be filled with visual indications of the application's status, the data's status, the system's state, and hints for more productive user actions. However, while enriching the visual feedback, you must be careful not to create an interface that is hopelessly cluttered.

The first-time user won't even notice such artifacts, let alone understand them, because of the subtle way they are shown on the screen. After a period of steady use, however, he will begin to see them, wonder about their meaning, and experimentally explore them. At this point, the user will be willing to expend a little effort to learn more. If you provide an easy means for him to find out what the artifacts are, he will become not only a better user but also a more satisfied user. His power over the application will grow with his understanding. Adding such richness to the interface is like adding a variety of ingredients to soup stock—it enhances the entire meal. We discuss this idea of *rich visual modeless feedback* in Chapter 15.

Support rich input

Sovereign applications similarly benefit from rich input. Every frequently used aspect of the application should be controllable in several ways. Direct manipulation, keyboard mnemonics, and keyboard accelerators are all appropriate (see Chapter 18). You can make more aggressive demands on the user's fine motor skills with direct-manipulation idioms. Sensitive areas on the screen can be just a couple of pixels across, because you can assume that the user is established comfortably in his chair, arm positioned in a stable way on his desk, rolling the mouse firmly across a resilient mouse pad.

> **DESIGN PRINCIPLE** *Sovereign applications should exploit rich input.*

Go ahead and use the corners and edges of the application's window for controls. In a jet cockpit, the most frequently used controls are situated directly in front of the pilot. Those needed only occasionally or in an emergency are found on the armrests, overhead, and side panels. In Word for Mac, Microsoft has put the most frequently used functions at the top of the window, as shown in Figure 9-2. Microsoft also put visually dislocating functions on small controls near the bottom of the screen. These controls change the appearance of the entire visual display—Draft View, Outline View, Publishing layout View, Print Layout View, Notebook Layout View, and Focus View. Neophytes do not often use them and, if accidentally triggered, they can be confusing. Placing them near the bottom of the screen makes them almost invisible to new users. Their segregated positioning subtly and silently indicates that they should be used with caution. More experienced users, with more confidence in their understanding of and control over the application, will begin to notice these controls and wonder about their purpose. They can experimentally select them when they feel fully prepared for the consequences. This is an accurate and useful mapping of control placement to usage.
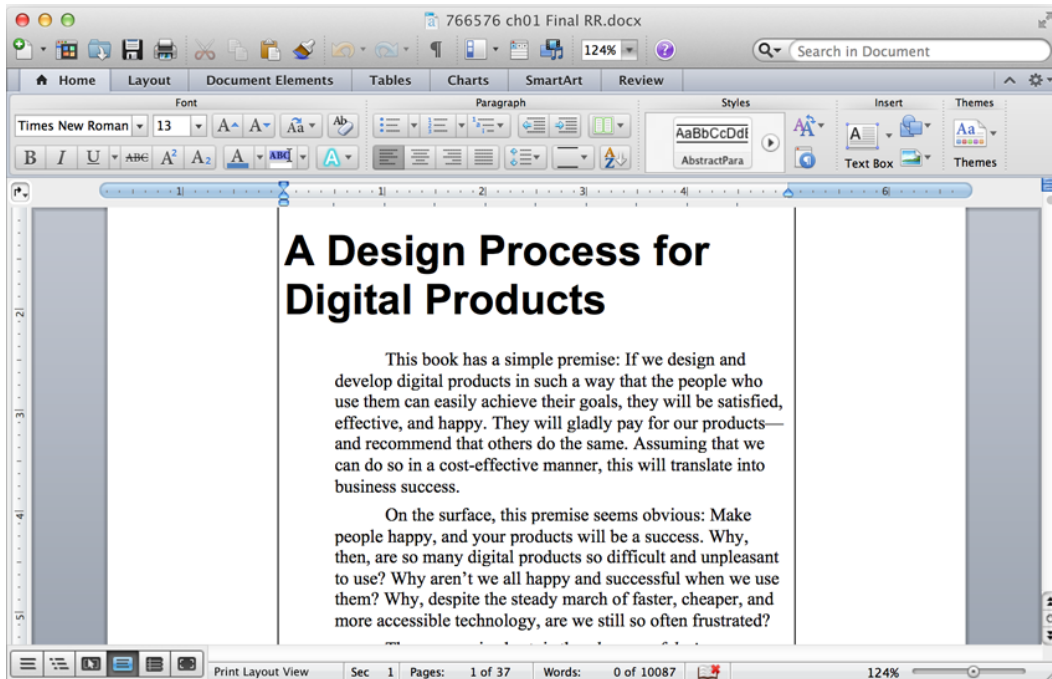
Figure 9-2: Microsoft Word has placed controls at both the top and bottom of the application. The controls at the bottom are used to change views and are appropriately segregated because they can cause significant visual dislocation.

Design for documents

The dictum that sovereign applications should fill the screen is also true of document windows within the application itself. Child windows containing documents should always be maximized inside the application unless the user explicitly instructs otherwise, or the user needs to simultaneously work in several documents to accomplish a specific task.

> **DESIGN PRINCIPLE**
>
> *Maximize document views within sovereign applications.*

Many sovereign applications are also document-centric. In other words, their primary functions involve creating and viewing documents containing rich data. This makes it easy to believe that the two are always correlated, but this is not the case. If an application

manipulates a document but performs only a single, simple function, such as scanning an image, it isn't a sovereign application and shouldn't exhibit sovereign behavior. Such single-function applications have a posture of their own—the transient posture.

Transient posture

A product with a *transient posture* comes and goes, presenting a single function with a constrained set of accompanying controls. The application is invoked when needed, appears, performs its job, and then quickly leaves, letting the user continue her normal activities, usually with a sovereign application.

The defining characteristic of a transient application is its temporary nature. Because it doesn't stay on the screen for extended periods of time, users don't get the chance to become very familiar with it. Consequently, the product's user interface should be obvious and helpful, presenting its controls clearly and boldly with no possibility of confusion or mistakes. The interface must spell out what it does. This is not the place for artistic-but-ambiguous images and icons. It *is* the place for big buttons with precise legends spelled out in a large, easy-to-read typeface.

> **DESIGN PRINCIPLE**
>
> *Transient applications must be simple, clear, and to the point.*

Although a transient application can certainly operate alone on your desktop, it usually acts in a supporting role to a sovereign application. For example, calling up Files Explorer in Windows to locate and open a file while editing another with Word is a typical transient scenario. So is setting your speaker volume. Because the transient application borrows space at the expense of the sovereign, it must respect the sovereign by not taking up more space onscreen than is absolutely necessary.

In cases when the entire computer system is fulfilling a transient role in the real world, it is not necessarily appropriate to minimize the use of pixels and visual attention. Examples of this include process monitors in a fabrication environment, or digital imaging systems in an operating room. Here, the entire computer screen is referred to in a transient manner, while the user is engaged in a sovereign mechanical activity. In these cases, it is still critical for information to be obvious and easily understood from across the room. This clearly requires a bolder use of color and a more generous allotment of real estate, as shown in Figure 9-3.

Figure 9-3: OS X Dashboard widgets and the iTunes Miniplayer are good examples of transient applications. They are referred to or interacted with briefly before the user's attention turns to an activity in a sovereign application. The use of rich dimensional rendering gives them an appropriate amount of visual gravity.

Make it bright and clear

Although a transient application must conserve the total amount of screen real estate it consumes, the controls on its surface can be proportionally larger than those on a sovereign application. More forceful visual design on a sovereign application would pall within a few weeks, but the transient application isn't onscreen long enough for it to bother the user. On the contrary, bolder graphics help users to orient themselves more quickly when the application pops up.

Transient applications should have instructions built into their surface. The user may see the application only once a month and will likely forget the meanings and implications of the choices presented. Instead of having a button labeled "Setup," it's better to make the button large enough so that it can be labeled "Set up user preferences." The verb/object construction results in a more easily comprehensible interface, and the results of clicking the button are more predictable. Likewise, nothing should be abbreviated on a transient application, and feedback should be direct and explicit to avoid confusion. For example, the user should be easily able to understand that the printer is busy, or that a piece of recently recorded audio is 5 seconds long.

Keep it simple

After the user summons a transient application, all the information and facilities he needs should be right there on the surface of the application's single window. Keep the user's attention on that window. Don't force him into supporting subwindows or dialog boxes to take care of the application's main function. If you find yourself adding a dialog box or second view to a transient application, that's a key sign that your design needs a review.

> **DESIGN PRINCIPLE**
>
> *Transient applications should be limited to a single window and view.*

Transient applications are not the place for tiny scrollbars and fussy mouse interactions. Keep demands on the user's fine motor skills to a minimum. Simple pushbuttons for simple functions are good. Direct manipulation can also be effective, but anything that can be directly manipulated must be discoverable and big enough to interact with easily. You can provide keyboard shortcuts, but they must be simple, and all important functions should also be visible on the interface.

Of course, rare exceptions to the monothematic nature of transient applications sometimes occur. If a transient application performs more than just a single function, the interface should communicate this visually and unambiguously and provide immediate access to all functions without the addition of windows or dialogs. One such application is the Art Directors Toolkit by Code Line Communications, shown in Figure 9-4. It performs a number of different calculator-like functions useful to users of graphic design applications.

Keep in mind that a transient application will likely be called on to help manage some aspect of a sovereign application (as shown in Figure 9-4). This means that the transient application, as it is positioned on top of the sovereign, may obscure the very information that it is chartered to work on. This implies that the transient application must be movable, which means it must have a title bar or other obvious affordance for dragging.
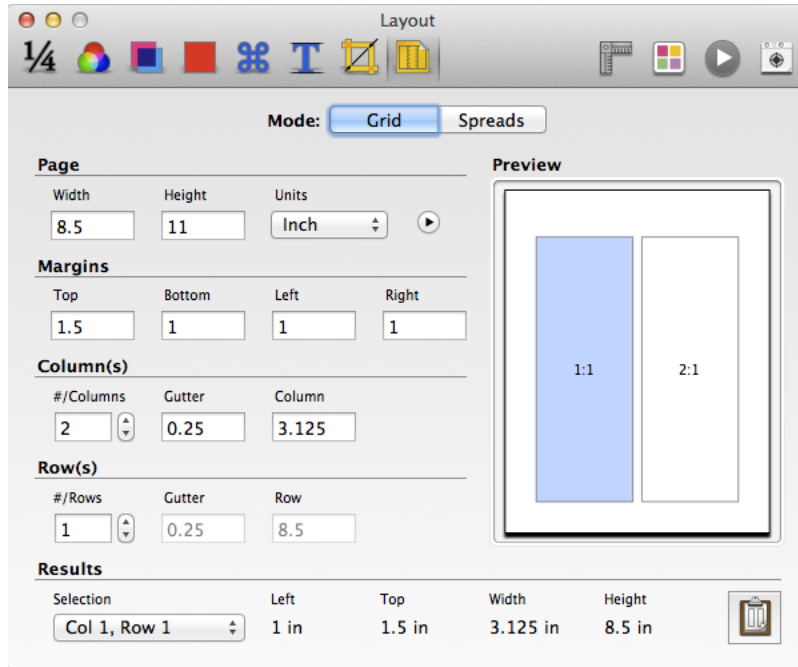
Figure 9-4: Art Directors Toolkit by Code Line Communications is a transient application. It provides a number of discrete functions such as calculating dimensions of a layout grid. These functions are designed to support the use of a sovereign layout application such as Adobe InDesign. The many functions are organized into tabs and are directly accessible at all times.

It is vital to keep the amount of management overhead as low as possible with transient applications. All the user wants to do is perform a specific function or get a certain piece of information, and then move on. It is unreasonable to force the user to add nonproductive window-management tasks to this interaction.

Remember user choices

The most appropriate way to help users with both transient and sovereign apps is to give the applications a memory. If a transient application remembers where it was the last time it was used, that same size and placement probably will be appropriate next time, too. These settings will almost always be more suitable than any default setting might happen to be. Whatever shape and position the user left the application in should be the shape and position the application reappears in when it is next summoned. Of course, this holds true for its logical settings, too.

No doubt you have already realized that most dialog boxes behave very much like transient applications. Therefore, the preceding guidelines for transient applications apply equally well to the design of dialog boxes. (Chapter 21 covers dialogs in more detail.)

Daemonic posture

Applications that normally do not interact with the user are *daemonic-posture* applications. These applications serve quietly and invisibly in the background, performing vital tasks without the need for human intervention. A printer driver and network connection are excellent examples.

As you might expect, any discussion of the user interface of daemonic applications is necessarily short. Whereas a transient application controls the execution of a function, daemonic applications usually manage processes. Your heartbeat isn't a function that must be consciously controlled; rather, it is a process that proceeds autonomously in the background. Like the processes that regulate your heartbeat, daemonic applications generally remain invisible, competently performing their process as long as your computer is turned on. Unlike your heart, however, daemonic applications must occasionally be installed and removed, and, also occasionally, they must be adjusted to deal with changing circumstances. It is at these times that a daemon talks to the user (or vice versa). Without exception, the interaction between the user and a daemonic application is transient in nature, and all the imperatives of transient application design hold true here also.

The principles of transient design that are concerned with keeping users informed of an application's purpose and of the scope and meaning of the available choices become even more critical with daemonic applications. In many cases, users are unaware of the existence of the daemonic application. If you recognize that, it becomes obvious that reports about status from that application can be disconcerting if not presented with great attention to clarity. Because many of these applications perform esoteric functions—such as printer drivers—the messages from them must not confuse users or lead to misunderstandings.

A question that is often taken for granted with applications of other postures becomes very significant with daemonic applications: If the application normally is invisible, how should the user interface be summoned on those rare occasions when it is needed? Windows 8 Desktop represents these daemons with icons on the right side of the taskbar. (OS X does something similar on the right side of the menu bar.) Putting icons onscreen

when they are almost never needed leads to useless visual clutter. Daemonic icons should be employed persistently only if they provide continuous and useful status information. Microsoft solved this problem by hiding daemonic icons that are not actively being used to report status or access functionality in a pop-up menu, as shown in Figure 9-5.

Figure 9-5: The status area of the taskbar in Windows 8. The speaker icon provides modeless visual status information, because the icon changes if the speaker's volume is low or muted. Hovering over the icon provides more information and clicking or right-clicking it provides access to the volume and other audio controls. To the right of the speaker icon, the Dropbox icon modelessly indicates that Dropbox is automatically syncing its desktop folder.

Both Mac OS and Windows also employ *control panels* as an effective approach to configure daemonic applications. These user-activated transient applications give users a consistent place to go to configure daemons. It is also important to provide direct, inline access to daemonic applications anytime an issue with them prevents someone from accomplishing what he aims to. (Of course, the standard disclaimer applies: Don't interrupt users unnecessarily.) For example, if a taskbar icon indicates a problem with a printer, clicking that icon should provide a mechanism to troubleshoot and rectify the problem.

Postures for the Web

The advent of the World Wide Web was initially both a boon and a curse for interaction designers. For perhaps the first time since the invention of graphical user interfaces, corporate decision makers began to understand and adopt the language of user-centered design. On the other hand, the limitations and challenges of web interactivity, which were the natural results of its historical evolution, set *back* interaction design by nearly a decade. However, since the publication of the third edition of this book, the web has become a much friendlier place for the kind of rich interactions (such as drag and drop or gestures) that were long possible only in native desktop applications.

Today's websites can be grouped into three basic categories that in a way recapitulate the development of web technology: *informational websites*, *transactional websites*, and *web applications*. Each of these types has its own postural considerations. As with many of the categorizations we offer in this book, the lines between them can be indistinct.

Think of them as representing a spectrum on which any website or web application can be located.

Informational website posture

Web browsers were originally conceived of as a way to view shared and linked documents without the need for cumbersome data transfer utilities like File Transfer Protocol (FTP), Gopher, and Archie. The early web was made up of sequential or hierarchical sets of these documents (*web pages*), collectively called *websites*. In essence, these were places for users to go to get information. In this book we call these *informational websites* to distinguish them from the more interactive web-delivered services that came later. From an interaction standpoint, informational websites consist of a navigation model to take users from one page to another, as well as a search facility to provide more goal-directed location of specific pages.

Although informational websites hark back to the early web of the 1990s, plenty of them still exist, in the form of personal sites, corporate marketing and support sites, and information-centric intranets. Wikipedia is the number 5 site in the world, and is an informational website. In such sites, the dominant design concerns are the visual look and feel, layout, navigational elements, and site structure (information architecture). *Findability*, a term coined by Peter Morville, is an apt way to describe the biggest design issue for informational websites in a nutshell: the ease of finding specific information held within them.

Balancing sovereign and transient

Sites that are purely informational, that require no complex transactions beyond navigating from page to page and limited searching, must balance two forces: the need to display a reasonable density of useful information, and the need to allow first-time and infrequent users to learn and navigate the site easily. This implies a tension between sovereign and transient attributes in informational sites. Which stance is more dominant depends largely on who the target personas are and what their behavior patterns are when they use the site: Are they infrequent or one-time users, or are they repeat users who will return weekly or daily to view content?

The frequency with which content is updated on a site influences this behavior in some respects: Informational sites with real-time information naturally will attract repeat users more than a site updated once a month. Infrequently updated sites may be used more for occasional reference (assuming that the information is not too topical) rather than for heavy repeat use and thus should be given more of a transient stance than a sovereign one. What's more, the site can configure itself into a more sovereign posture by paying attention to how often a particular user visits.

Sovereign attributes

Detailed information display is best accomplished by assuming a sovereign stance. By assuming full-screen use, designers can take advantage of all the space available to clearly present the information as well as navigational tools and wayfinding cues to keep users oriented.

The only fly in the ointment of sovereign stance for the web is choosing which full-screen resolution is appropriate. (To a lesser degree, this is an issue for desktop applications, although it is easier for creators of desktop software to dictate the appropriate display.) Web designers need to decide early on what resolution they will optimize the screen designs for. You can use a "responsive" approach to flexibly display content in a variety of browser window sizes, with interfaces that can even scale smoothly between the many variants of mobile and desktop screen size. However, your designs should be optimized for the most common display sizes used by your primary (and sometimes secondary) persona. Quantitative research is helpful in determining this: Among people similar to your personas, how many are still using 800 × 600-pixel displays?

Transient attributes

The less frequently your primary personas access the site, the more transient a stance the site needs to take. In an informational site, this manifests itself in terms of ease and clarity of navigation and orientation.

Because users might bookmark sites that they use infrequently for reference, you should make it possible for them to bookmark any page of information so that they can reliably return to it later.

Users likely will visit sites that are updated weekly to monthly only intermittently, so navigation on such sites must be particularly clear. It's beneficial if these sites can retain information about past user actions via cookies or server-side methods and present information that is organized based on what interested the user previously. This can help less-frequent users find what they need with minimal navigation. (This assumes that the user is likely to return to the same content on each visit to the site.)

Mobile web access may also point toward a transient posture. Mobile users are likely multitasking and have limited time and cognitive resources to get the information they seek. Mobile versions of your site need to streamline navigation and eliminate verbiage, allowing users to rapidly find what they are looking for. Responsive techniques allow a website to be rendered for desktop or handheld screens, but you must take great care with the navigation and information flow.

Transactional website posture

More and more websites go beyond simple clicking and searching to offer *transactional* functionality that allows users to accomplish something beyond acquiring information. Classic examples of *transactional websites* are online storefronts and financial services sites, as shown in Figure 9-6.
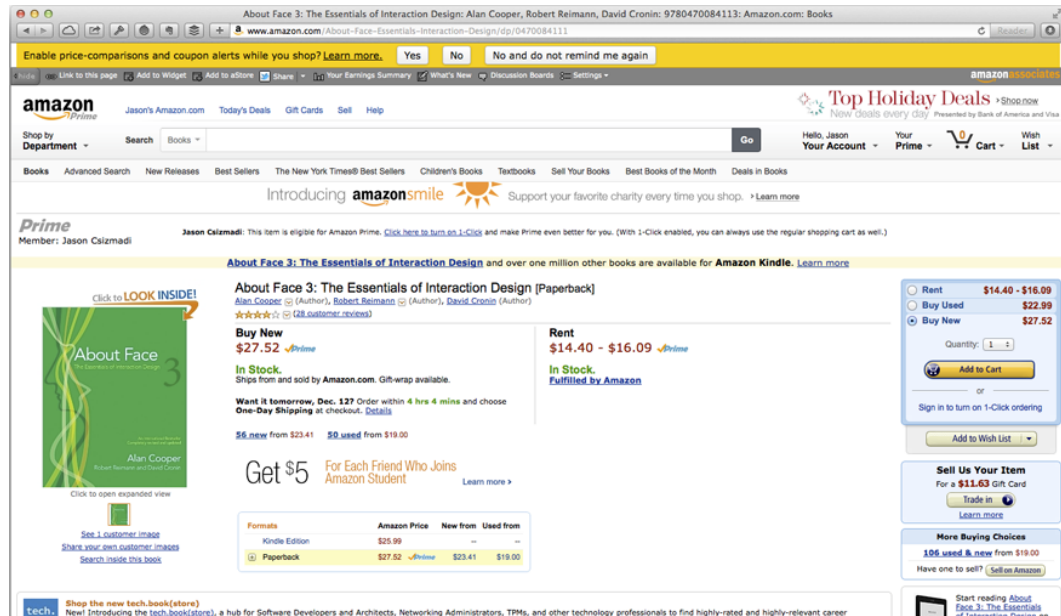
Figure 9-6: Amazon is the classic example of a transactional e-commerce website. It was one of the first, and most successful, of its kind.

These typically are structured in a hierarchical page-based manner, similar to an informational website, but in addition to informational content, the pages contain functional elements with complex behaviors. In the case of the online store, these functional elements include the shopping cart, the checkout features, and the ability to save a user profile. Some shopping sites have more sophisticated and interactive tools as well, such as "configurators," which allow users to customize or choose options related to their purchases.

Transactional sites must, like informational sites, strike a balance between sovereign and transient stances. In fact, many transactional sites have a significant informational aspect. For example, online shoppers like to research and compare products or investments. During these activities, users are likely to devote significant attention to a single site. But in some cases, such as comparison shopping, they are also likely to bounce among several sites. For these types of sites, navigational clarity is very important, as are access to supporting information and efficient transactions.

Search engines like Google search and Bing are a special kind of transactional site designed to provide navigation to other websites, as well as access to aggregated news and information from a variety of sources. Performing a search and navigating to resulting sites is a transient activity, but the information aggregation aspects of a portal like Yahoo! sometimes require a more sovereign stance.

The transient aspects of users' experiences with transactional sites make it especially important that users not be forced to navigate more than necessary. It may be tempting to break information and functions into several pages to reduce load time and visual complexity (both of which are good objectives). But also consider the potential for confusion and click fatigue on the part of your audience. Jared Spool's usability firm, User Interface Engineering, conducted a landmark usability study in 2001 of user perception of page load times for e-commerce sites. The results confirmed that user perception of load time is more closely correlated to *whether the user can achieve his or her goals* than to actual page load times.[1]

Designing transactional websites requires attention to information architecture for content and page organization and attention to interaction design for the creation of appropriate behaviors for the more functional elements. Visual design must serve both of these ends. It also must effectively communicate key brand attributes. This is often particularly important considering the commercial nature of most transactional sites.

Web application posture

Web applications are heavily interactive and exhibit complex behaviors in much the same way that robust desktop applications do. While some web applications maintain a page-based navigation model, these pages are more analogous to views than they are to web documents. A few of these applications are still bound by the archaic server query/response model, which requires users to manually "submit" each state change. However, technology now supports robust asynchronous communication with the server and local data caching. This allows an application delivered through a browser to behave in much the same way as a networked desktop application.

Here are some examples of web applications:

- Enterprise software, ranging from old-school SAP interfaces duplicated in a browser to contemporary collaborative tools such as Salesforce.com and 37signals' Basecamp
- Personal publishing and sharing tools, including blogging software such as WordPress, photo-sharing software such as Flickr, and, cloud storage such as Dropbox
- Productivity tools such as Zoho Docs and the Google Docs suite
- Social software, such as Facebook and Google+
- Web-based streaming media apps such as Hulu, Pandora, and Rdio

Web applications like these are presented to users very much like desktop applications that happen to run inside a browser window. There's little penalty for this, as long as the interactions are carefully designed to reflect the technology constraints (since even rich web interactions still don't always match the capabilities of desktop apps). These applications can act as replacements for sovereign desktop apps, but they also can be employed for infrequently used functionality for which the user may not want to go to the trouble of installing a dedicated executable.

It can be challenging to design and deliver sophisticated interactions that work across a number of different browsers and browser versions. Nonetheless, the web platform is an excellent means of delivering tools that enable and facilitate collaboration. In addition, it can be of significant value to allow people to effortlessly access the same data and functionality *from the cloud*, one of the core strengths of web applications.

Sovereign web applications

Web applications, much like desktop applications, can have sovereign or transient posture. But because we use the term to refer to products with complex and sophisticated functionality, by definition they tend toward sovereign posture.

Sovereign web applications strive to deliver information and functionality in a manner that best supports more-complex human activities. Often this requires a rich and interactive user interface. A good example of such a web application is Proto.io, shown in Figure 9-7. This online interactive prototyping service offers tasks such as drag-and-drop assembly of prototypes using a library of interactive objects and behavior specification tools, in-place editing for text labels, and other direct manipulation tools. Other examples of sovereign web applications include enterprise software and engineering tools, such as Jira, delivered through a browser.

Unlike page-oriented informational and transactional websites, the design of sovereign web applications is best approached in the same manner as desktop applications. Designers also need a clear understanding of the medium's technical limitations and what the development organization can reasonably accomplish on time and within budget. Like sovereign desktop applications, most sovereign web applications should be full-screen applications, densely populated with controls and data objects. They also should make use of specialized panes or other screen regions to group related functions and objects. Users should have the feeling that they are in an environment, not that they are navigating from page to page or place to place. Redrawing and re-rendering information should be minimized (as opposed to the behavior on websites, where almost any action requires a full redraw).
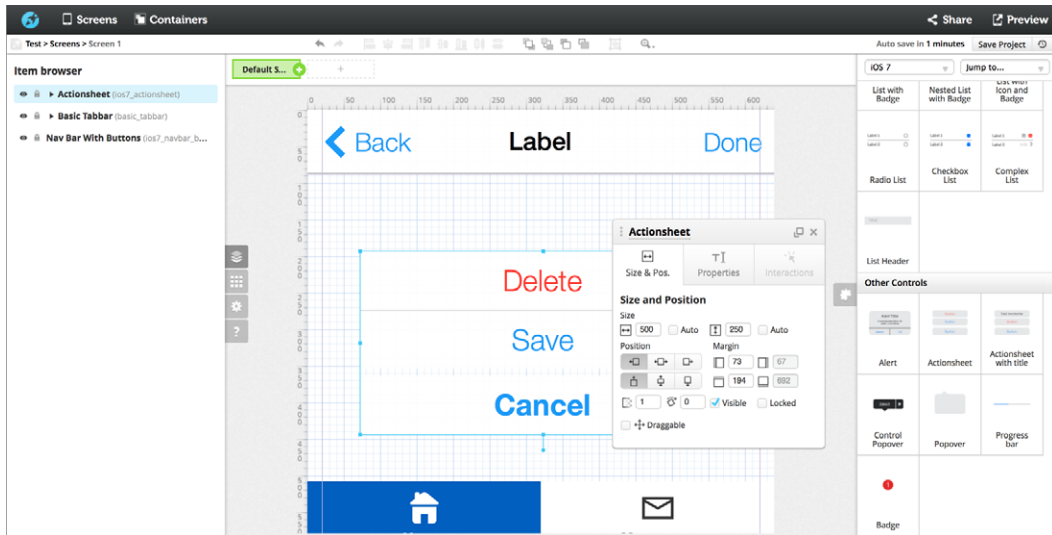
Figure 9-7: Proto.io's web-based interactive prototyping environment is as rich and refined as many desktop authoring environments, featuring drag-and-drop assembly and direct manipulation of all interactive objects.

Treating sovereign web applications as desktop applications rather than as collections of web pages has a benefit. It allows designers to break out of the constraints of page-oriented models of browser interaction to address the complex behaviors that these client-server applications require. Websites are effective places to get information you need, just as elevators are effective places to get to a particular floor in a building. But you don't try to do actual work in elevators. Similarly, users are not served by being forced to attempt to do real, interaction-rich transactional work using page-based websites accessed through a browser.

Transient web applications

Delivering enterprise functionality through a browser-based user interface has one particular advantage. If done correctly, it can give users better access to occasionally used information and functionality without requiring them to install every tool they may need on their computers. Whether it is a routine task that is performed only once a year to file taxes or the occasional generation of an ad hoc report, *transient web applications* aim to accomplish just this.

When designing transient web applications, as with all transient applications, it is critical to provide clear orientation and navigation. Also keep in mind that one user's transient application may be another user's sovereign application. Think hard about how compatible the two users' needs are. An enterprise web application often serves a wide range of personas and requires multiple user interfaces accessing the same set of information.

Postures for Mobile Devices

Since the publication of the third edition of *About Face*, a sea change in the world of personal computing has occurred. New and exceptionally powerful mobile devices with high-resolution displays and capacitive multi-touch input technology have become the mainstream platform of choice, converging functionality and integrating into people's lives like no interactive devices before them. Constraints of form factor, new and gestural forms of input, and dynamic, on-the-go use contexts all provide unique challenges for designers and unique considerations for application posture.

Smartphone and handheld posture

Handheld devices present special challenges for interaction designers. Because they are designed specifically for mobile use, these devices must be small, lightweight, economical in power consumption, ruggedly built, and easy to hold and manipulate in busy, distracting situations. Especially for handhelds, close collaboration among interaction designers, industrial designers, developers, and mechanical engineers is a real necessity. Of particular concern are size and clarity of display, ease of input and control, and sensitivity to context.

Functionally and posturally speaking, handheld devices have gone through a steep evolutionary curve in the past decade. Prior to the iPhone, handheld devices were characterized by small, low-resolution screens with input and navigational mechanisms that can be categorized as awkward at best. Even the best in class of these devices, the Palm Treo (the direct descendant of the groundbreaking Palm Pilot), suffered from a small, low-resolution, and rudimentary (by current standards) touchscreen. It also only moderately successfully integrated hardware navigation and touch input controls. Such devices also had rudimentary and cumbersome ecosystems for updating or adding applications to the device. This tended to result in their use being limited primarily to their default app suite.

However, that all changed with the introduction of the iPhone and Android smartphones, which together heralded the dawn of a new era of standalone on-the-go computing.

Satellite posture

In the early days of PDAs, media players, and phone-enabled communicators, handheld devices were best designed as *satellites* of desktop computer systems. Palm and early Windows Mobile devices both succeeded best as portable extensions of the desktop geared primarily toward accessing and viewing information and providing only lightweight input and editing features. These devices were optimized for viewing (or playing) data loaded from desktop systems, and they included a means of syncing handheld

data with desktop data. As cloud storage and services have become mainstream, these devices have replaced wired desktop syncing with wireless cloud syncing.

Satellite posture, then, emphasizes retrieving and viewing data. It uses as much of the limited screen real estate available on the device as possible to faithfully display content authored on or loaded from the desktop. Controls are limited to navigating and viewing data or documents. Some devices with satellite posture may have an onscreen keyboard, but these usually are small and are designed for brief and infrequent use.

Satellite posture is less common these days than *convergence* handheld devices. Since the advent of the iPhone and its competitors, these have become tiny, full-fledged computers in their own right. However, satellite posture is still the model for dedicated content-oriented devices such as digital cameras, highly portable dedicated e-readers like the e-ink Kindles (see Figure 9-8), and what remains of the dedicated digital audio and video player market, such as the iPod Nano. Applications on convergence devices that are focused on content navigation and/or playback may adopt what is essentially a satellite posture.
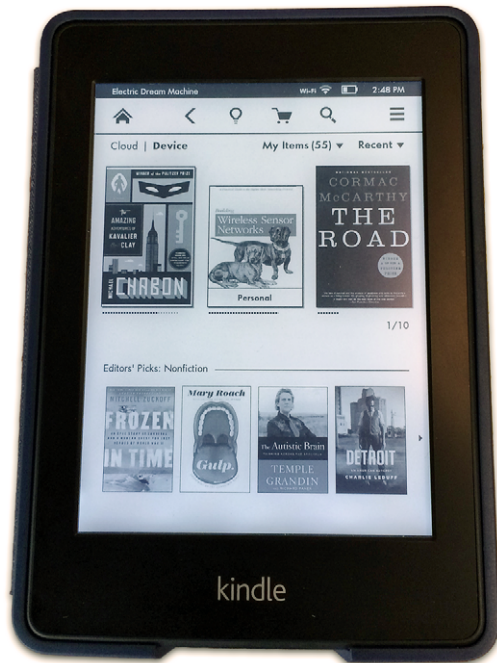
Figure 9-8: Amazon's Kindle is a good example of a satellite posture device. It is used almost exclusively to view content (e-books) that has been purchased and synced from the cloud. Previous-generation satellite posture devices relied on syncing to a desktop computer to retrieve their data. The Kindle was one of the first to provide direct syncing with a cloud service.

One new development for satellite devices is the advent of wearable computing. Wristwatch and eyeglass format devices typically pair with a standalone convergence device via Bluetooth or other wireless connections, and provide notifications and other contextual information via small touchscreens or heads-up displays and voice commands. These devices take a highly transient posture, providing just enough information and possible actions to be relevant in the moment. The Samsung Gear smart watch and Google Glass are excellent examples of this new and rapidly evolving breed of satellite posture devices (see Figure 9-9).

Figure 9-9: The new frontier of wearable computing is represented by a new generation of satellite devices, such as the Samsung Gear smart watch and Google Glass. These devices provide succinct information and the minimum set of options necessary to support activity in a completely on-the-go context.

Standalone posture

Beyond its innovations in gestural computing, the iPhone almost singlehandedly transformed cellular smartphones into handheld general-purpose computing devices. The iPhone's large, ultra-high-resolution screen with multi-touch input resulted in a new posture for handheld apps, which we'll call *standalone posture*.

Standalone posture applications share some attributes with both sovereign and transient applications. Like sovereign applications, they are full-screen and sport functions accessible via menus (often through a left or right swipe gesture) and toolbars placed along the top or bottom of the screen. Also like sovereign applications, standalone

applications can include transient, modal, dialog-like screens or pop-ups, most of which should be used to configure settings or confirm destructive actions.

Like transient applications, standalone handheld applications make relatively little use of comparatively larger controls and text, due to limitations with legibility and finger-based input on multi-touch screens. Standalone apps for handhelds, like transient apps, need to be self-explanatory. The on-the-go nature of handheld app usage means that most people will use a wide variety of apps for relatively brief sessions over any given period of time. People may bounce between e-mail, instant messaging, social media, weather, news, phone, shopping, and media playback apps over only a few hours—or even minutes.

The telephone apps in modern smartphones also behave transiently. Users place their call as quickly as possible and then abandon the interface in favor of the conversation (and, on phone carriers that support it, other apps while the call takes place). The best interface for a phone is arguably nonvisual, especially when used in an automobile. Voice activation such as that offered by Apple's Siri service or Google's Android OS is perfect for placing a call; the more transient the phone's interface, the better.

Tablet device posture

After Apple transformed the smartphone from clumsy satellite device to standalone handheld computer/convergence media device, it successfully applied the same multi-touch and high-resolution display technology to the larger page-sized tablet form factor. The larger-format (more than 9 inches) high-resolution tablets such as the iPad have more than enough real estate to support true sovereign-posture apps, though the limitations of hand inputs have their own real-estate challenges. Keynote for iPad is able to support desktop-style presentation authoring on a touchscreen tablet, as shown in Figure 9-10.

Seven-inch tablets, especially those with a 16-by-9 aspect ratio, such as the Google Nexus 7 and the Amazon Kindle Fire HD, live in an awkward dimensional space between smaller handheld form factors and larger tablets. List-oriented views are uncomfortably wide, while grid views with more than two rows or columns (depending on orientation) seem cramped. Designers should be sure not to treat 7-inch tablets like oversized phones when designing their layout.

Specific platform issues aside, tablets for the most part enforce the sovereign quality of their apps; the popular tablet operating systems permit *only* full-screen applications. These sovereign apps often have scrollable or zoomable main content views, with top, bottom, or side toolbars or palettes. They are similar to their desktop brethren in concept but are more sparse and simplified in execution, as shown in Figure 9-11.
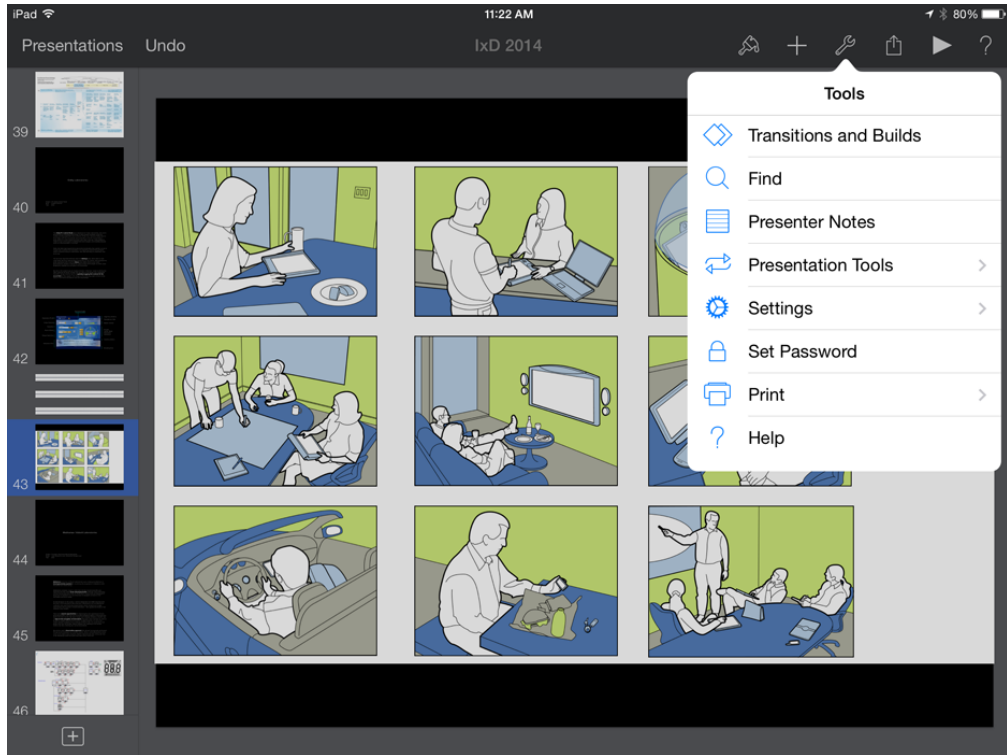
Figure 9-10: Keynote for iPad is a sovereign-posture, iOS version of Apple's presentation software for the Mac. It has functions equivalent to its desktop cousin.
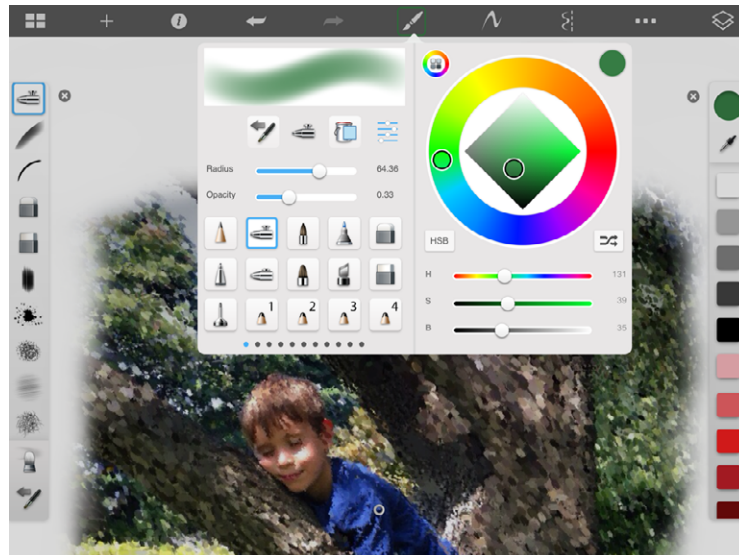
Figure 9-11: Adobe Sketchbook Pro is a drawing and painting app on the iPad. It supports a zoomable main drawing area, along with a top toolbar, and hideable tool palettes on the left and right.

Android tablets support the concept of *widgets*—transient-posture micro-apps that access the functionality of an installed sovereign app without bringing it into the foreground. Users may position these widgets on a special home screen for easy access to things like weather, stock reports, or music playback controls. Windows Surface, shown in Figure 9-12, has a similar concept called tiles. They can contain active content from an installed sovereign app, but not controls, providing similar transient-posture access to content only.

Figure 9-12: Windows Surface supports tiles containing dynamic content.

Postures for Other Platforms

Unlike software running on a computer, which has the luxury of being fairly immersive if need be, interaction design for mobile and public contexts requires special attention to creating an experience that coexists with the noise and activity of the real world happening all around the product. Kiosks and other embedded systems, such as TVs, household appliances, automobile dashboards, cameras, ATMs, and laboratory equipment, are unique platforms with their own opportunities and limitations. Without careful consideration, adding digital smarts to devices and appliances runs the risk that they will behave more like desktop computers than like the products your users expect and desire.

Kiosk posture

Kiosks are interactive systems at a specific location available for use by the public. Kiosks exist for wayfinding in malls, purchasing tickets on public transportation, checking-in at airports, self-checkout in grocery stores, and even ordering meals at some take-out restaurants. The large, full-screen nature of kiosks would appear to bias them toward sovereign posture, but there are several reasons why the situation is not quite that simple. First, users of kiosks often are first-time users (with some obvious exceptions, such as ATM users and users of ticket machines for public transport) and usually are not daily users. Second, most people do not spend a significant amount of time in front of a kiosk:

They perform a simple transaction or search, get the information they need, and move on. Third, most kiosks employ either touchscreens or bezel buttons to the side of the display, and neither of these input mechanisms supports the high data density you would expect of a sovereign application. Fourth, kiosk users rarely are comfortably seated in front of an optimally placed monitor. Instead, they stand in a public place with bright light and many distractions. These user behaviors and constraints should bias most kiosks toward transient posture, with simple navigation; large, colorful, engaging interfaces with clear affordances for controls; and clear mappings between hardware controls (if any) and their corresponding software functions. As in the design of handhelds, floating windows and dialogs should be avoided; any such information or behavior is best integrated into a single, full screen (as in sovereign-posture applications). Kiosks thus tread an interesting middle ground between the two most common desktop postures.

Because transactional kiosks often guide users through a process or set of information screen by screen, contextual orientation and navigation are more important than global navigation. Rather than helping users understand where they are *in the system*, help them understand where they are *in their process*. It's also important for transactional kiosks to provide escape hatches that allow users to cancel transactions and start over at any point.

> **DESIGN PRINCIPLE**
>
> *Kiosks should be optimized for first-time use.*

Educational and entertainment kiosks vary somewhat from the strict transient posture required of more *transactional kiosks*. In this case, exploring the kiosk environment is more important than completing single transactions or searches. In this case, more data density and more complex interactions and visual transitions sometimes can be introduced to positive effect. But the limitations of the input mechanisms need to be respected, lest the user lose the ability to successfully navigate the interface.

"Ten-foot" interface posture

Ten-foot, i.e. television and console gaming, interfaces offer an interesting posture variant. In some ways, they resemble the satellite posture of content browsing mobile touchscreen applications. For instance, as in multi-touch mobile UIs, navigation typically is both horizontal and vertical, with content options organized into grids, and with filtering and navigation options frequently available at the top or left. The primary difference is, of course, that the touchscreen's direct swipe and tap gestures are replaced with the five-way D-pad interaction of an infrared or Bluetooth remote control.

In one respect this is a big difference: It introduces the need for a current-focus item. Current focus needs to be obvious in a ten-foot UI so that the user always knows where he is and where he can go next.

The PlayStation 4 is a good example of how 10-foot UIs can use a layout similar to tablet UIs. Large buttons and simple left-right or up-down navigation, with at most 2 columns is the norm (see Figure 9-13). Seeing this screen out of context, you might believe it was from a multi-touch app.
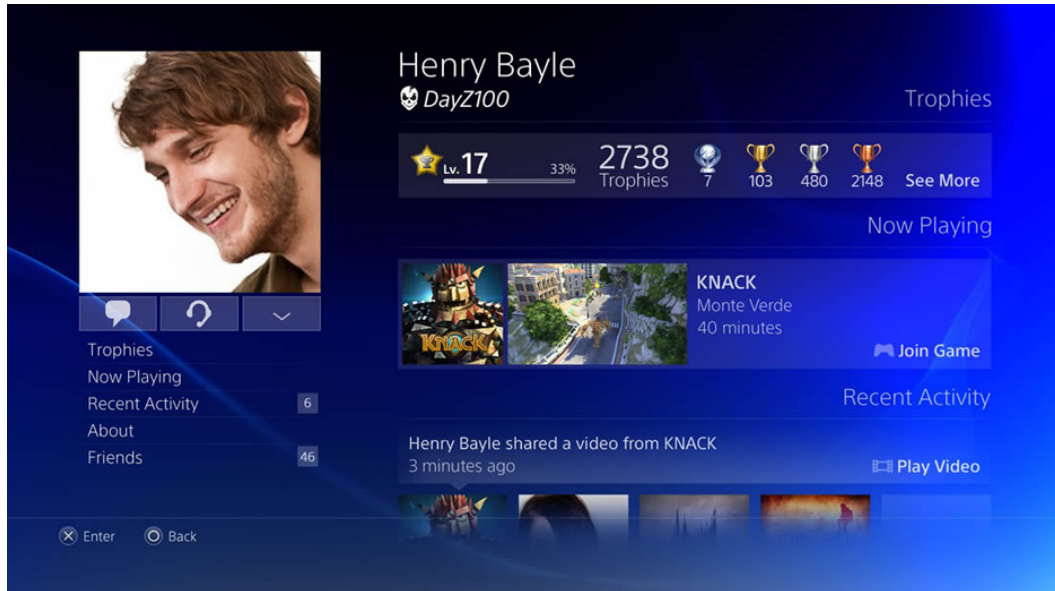
Figure 9-13: The PlayStation 4 UI bears more than a passing resemblance to a touchscreen tablet app, and for good reason. Despite the differences in input mechanism, navigation is rather similar between 10-foot UIs and many content-browsing multi-touch tablet apps.

Automotive interface posture

Automotive interfaces resemble kiosks in terms of posture. Typically they are touchscreens, but frequently they include hardware bezel buttons that encircle the screen area and thus are almost predisposed to transient interactions. Unlike kiosks, users are seated, but similar to kiosks, users typically attempt one relatively simple transaction at a time if they are also driving. Automotive interfaces, of course, have the additional constraint that they must be of minimal distraction to the driver, since keeping control of the vehicle and avoiding harm are always the primary tasks. The transaction with the system is secondary at best. At the same time, the same system should be available for focused use by a passenger if one is present.

For entertainment, HVAC control, and settings changes, automotive interfaces are transient in the way we would expect: large controls and simple screens. But navigation interfaces may take more of a sovereign stance. The interface will persist for the duration of the trip, which could be hours or even days. (Most navigation systems remember their state even if the car has been turned off to get gas or parked overnight at a hotel along the route.) Also, relatively complex information must be displayed.

Automotive navigation interfaces focus on rich, dynamic content. Map and current route information take center stage. A host of ancillary information is located in the periphery of the screen, such as the road's name, the time, the arrival time, the distance to the destination, the distance to the next turn, and the direction of the next turn. Although this kind of information hierarchy is more typical of a sovereign-posture interface, it must in the case of an automotive system be designed more like a transient UI: clear, simple, and readable at a glance.

An impressive and beautiful—but perhaps a bit worrisome—exception to the typical automotive interface is the Tesla Model S infotainment interface, shown in Figure 9-14. It sports a single 17-inch multi-touch screen with adjustable panes for simultaneous navigation, entertainment, and HVAC controls. The interface resembles a tablet's interactive posture much more than it does a kiosk's. Perhaps this is the wave of the future. If so, we hope new cars will also include active accident avoidance systems to counteract any driver distraction that might occur as a result of such large and information-rich interactive displays on the dashboard.

Figure 9-14: The Tesla Model S infotainment interface is impressive in both its size and level of interactivity. Its 17-inch multi-touch screen allows navigation, entertainment, and HVAC functions to be displayed simultaneously. This system bears more of a postural resemblance to a tablet than to a kiosk, as is more typical for automotive info systems.

Smart appliance posture

Most appliances have simple displays and rely heavily on hardware buttons and dials to manipulate the appliance's state. In some cases, however, "smart" appliances (notably, washers and dryers) most often sport color LCD touchscreens allowing rich output and direct input, as shown in Figure 9-15.

Figure 9-15: This Samsung washing machine has a well-designed color touchscreen display, with a clear and simple navigational structure.

Appliance interfaces usually are transient-posture interfaces. Users of these interfaces seldom are technology-savvy and therefore should be presented with the most simple and straightforward interface possible. These users are also accustomed to hardware controls. Unless an unprecedented ease of use can be achieved with a touchscreen, dials and buttons (with appropriate tactile, audible, and visual feedback via a view-only display or even hardware lamps) may be a better choice. Many appliance makers make the mistake of putting dozens of new—and unwanted—features into their new, digital models. Instead of making things easier, that "simple" LCD touchscreen becomes a confusing array of unworkable controls.

Another reason for a transient stance in appliance interfaces is that users of appliances need to accomplish something very specific. Like users of transactional kiosks, they are uninterested in exploring the interface or getting additional information. They simply want to put the washer on normal cycle or cook their food.

One aspect of appliance design demands a different posture. Status information indicating what cycle the washer is on or what the DVR is set to record should be presented as a daemonic icon, providing minimal status quietly in a corner. If more than minimal status is required, an auxiliary posture for this information becomes appropriate.

Give Your Apps Good Posture

In conclusion, it's important to remember that the top-level patterns of posture and platform should be among the first decisions to be made in the design of an interactive product. In our experience, many poorly designed products suffer from the failure to make these decisions consciously at any point. Rather than diving directly into the details, take a step back and consider what technical platform and behavioral posture will best meet the needs of your users and business. Also consider the possible implications of these decisions on detailed interactions.

Notes
1. Perfetti and Landesman, 2001

OPTIMIZING FOR INTERMEDIATES

Most users of technology know all too well that buying a new digital appliance or down-loading a new software app often means several days of frustration while learning a new interface. On the other hand, many experienced users of digital products may find them-selves frustrated because that product treats them like beginners. It seems impossible to find the right balance between catering to the needs of the first-timer and the needs of the expert.

One of the eternal conundrums of digital product development is how to address the needs of both beginning users and expert users with a single, coherent interface.

Left to their own devices, developers typically create interactions suitable for experts. Developers are by necessity experts on the features they build, and they tend to consider the presentation of each function in the interface as having equal weight. (From a coding and debugging standpoint, they *are* all of approximately equal weight, because they all need to operate bug-free.)

Marketing departments, on the other hand, typically demand interactions suitable only for beginners. They spend much of their time demonstrating and selling their product to people unfamiliar with it, so over time they get a biased view of user behavior and feature priority. They demand that the training wheels get bolted on. Both of these approaches lead to a frustrating experience for the majority of users, who are neither beginners nor experts.

Some developers and designers try to have it both ways, choosing to segregate these two experiences by creating wizards for beginners and burying critical functionality for experts deep in nested menus or multilayered dialog boxes. Many nonbeginners don't want to deal with the extra time and effort associated with moving step by step through a wizard each time they access a feature. But the leap from there to knowing what esoteric command to select from a set of lengthy menus is like jumping off a cliff into a shark-infested moat of implementation-model design.

What, then, is the answer? The solution to this predicament lies in a different understanding of how users master new concepts and tasks.

Perpetual Intermediates

Most users, for most of the time they are using a product, are neither beginners nor experts; instead, they are *intermediates*.

The experience level of people performing an activity tends, like most population distributions, to follow the classic statistical bell curve (see Figure 10-1). For almost any activity requiring knowledge or skill, if we graph the number of people against skill level, a relatively small number of beginners are on the left side, a few experts are on the right, and the majority—intermediate users—are in the center.

Statistics don't tell the whole story, however. The bell curve is a snapshot of many users across time. Although most intermediates tend to stay in that category, the beginners do not remain beginners for very long. The difficulty of maintaining a high level of expertise also means that experts come and go rapidly, but beginners change even more rapidly. Both beginners and experts tend over time to gravitate toward intermediacy.

Although *everybody* spends some minimum time as a beginner, *nobody* remains in that state for long. People don't like to be incompetent, and beginners, by definition, are learning to be competent. Conversely, learning and improving are rewarding, so beginners become intermediates very quickly—or they drop out. All skiers, for example, spend time as beginners, but those who find they don't rapidly progress beyond more-falling-than-skiing quickly abandon the sport. The rest soon move off the bunny slopes onto the regular runs. Only a few ever make it onto the double black diamond runs for experts.
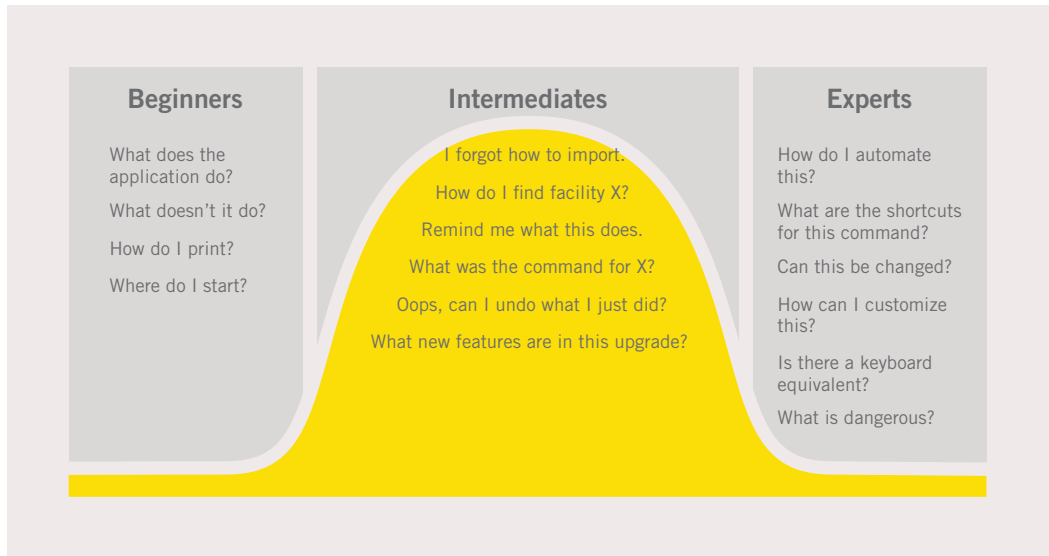
Figure 10-1: The demands that users place on digital products vary considerably with their experience.

Beginners	Intermediates	Experts
What does the application do?	I forgot how to import.	How do I automate this?
What doesn't it do?	How do I find facility X?	What are the shortcuts for this command?
How do I print?	Remind me what this does.	Can this be changed?
Where do I start?	What was the command for X?	How can I customize this?
	Oops, can I undo what I just did?	Is there a keyboard equivalent?
	What new features are in this upgrade?	What is dangerous?

Most occupants of the beginner end of the curve either migrate to the center bulge of intermediates or drop off the graph and find some product or activity in which they *can* migrate into intermediacy. Most users thus remain in a perpetual state of adequacy, striving for fluency, with their skills ebbing and flowing like the tides, depending on how frequently they use the product. Larry Constantine first identified the importance of designing for intermediates, and in his book *Software for Use* (Addison-Wesley, 1999), he calls such users *improving intermediates*. We prefer the term *perpetual intermediates*, because although beginners quickly improve to become intermediates, they seldom go on to become experts.

Most users in this middle state would like to learn more about the product but usually don't have the time. Occasionally, the opportunity or need to do so arises. Sometimes these intermediates use the product extensively for weeks at a time to complete a big project. During this time, they learn new things about the product. Their knowledge grows beyond its previous boundaries.

Other times, however, they do not use the product for months at a time and forget significant portions of what they knew. When they return to the product, they are not beginners, but they need reminders to jog their memory.

Given that most users are intermediates, how do we design products that meet their needs but don't leave beginners or advanced users out in the cold?

Inflecting the Interface

Many popular ski resorts have a gentle slope for learning and a few expert runs to really challenge the serious skier. But if the resort wants to stay in business, it will cater to the perpetual intermediate skier without scaring off the beginner or insulting the expert. The beginner must find it easy to graduate into the world of intermediacy, and the expert must not find his vertical runs obstructed by aids for cautious or conservative intermediates.

A well-balanced user interface should take the same approach. It doesn't cater to beginners or to experts, but rather devotes the bulk of its efforts to satisfying perpetual intermediates. At the same time, it provides mechanisms so that both of its smaller constituencies can be effective.We can accomplish the same in our digital products via a process of *inflection*.

Inflecting an interface means organizing it to minimize typical navigation within the interface. In practice, this means placing the most frequently desired functions and controls in the most immediate and convenient locations, such as toolbars or palettes. Less frequently used functions are pushed deeper into the interface, where users won't stumble over them. Advanced features that are less often used but have a big payoff for users can be safely tucked away in menus, dialog boxes, or drawers, where they can be accessed only when needed.

> **DESIGN PRINCIPLE**
>
> *Inflect the interface for typical navigation.*

Almost any point-and-shoot digital camera is a good example of inflection: The most commonly used function—taking a picture—is provided by a prominent hardware button that is easily accessible at a moment's notice. Less commonly used functions, such as adjusting the exposure, require interaction with menus or touchscreen controls.

Commensurate effort

The most important principle in the proper inflection of interfaces is *commensurate effort*. Although it applies to all users, it is particularly pertinent to perpetual intermediates. This principle simply states that people will willingly work harder for something that is more valuable. That value, by the way, is in the eye of the user. It has nothing to do with how technically difficult a feature is to implement, but rather is entirely related to the users' goals.

If the user really wants something, he will work harder to get it. If he needs to format beautiful documents with multiple columns, several fonts, and fancy headings to impress his boss, he will be highly motivated to explore the application's recesses to learn how. That user will put commensurate effort into the project.

But if another user wants only to print plain documents in a single column and one font, no amount of inducement will get him to learn those more advanced formatting features. Providing him with those options is unwelcome noise.

> **DESIGN PRINCIPLE**
>
> *Users make commensurate effort if the rewards justify it.*

If you add features to your application that are complex to manage, users will be willing to tolerate that complexity only if the rewards are worth it. This is why a product's user interface can't be complex for achieving simple results, but it *can* be complex for achieving *complex* results (as long as such results aren't needed very often).

Progressive disclosure

A particularly useful design pattern that exemplifies commensurate effort is *progressive disclosure*. In progressive disclosure, advanced or less frequently used controls are hidden in an expanding pane, which offers a small expand/hide toggle control to give the user access. This type of design is a boon to expert users, because the toggle is usually "sticky"; that is, once left open, it stays that way. It also gives intermediates an easy window into more advanced features but allows them to be stowed away neatly when not in use. Many of Adobe's Creative Suite tools make good use of progressive disclosure in their tool palettes, as shown in Figure 10-2.
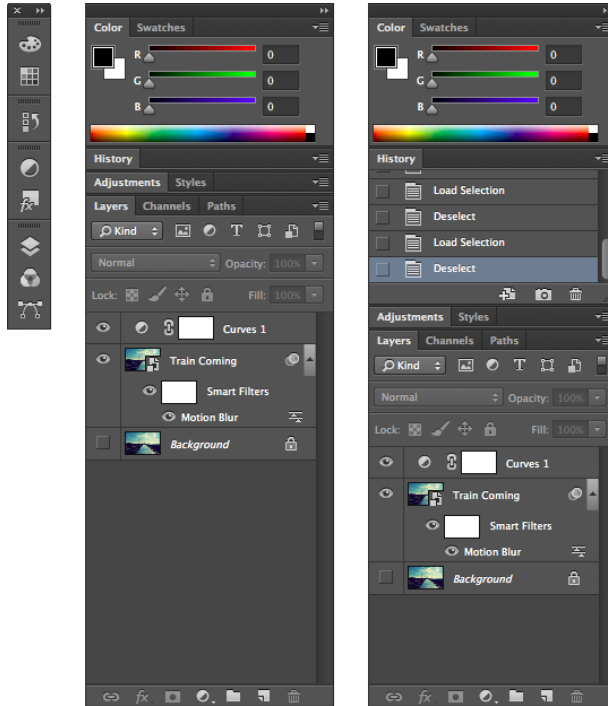
Figure 10-2: Adobe Creative Suite applications all make similar use of progressive disclosure to tame the complexity of their tool palettes for intermediates. Experts can expand them and take advantage of the sticky expansion state, which is remembered across sessions.

Organizing for inflection

In general, controls and displays should be organized in an interface according to three attributes: frequency of use, degree of dislocation, and degree of risk exposure.

- **Frequency of use** means how often the controls, functions, objects, or displays are used in typical daily patterns of use. Items and tools that are most frequently used (many times a day) should be immediately in reach. Less frequently used items, used perhaps once or twice a day, should be no more than a click or two away. Other items can be two or three clicks away. Rarely used facilities shouldn't be removed from the product if they provide real benefits to your personas, but they should be removed from the everyday work space.

- **Degree of dislocation** refers to the amount of sudden change in an interface or in the document/information being processed by the application caused by the invocation of a specific function or command. Generally speaking, it's a good idea to put these types of functions deeper into the interface.

- **Degree of risk exposure** deals with functions that are irreversible or may have other dangerous ramifications. Missiles require two humans turning keys simultaneously on opposite sides of the room to arm them. As with dislocating functions, you want to make these types of functions more difficult for your users to stumble across. The greater the ramifications, the more care you should take in exposing the function.

As users get more experienced with more complex features, they will search for shortcuts, and you should provide them. This not only helps intermediates expand their reach over time, but it's also a necessity for expert users.

Designing for Three Levels of Experience

In designing digital products, our goal should be neither to pander to beginners (since they don't stay beginners for long) nor to rush intermediates into expertise. Our approach should be threefold:

- To rapidly and painlessly move beginners into intermediacy
- To avoid putting obstacles in the way of intermediates who want to become experts
- Most of all, to keep perpetual intermediates happy as they move around the middle of the skill spectrum

We need to spend more time making our products powerful and easy to use for perpetual intermediate users. We must accommodate beginners and experts too, but not to the discomfort of the largest segment of users. The remainder of this chapter describes some basic strategies for accomplishing this.

What beginners need

Beginners are undeniably sensitive, and it is easy to demoralize a first-timer, but we must keep in mind that the state of beginnerhood is *never* an objective. Nobody wants to remain a beginner. It is merely a rite of passage everyone must experience. Good software shortens that passage without bringing attention to it.

As an interaction designer, it's best to imagine that users—especially beginners—are simultaneously very intelligent and very busy. They need some instruction, but not very much, and the process has to be rapid and targeted. If a ski instructor begins lecturing on snowpack composition and meteorology, he will lose his students, regardless of their aptitude for skiing. Just because a user needs to learn how to operate a product doesn't mean that he needs or wants to learn how it works inside.

Imagine users as very intelligent but very busy.

On the other hand, intelligent people always learn better when they understand cause and effect, so you must help them understand why things work as they do. We use mental models to bridge the contradiction. If the interface's represented model closely follows the user's mental model (as discussed in Chapter 1), it will provide the understanding the user needs without forcing him or her to figure out the implementation model.

Certain kinds of products, especially those used in a transient manner (like most mobile apps), a distracted manner (Google Glass and other heads-up displays would qualify here), or those used by people with certain disabilities, must be optimized for beginners rather than intermediates. Examples include devices such as ATMs, informational kiosks designed for public spaces like museums, and consumer medical devices such as blood glucometers (used by patients with diabetes, who may have visual acuity problems and dexterity issues due to chronic numbness in their fingers).

Getting beginners on board

A new user must grasp the product's concepts and scope quickly, or he will abandon it. Thus, the designer's first order of business is to ensure that the product adequately reflects the user's mental model of his tasks. He may not recall from use to use exactly which command is needed to act on a particular object, but he will definitely remember the relationships between objects and actions—the important concepts—if the interface's conceptual structure is consistent with his mental model.

Getting beginners to a state of intermediacy requires extra help from the application, but this extra help will get in their way as soon as they become intermediates. This means that whatever extra help you provide, it must not be fixed into the interface. It must know how to go away when its services are no longer required.

Standard online help is a poor tool for providing such beginner assistance. We'll talk more about help in Chapter 16, but its primary utility is as a reference, and beginners don't need reference information; they need overview information, such as a guided tour, or UI elements designed to help users get accustomed to new functions, but which cease to be presented after repeated successful use.

A separate guide facility—displayed within a dialog box—is a fine means for communicating overview, scope, and purpose. As the user begins to use the product, a dialog box can appear that states the product's basic goals and tools, naming the main features. As long as the guide stays focused on beginner issues, like scope and goals, and avoids

perpetual intermediate and expert issues (discussed later), it should be adequate for assisting beginners.

Beginners also rely heavily on menus to learn and execute commands (see Chapter 18 for a detailed discussion about why this is true). Menus may be slow and clunky, but they are also thorough and verbose, so they offer reassurance. The dialog boxes that the menu items launch (if they do so at all) should also be tersely explanatory and come with convenient Cancel buttons.

Beginners across platforms

We are often asked if the concept of perpetual intermediates applies to non-desktop products. Ultimately, we believe the same considerations we apply to desktop software should be used here. A well-designed interface, regardless of platform should help its users quickly become familiar and comfortable with navigation and functionality.

Something else is worth considering: Users of websites, mobile apps, and devices that are not a critical path for their workflow—or are subject to casual consumer use—may not be accessed frequently enough by users for them to readily memorize their organizational constructs. This increases the importance of making such interactions as transparent and discoverable as possible, as well as the need for temporary assistive UI elements or guided tours that help reinforce understanding for new users.

What experts need

Experts (sometimes called influencers in marketing circles) are also a vital group, because their opinions have a disproportionate effect on purchasing. Experts of course listen to other experts, but they also exert an influence on other prospective customers, setting the tone for product reviews and discussions. This remains true even with the rise of online product ratings, though perhaps less so than before the likes of Amazon came into existence. Still, in many cases, when a beginner considers your product, he will trust the expert's opinion more than an intermediate's. This sometimes results in a disconnect: When an expert says, "It's not very good," she may really mean "It's not very good for experts like me." Beginners don't realize this, however, and will often take an expert's advice, even though it may not apply to their situational needs.

Experts might occasionally look for esoteric features, and they might make heavy use of a few of them. However, they will *definitely* demand faster access to their regular working set of tools, which may be quite large. In other words, *experts want shortcuts to everything.*

Anyone who uses a digital product for hours a day will very quickly internalize the nuances of its interface. It isn't so much that they *want* to cram frequently used commands

into their heads, as much as it is unavoidable. Their frequency of use both justifies and requires the memorization.

Expert users constantly and aggressively seek to learn more and to see more connections between their actions and the product's behavior and representation. Experts appreciate new, powerful features. Their mastery of the product insulates them from becoming disturbed by the added complexity. Experts also appreciate high information density relative to intermediate or beginning users.

For some specialized products, it is appropriate to optimize the user experience for experts. In particular, tools that technically minded people rely on for a significant portion of their professional responsibilities should be aimed at a high degree of proficiency. Development and creative authoring tools generally fall into this category, as do scientific instrumentation and (nonconsumer) medical devices. We expect the users of those products to already possess the necessary technical knowledge and to be willing to invest significant time and effort in mastering the application.

What perpetual intermediates need

It's amazing to think that the majority of real users—intermediates—typically are ignored, but more often than not that is still the case. You can see this in many enterprise applications and digital products. The overall design biases them toward expert users. At the same time, cumbersome tools such as wizards or the likes of Clippy—the infamously annoying ("Would you like help?") "smart" assistant created by Microsoft for its Office suite of products in the 1990s—are grafted onto the product to meet marketing's perception of new users. Experts rarely use them, and beginners soon want to discard these embarrassing reminders of their ignorance. But the perpetual intermediate majority is perpetually stuck with them.

Instead, perpetual intermediates need fast access to the most common tools. They don't need scope and purpose explained to them, because they already know these things. ToolTips (see Chapter 20) are the perfect perpetual intermediate idiom. ToolTips say nothing about scope and purpose and meaning; they only state function in the briefest of idioms, consuming the least amount of video space in the process.

Perpetual intermediates know how to use reference materials. They are motivated to dig deeper and learn, as long as they don't have to tackle too much at once. This means that online help is a perpetual intermediate tool. They use it by way of the index, so that part of help must be comprehensive.

Perpetual intermediates establish the functions that they use with regularity and those that they use only rarely. The user may experiment with obscure features, but he will soon identify—probably subconsciously—his frequently used working set. The user will

demand that the tools in his working set be placed front and center in the user interface, where they are easy to find and remember.

Perpetual intermediates usually know that advanced features exist, even though they may not need them or know how to use them. But the knowledge that they are there is reassuring to the perpetual intermediate, convincing him that he made the right choice investing in this product. The average skier may find it inspirational to know that a scary, black-diamond, expert run is just beyond those trees, even if she never intends to use it. It gives her something to aspire to and dream about, and it gives her the sense that she's at a good ski resort.

Your product must likely provide for both absolute newbies and the many possible cases an expert might encounter. But don't let this business requirement influence your design thinking. Yes, you must provide those features for expert users. Yes, you must support those transient beginners. But in most cases, you need to apply the bulk of your talents, time, and resources to designing the best interaction possible for your most representative users: the perpetual intermediates. When digital products follow the principle of commensurate effort, the learning curve doesn't go away, but it disappears from the user's mind—which is just as good.

ORCHESTRATION AND FLOW

If our goal is to make the people who use our products more productive, effective, and engaging, we must ensure that users remain in the right frame of mind. This chapter discusses a kind of mental ergonomics. It describes how we can ensure that our products support user intelligence and effectiveness. It also covers how we can avoid disrupting the state of productive concentration that we want our users to be able to maintain.

Flow and Transparency

When people concentrate wholeheartedly on an activity, they lose awareness of peripheral problems and distractions. The state is called *flow*, a concept first identified by Mihaly Csikszentmihalyi in *Flow: The Psychology of Optimal Experience* (Harper, 2008).

In *Peopleware: Productive Projects and Teams* (Dorset, 1999), Tom DeMarco and Timothy Lister describe flow as a "condition of deep, nearly meditative involvement." Flow often induces a "gentle sense of euphoria" and can make you unaware of the passage of time. Most significantly, a person in a state of flow can be extremely productive, especially when engaged in constructive activities such as engineering, design, development, or writing. To state the obvious, then, to make people more productive and happy, it behooves us to design interactive products to promote and enhance flow. We also should go to great lengths to avoid any potentially flow-disturbing behavior. If the application consistently rattles a user and disrupts her flow, it becomes difficult for her to maintain that productive state.

In most cases, if the user could achieve his goals magically, without your product, he would. By the same token, if the user needs the product but could achieve his goals without

messing about with a user interface, he would. Interacting with productivity software seldom is an entirely aesthetic experience. Entertainment and creative tools aside, interacting with software (especially business software) is very much a pragmatic exercise.

Directing your attention to the interaction itself puts the emphasis on the side effects of the tools and technology rather than on the user's goals. A user interface is an artifact, not directly associated with the user's goals. Next time you find yourself crowing about a cool interaction you've designed, remember that the ultimate user interface can often be *no* interface.

To create a sense of flow, our interaction with software must become *transparent*. When a novelist writes well, the writer's craft becomes invisible, and the reader sees the story and characters with clarity undisturbed by the writer's technique. Likewise, when a product interacts well with a person, interaction mechanics disappear, leaving the person face to face with his objectives, unaware of the intervening software. Just as a poor writer is a visible writer, a poor interaction designer looms with a clumsily visible presence in his software.

Orchestration

To a novelist, there is no such thing as a "good" sentence in isolation from the story being told. No rules govern how sentences should be constructed to be transparent. It all depends on what the protagonist is doing, or what effect the author wants to create. The writer knows not to insert an obscure word in a particularly quiet and sensitive passage, lest it sound like a sour note in a string quartet. The same goes for software. The interaction designer must train himself to hear sour notes in the *orchestration* of software interaction. It is vital that all the elements in an interface work together coherently toward a single goal. When an application's communication with a person is well orchestrated, it becomes almost invisible.

Webster defines orchestration as "harmonious organization," a reasonable phrase for what we should expect from interactive products. Harmonious organization doesn't yield to fixed rules. You can't create guidelines like "Four buttons on a thumb-driven mobile menu is good" and "Six buttons on a thumb-driven mobile menu is too many." Yet it is easy to see that a thumb-driven menu with 35 buttons wouldn't work. The major difficulty with such analysis is that it treats the problem *in vitro*. It doesn't take into account the problem being solved; it doesn't take into account what a person is doing at the time or what he is trying to accomplish.

Harmonious Interactions

Although no universal rules define a harmonious interaction (just as no universal rules define a harmonious interval in music), we've found the following strategies to be effective for designing interactions that go with the user's flow:

- Follow users' mental models.
- Less is more.
- Let users direct rather than discuss.
- Provide choices rather than ask questions.
- Keep necessary tools close at hand.
- Provide modeless feedback.
- Design for the probable but anticipate the possible.
- Contextualize information.
- Reflect object and application status.
- Avoid unnecessary reporting.
- Avoid blank slates.
- Differentiate between command and configuration.
- Hide the ejector seat levers.
- Optimize for responsiveness but accommodate latency.

Follow users' mental models

We introduced the concept of user mental models in Chapter 1. Different people have different mental models of a given activity or process, but they rarely imagine them in terms of the detailed mechanics of how computers function. Each user naturally forms a mental image of how the software performs its task. The mind looks for some pattern of cause and effect to gain insight into the machine's behavior.

For example, in a hospital information system, the physicians and nurses have a mental model of patient information that derives from how they think about patients and treatment. It therefore makes the most sense to find patient information by using patient names as an index. Each physician has certain patients, so it makes additional sense to filter the patients in the clinical interface so that each physician can choose from a list of his or her own patients, organized alphabetically by name. On the other hand, in the hospital's business office, the clerks there are worried about overdue bills. They don't initially think about these bills in terms of who or what the bill is for, but rather in terms of how late the bill is (and perhaps how big it is). Thus, for the business office interface,

it makes sense to sort first by time overdue and perhaps by amount due, with patient names as a secondary organizational principle.

Less is more

For many things, more is better. In the world of interaction design, the contrary is usually true. We should constantly strive to reduce the number of elements in user interfaces without reducing the capabilities of the products we are creating and without increasing the effort it takes to use them. To do this, we must do more with less; this is where careful orchestration becomes important. We must coordinate and control the product's power without letting the interface become a gaggle of screens and widgets, covered with a scattering of unrelated and rarely used controls.

It is common for user interfaces of professional and business software to be complex but not very powerful. Products like this typically segregate functionality into silos and allow the user to perform a single task without providing access to related tasks. When the first edition of this book was published in 1995, this problem was ubiquitous. Something as common as a Save dialog in a Windows application failed to allow users to also rename or delete the files they were looking at. The users had to go to a different place to accomplish these very similar tasks, ultimately requiring applications and operating systems to provide *more interface*. Thankfully, contemporary operating systems are much better at this sort of thing. Because they have started to offer appropriate functionality based on the user's context, users are less often required to shuffle off to various places in the interface to accomplish simple and common tasks.

However, we have a rather long way to go. In the enterprise software we see, each function or feature is often housed in a separate dialog or window, with little consideration for how people must use these functions together to accomplish something. It is not uncommon for a user to use one menu command to open a window to find a bit of information, copy that information to the clipboard, and then use a different menu command for a different window, merely to paste that bit of information into a field. Not only is this procedure inelegant and crude, but it is also error-prone and fails to capitalize on a productive division of labor between humans and machines. Typically, products don't end up this way on purpose. They have been built either in an ad hoc manner over years or by several disconnected teams in different organizational silos.

Motorola's once popular Razr V3 flip-phone was an example of this problem. Although the phone's industrial design was deservedly award-winning for its elegance, the software was inherited from a previous generation of Motorola phones and appeared to have been developed by multiple teams who didn't coordinate their efforts. For example, the phone's address book used a different text-entry interface than its calendar application. Each software team must have devised a separate solution, resulting in two interfaces doing the job that one should have done. This was both a waste of development resources

and a source of confusion and friction for Motorola's users. A year after the V3 reached the height of its popularity, the iPhone, with its modern, well-considered user interface arrived, and the V3 and all their flip-phone brethren were soon history. Tight integration of the complete hardware and software experience finally won the day.

Mullet and Sano's classic *Designing Visual Interfaces* (Prentice Hall, 1994) includes a useful discussion of the idea of *elegance*, which can be thought of as a novel, simple, economical, and graceful way of solving a design problem. Because the software inside an interactive product is typically complex, it becomes all the more important to value elegance and simplicity; these attributes are crucial for technology to effectively serve human needs.

A minimalist approach to product design is inextricably tied to a clear understanding of purpose—what the user of a product is trying to accomplish using the tool. Without this sense of purpose, interactive products are just a disorganized jumble of technological capabilities. A model example where a strong sense of purpose has driven a minimal user interface is the classic Google search interface, shown in Figure 11-1. It consists of a text field, two buttons (Google Search, which takes the user to a list of results, and I'm Feeling Lucky, which takes the user directly to the top result), the Google logotype, and a couple of links to the broader universe of Google functionality. Another good example of a minimal user interface is the iPod Shuffle. By carefully defining an appropriate set of features to meet a specific set of user needs, Apple created a highly usable product with one switch and five buttons (and no screen!). Still another example is iA Writer, an incredibly simple iOS text editor app. It doesn't have much of a user interface other than an area in which to write text. The text is saved automatically, eliminating the need to interact with files.

Figure 11-1: The celebrated Google search interface is a classic example of minimalist interface design, where every screen element is purposeful and direct.

It's worth noting that the quest for simplicity can be taken too far; reduction is a balancing act that requires a good understanding of users' mental models. The iPod Shuffle's

interface, an example of elegance and economy in design, is also at odds with some users' expectations. If you come from the world of CD players, or even the high-resolution screens of most other digital audio players, it probably feels a bit weird to use the iPod's Play/Pause toggle to shut off the device and the Menu button to turn it on. This is a classic case of visual simplicity leading to cognitive complexity. In this situation, these idioms are simple enough to learn easily, and the consequences of getting it wrong are fairly small, so the product's overall success hasn't been affected much.

Stick to "less is more" to keep out of your users' way and keeping them in flow.

Let users direct rather than discuss

Some developers might imagine that the ideal user interface is a two-way conversation between human and machine. However, most people don't see it that way. Most people would rather interact with the software in the same way they interact with, say, their cars. They open the door and get in when they want to go somewhere. They step on the accelerator when they want the car to move forward and the brake when it's time to stop. They turn the wheel when they want the car to turn.

This ideal interaction is not a dialogue—it's more like using a tool. When a carpenter hits nails, he doesn't discuss the nail with the hammer; he directs the hammer onto the nail. In a car, the driver gives the car direction when he wants to change the car's behavior. The driver expects direct feedback from the car and its environment in terms appropriate to the device: the view out the windshield, the readings on the various gauges on the dashboard, the sound of rushing air and tires on pavement, and the feel of lateral g-forces and vibration from the road. The carpenter expects similar feedback: the feel of the nail sinking, the sound of steel striking steel, and the shifting weight as the hammer recoils.

The driver certainly doesn't expect the car to interrogate him with a dialog box, nor would a carpenter appreciate the dialog shown in Figure 11-2 if it appeared on his hammer.
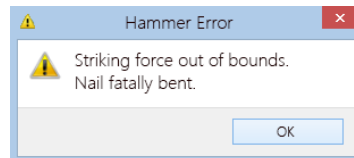
Figure 11-2: Nobody wants to be scolded, particularly by a machine. If we guide our machines in a dunderheaded way, we *expect* to get a dunderheaded response. Sure, they can protect us from fatal errors, but scolding isn't the same thing as protecting.

One of the reasons interactive products often aggravate people is that they don't act enough like cars or hammers. Instead, they have the temerity to try to engage us in a dialogue—to inform us of our shortcomings and to demand answers. From the user's point of view, the roles are reversed: The person should do the demanding, and the software should do the answering. One of the most important ways of letting the users direct the action in an interface is *direct manipulation*. We'll discuss this at length in Chapter 13.

Provide choices rather than ask questions

Dialog boxes (confirmation dialogs in particular) ask questions. Toolbars and palettes offer choices. The confirmation dialog stops the proceedings, demands an answer, and doesn't leave until it gets what it wants. Toolbars and palettes, on the other hand, are always there, quietly and politely offering their wares like a well-appointed store, giving you the luxury of selecting what you want with just a flick of your finger.

Choices are important, but there is a difference between being free to make choices based on presented information and being interrogated by the application in modal fashion. Users would much rather direct their software the way they direct their automobiles down the street. Automobiles offer drivers sophisticated choices without once issuing a dialog box. Imagine the situation shown in Figure 11-3.
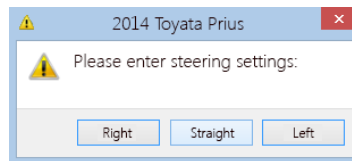
Figure 11-3: Imagine if you had to steer your car by clicking buttons on a dialog box! This dialog box gives you some idea of how normal people feel about the dialog boxes in *your* software.

Not only is directly manipulating a steering wheel a more appropriate idiom for communicating with your car, but it also puts you in the superior position, directing your car where it should go. Modeless choices help give users the feeling of control and mastery they want when using digital products.

Keep necessary tools close at hand

Most desktop applications are too complex for one mode of interaction to cover all their features. Consequently, many applications offer users a palette of tools. These tools are actually different modes of behavior that the product enters. Offering tools is a compromise with complexity, but we can still do a lot to make tool selection and manipulation

easy and to prevent it from disturbing flow. Mainly, we must ensure that information about tools and application state is clear and present and that transitions between tools are quick and simple.

Tools should be close at hand, commonly on palettes or toolbars for beginner and intermediate users and accessible by keyboard command for expert users. This way, the user can see them easily and can select them with a single click or keystroke. If the user must divert his attention from the application to search out a tool, his concentration will be broken. It's as if he had to get up from his desk and wander down the hall to find a pencil. Also, he should never have to put tools away.

Provide modeless feedback

When users of an interactive product manipulate tools and data, it's usually important to clearly present the status and effect of these manipulations. This information must be easy to see and understand without obscuring or interfering with the user's actions. Feedback of progress is one of the key elements of flow.

An application has several ways to present information or feedback to users. One egregious way done on the desktop is to pop up a dialog box. This technique is modal: It puts the application into a special state that must be dealt with before it can return to its normal state and before the person can continue with her task. A better way to inform users is with *modeless feedback*.

Feedback is *modeless* whenever information for users is built into the structures of the interface and doesn't stop the normal flow of activities and interaction. In Microsoft Word 2010, shown in Figure 11-4, you can see what page you are on, what section you are in, how many pages are in the current document, and what position the cursor is in— modelessly. You just have to look at the left navigation pane and status bar at the bottom of the screen. You don't have to go out of your way to ask for that information.

Another good example is the iOS notification center, which displays a brief heads up alert when an app that isn't currently active on the screen has an important event to report, such as an upcoming appointment. The message stays at the top of the screen for a few seconds, and then disappears, and tapping it while it is displayed takes you to the notifying app.

Jet fighters have a heads-up display, or HUD, that superimposes the readings of critical instrumentation onto the forward view of the cockpit's windscreen. The pilot doesn't even have to use peripheral vision; she can read vital gauges while keeping her eyes on the opposing fighter. Applications can use the edges of the display screen to show users

information about activity in the main work area. Many drawing applications, such as Adobe Photoshop, already provide ruler guides, thumbnail maps, and other modeless feedback in the periphery of their windows. We further discuss rich modeless feedback in Chapter 15.
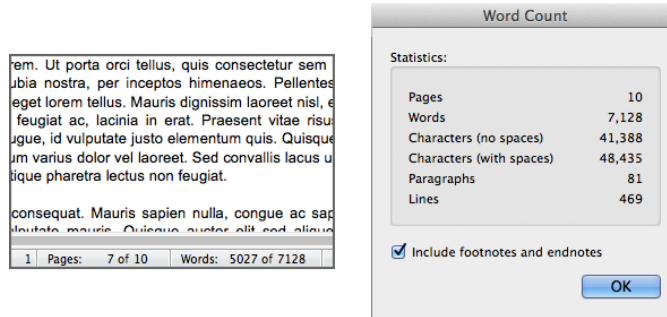
Figure 11-4: In Word 2010, Microsoft lets you see what page you are on, the number of total pages, and the number of words in the document displayed modelessly on the lower-left edge of the window. Clicking on the word count opens the Word Count dialog, which provides more detailed information.

Design for the probable but anticipate the possible

Superfluous interaction, usually in the form of a dialog box, often slips into a user interface. This is often the result of an application being faced with a choice— developers tend to resolve choices from the standpoint of logic, and this carries over to their software design. To a logician, if a proposition is true 999,999 times out of a million and is false one time, the proposition is false—that's how Boolean logic works. However, to the rest of us, the proposition is overwhelmingly true. The proposition has a *possibility* of being false, but the *probability* of its being false is minuscule to the point of irrelevancy. One of the most potent methods for better orchestrating your user interfaces is segregating the possible from the probable.

Developers tend to view possibilities as being the same as probabilities. For example, the user can decide to end the application and save his work, or end the application and discard the document he has been working on for the last six hours. Either of these choices is possible. The probability that this person will discard his work is at least a thousand to one against, yet the typical application always includes a dialog box asking the user if he wants to save his changes, like the one shown in Figure 11-5.

This dialog box is inappropriate and unnecessary. How often do you choose to abandon changes you make to a document? This dialog is tantamount to your spouse telling you not to spill soup on your shirt every time you eat. We'll discuss the implications of removing this dialog in Chapter 14.
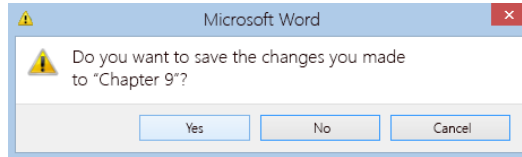
Figure 11-5: This is easily the most unnecessary dialog box in the world of GUI. Of course we want to save our work! It is the normal state of events. *Not* saving it would be out of the ordinary and would be worthy of a dialog, but not this.

Developers are judged by their ability to create software that handles the many possible, but improbable, conditions that crop up inside complex logical systems. This doesn't mean, however, that they should render that readiness to handle offbeat possibilities directly into a user interface. This sort of thing runs counter to a user's expectations and interrupts their flow by asking them to accommodate the possibility. Dialogs, controls, and options that are used a hundred times a day should not sit side by side with dialogs, controls, and options that are used once a year or never.

You *might* get hit by a bus, but you probably will get to work safely this morning. You don't stay home out of fear of the killer bus, so don't let what might possibly happen alter how you treat what almost certainly will happen in your interface.

Contextualize information

How an application chooses to represent information is another thing that can confuse or overwhelm normal humans. One area frequently abused is the representation of quantitative, or numeric, information. If an application needs to show the amount of free space on disk, it *could* do what the ancient Windows 3.0 File Manager did: give you the *exact* number of free bytes, as shown in Figure 11-6.
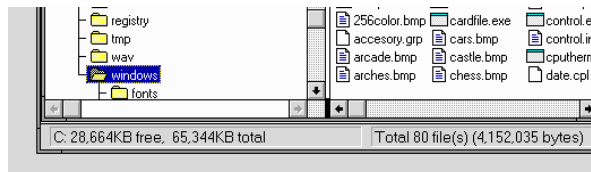
Figure 11-6: The old Windows 3.0 File Manager took great pains to report the exact number of bytes used by files on the disk. Did this precision help us understand if we needed to clear space on the disk? Wouldn't a visual representation that showed disk usage in a proportional manner be more meaningful? Luckily, Windows now employs bar and pie charts to indicate disk usage.

In the lower-left corner, the application tells us the number of free bytes and the total number of bytes on the disk. These numbers are hard to read and interpret. With billions of bytes of disk storage, it ceases to be important to us just how many hundreds are left, yet the display rigorously shows us down to the kilobyte. But even while the application is telling us the state of our disk with precision, it is failing to communicate. What we really need to know is whether the disk is getting full, or whether we can add a new 20 MB application and still have sufficient working room. These raw numbers, precise as they are, do little to help us make sense of the facts, and pull us out of flow as we try and figure out what's really happening.

Visual information design expert Edward Tufte says that quantitative presentation should answer the question "Compared to what?" Knowing precisely how many bytes are free on your hard disk is less useful than knowing that it is 22 percent of the disk's total capacity. Another Tufte dictum is "Show the data (visually)," rather than simply telling about it textually or numerically.

A bar or pie chart showing the used and unused portions would make it much easier to comprehend the scale and proportion of hard disk use. The numbers shouldn't go away entirely, but they should be relegated to the status of labels on the display and not *be* the display. They should also be shown with more reasonable and consistent precision. The meaning of the information could be shown visually; the numbers would merely add support. Today, this is exactly what is shown in Windows Explorer. Unfortunately, this useful info is shown in only one place, rather than as a persistent status indicator at the bottom of all Explorer windows. And, unfortunately, the problem persists in lots of other applications.

Reflect object and application status

When someone is asleep, he usually looks asleep. When someone is awake, he looks awake. When someone is busy, he looks busy: His eyes are focused on his work, and his body language is closed and preoccupied. When someone is unoccupied, he looks unoccupied: His body is open and moving; his eyes are willing to make contact. People not only expect this kind of subtle feedback from each other, they also depend on it to maintain social order.

These sorts of cues are important enough that they became a core part of the user interface of Baxter, a two-armed stationary industrial robot created by Rethink Robotics (see Figure 11-7), whose founder, Rodney Brooks, also invented the Roomba vacuuming robot. Baxter is designed to work alongside humans on a light manufacturing line. It features a large, face-like screen with cartoonish animated eyes that can look in a direction before reaching the destination. It reports system status via simple and universal facial expressions.
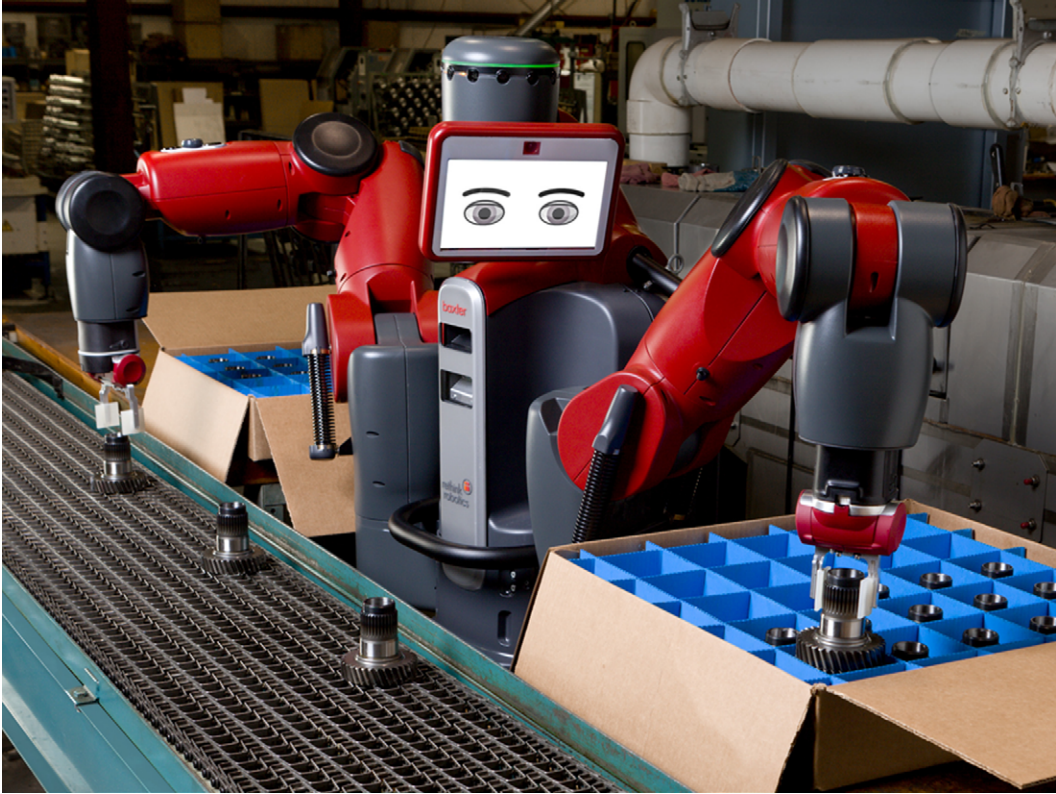
Figure 11-7: Baxter is a two-armed industrial robot designed to work alongside humans in a light manufacturing production line. It communicates status using facial expressions.

While they probably should not be anthropomorphized as fully as Baxter, our day-to-day software applications and devices should provide similar clues. When an application is asleep, it should look asleep. When an application is awake, it should look awake. When it's busy, it should look busy. When the product is engaged in some significant internal action like performing a complex calculation and connecting to a database, it should be obvious to us that it won't be quite as responsive as usual. When the app is sending a large file, we should see a modeless progress bar. This lets the user plan their next steps accordingly.

Similarly, the status of user interface objects should be apparent to users. Most e-mail applications do a good job of making it obvious which messages have not been read and which have been responded to or forwarded. Let's take this concept a step further. Wouldn't it be great if, when you were looking at events in the day or week views of Microsoft Outlook or Google Calendar, you could tell how many people had agreed to attend and how many hadn't responded yet (either right inline or via ToolTip) without drilling down into the details?

Application and object state is best communicated using forms of rich modeless feed-back, briefly discussed earlier in this chapter. More detailed examples of rich modeless feedback may be found in Chapter 15.

Avoid unnecessary reporting

Some applications are quick to keep users apprised of the details of their progress even though the user has no idea what to make of this information. Applications pop up noti-fications telling us that connections have been made, that records have been posted, that users have logged on, that transactions were recorded, that data has been trans-ferred, and other useless factoids. To software engineers, these messages are equivalent to the humming of the machinery: They indicate that all is well. In fact, they probably were used while debugging the application. To a normal person, however, these reports can feel like eerie lights beyond the horizon, screams in the night, or unattended objects flying about the room.

For users, it is disconcerting and distracting to know all the details of what is happening under normal conditions. Nontechnical people may be alarmed to hear that the data-base has been modified, for example. It is better for the application to simply do what has to be done, issue reassuring (and modeless) visual or auditory feedback when all is well, and not burden users with the trivia of *how* it was accomplished. It is important that we not stop the proceedings to report *normalcy*. If you must use them, reserve notifications for events that are outside the normal course of events. If your users benefit from know-ing things are running smoothly, use some more ambient signal.

> **DESIGN PRINCIPLE**
>
> *Don't use dialogs to report normalcy.*

By the same token, don't stop the proceedings and bother the user with minor problems. If the application is having trouble creating a connection to a server, don't put up a dialog box to report it. Instead, build a status indicator into the application so that the problem is clear to the interested user but is unobtrusive to someone who is busy elsewhere.

Avoid blank slates

The key to orchestrating a user interaction is to take a goal-directed approach. You must ask yourself whether a particular interaction moves a person effectively and confidently toward his goal. Timid applications are reluctant to carry out any forward motion without someone's directing them in advance. But most people would rather see the application take a "good enough" first step and then manually tweak it to what is desired. This way, the application moves the person closer to his goal.

It's easy to assume nothing about what your users want, instead asking a bunch of questions up front to help determine what they want. How many applications have you seen that start by asking a bunch of questions upfront? Or that punt every decision to a litany of user options? But normal people—rather than "expert users"—sometimes are incapable of or uncomfortable with explaining what they want to do to an interactive product, especially in advance. They would much rather see what the application *thinks* is right and then manipulate that to make it exactly right. In most cases, your application can make a fairly correct assumption based on the designer's estimation, past experience with this user, or by reference to most other users.

For example, when you create a new document in PowerPoint, on the PC, the application creates a blank document with preset attributes rather than opening a dialog that asks you to specify every detail. Omnigraffle on the Mac does a less adequate job, asking you to choose the base style for a new presentation each time you create one. Both applications could do better by remembering frequently and recently used styles or templates and making those the defaults for new documents.

Just because we use the word *think* in conjunction with an interactive product doesn't mean that the software needs to be particularly intelligent (in the human sense) and must try to determine the right thing to do by reasoning. Instead, the software should simply do something that has a probability of being correct. Then it should provide the user with powerful tools for shaping that first attempt, instead of merely giving the user a blank slate and challenging him to have at it. This way, the application doesn't ask for permission to act, but rather for forgiveness after the fact.

> **DESIGN PRINCIPLE**   *Ask for forgiveness, not permission.*

For most users, a blank slate is a difficult starting point. It's much easier to begin where someone else has left off. They can easily fine-tune an approximation provided by the application into precisely what he wants with less risk of exposure and mental effort than he would have by drafting it from nothing. As we discussed in Chapter 8, endowing your application with a good memory is the best way to accomplish this.

Differentiate between command and configuration

Another problem crops up frequently whenever users invoke functions with many parameters. The problem comes from the lack of differentiation between a function and the *configuration* of that function. If you ask an application to perform a function by itself, the application should simply do it using a reasonable default or its last configuration. It should not interrogate you about precise configuration details each time it is

used. To express different or more precise demands to the application, you would launch the configuration interface for that function.

For example, when you ask many applications to print a document, they respond by launching a complex dialog box demanding that you specify how many copies to print; what the paper orientation is; what paper feeder to use; what margins to set; whether the output should be in monochrome or color; what scale at which to print; whether to use PostScript fonts or native fonts; whether to print the current page, the current selection, or the entire document; and whether to print to a file and, if so, what to name that file. All these options are useful, but all we want is to print the document, and that is all we thought we asked for.

More reasonable designs have one command to print and another command for print setup. The print command issues a dialog but just goes ahead and prints, using either previous settings or standard settings. The print setup function offers all those choices about paper and copies and fonts. Some applications allow the user to go directly from the configure dialog to printing, or vice-versa.
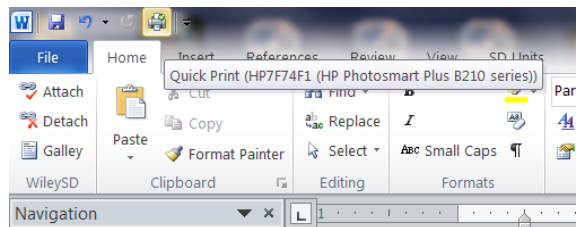
Figure 11-8: The Quick Print control in Microsoft Word offers immediate printing without a dialog box.

The Quick Print control in Microsoft Word offers immediate printing without a dialog box (although unfortunately it is very small and hidden by default—see Figure 11-8). This is perfect for many people, but for those with multiple printers or printers on a network, it may offer too little information. The user may want to see which printer is selected before he either clicks the control or summons the dialog to change it first. This is a good candidate for some simple modeless output placed on a toolbar or status bar. (It is currently provided in the control's ToolTip in the Windows version, which is good, but the feedback could be better still.) Word's print setup user interface (which also includes a Print button) is called Print and is available as a menu item on the File tab of Word's ribbon control (more about that in Chapter 18).

There is a big difference between configuring and invoking a function. The former may include the latter, but the latter shouldn't include the former. Generally speaking, a user invokes a command ten times for every one time he configures it. It is better to make the

user ask explicitly for configuration one time in ten than it is to make the user reject the configuration interface *nine* times in ten.

Thus, most desktop applications have a reasonable rule of thumb: Put immediate access to functions on buttons in toolbars, and put access to function-configuration user interfaces in menus. The configuration tools are better for learning and tinkering, whereas the buttons provide immediate and simple action.

Hide the ejector seat levers

In the cockpit of every fighter jet is a brightly colored lever that, when pulled, fires a small rocket engine under the pilot's seat (see Figure 11-9). This blows the pilot, still in his seat, out of the aircraft so that he can then parachute safely to Earth. Ejector seat levers can be used only once, and their consequences are significant and irreversible.
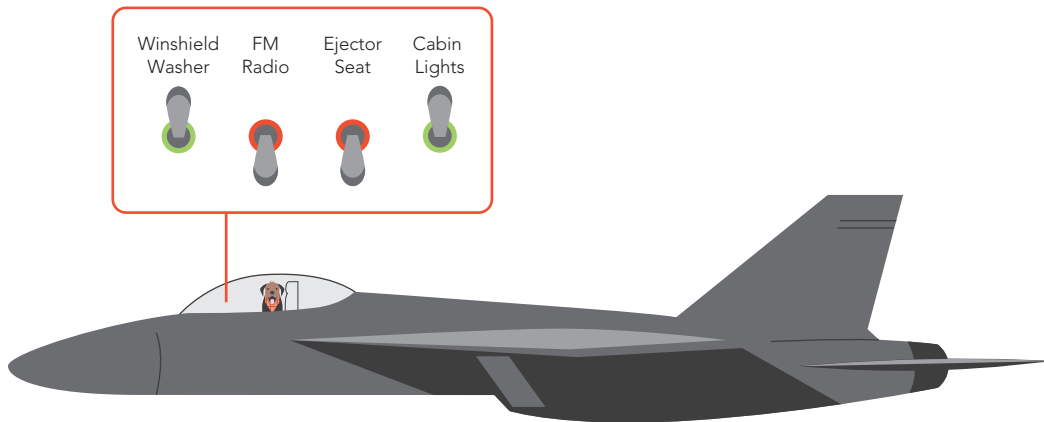
Figure 11-9: Ejector seat levers have catastrophic results. One minute, the pilot is safely ensconced in his jet, and the next he is tumbling end over end in the wild blue yonder, while his jet goes on without him. The ejector seat is necessary for the pilot's safety, but a lot of design work has gone into ensuring that it never gets fired inadvertently. Allowing an unsuspecting user to configure an application by changing permanent objects is comparable to firing the ejector seat by accident. Hide those ejector seat levers!

Just as a jet fighter needs an ejector seat lever, complex desktop applications need configuration facilities. Applications must have ejector seat levers so that users can occasionally move *persistent objects* (see Chapter 12) in the interface, or dramatically (sometimes irreversibly) alter the application's function, behavior, or content. The one thing that must never happen is accidental deployment of the ejector seat (see Figure 11-9). The interface design must ensure that the user can never inadvertently fire the ejector seat when all he wants to do is make a minor adjustment to the application.

Ejector seat levers come in two basic varieties: those that cause a significant visual dislocation (large changes in the layout of tools and work areas) in the application, and those that perform an irreversible action. Both of these functions should be hidden from inexperienced users. Of the two, the latter variety is by far the more dangerous. In the former, the user may be surprised and dismayed at what happens next, but she can at least back out of it with some work. In the latter case, she and her colleagues are likely to be stuck with the consequences.

If you keep in mind principles of flow and orchestration, your software can keep users engaged at maximum productivity for extended periods of time. Productive users are happy users, and customers with productive, happy users are the goal of almost any digital product manufacturer. In the next chapter, we further discuss ways to enhance user productivity by eliminating barriers to use that arise as a result of implementation-model thinking.

Optimize for responsiveness but accommodate latency

An application can become slow or unresponsive when it performs a large amount of data processing or when it waits on remote devices like servers, printers, and networks. Nothing is more disturbing to the user's sense of flow than staring at the screen, waiting for the computer to respond. It's critical to design your interfaces so that they are sufficiently responsive. All the lush visual style in the world won't impress anyone if the interface moves like molasses because the device is maxed out redrawing the screen.

This is one arena where collaboration with developers is quite important. Depending on the platform and technical environment, different interactions can be quite "expensive" from a latency perspective. You should advocate for implementation choices that provide the user with appropriately rich interactions with as little latency as possible. You also should design solutions to accommodate choices that have been made and cannot be revisited. When latency is unavoidable, it's important to clearly communicate the situation to users and allow them to cancel the operation causing the latency and ideally perform other work while they are waiting.

If your application executes potentially time-consuming tasks, make sure that it occasionally checks to see if someone is still out there madly clicking the mouse and whimpering, "No, no, I didn't mean to reorganize the *entire* database. That will take 4.3 million years!"

In a number of studies dating back to the late 1960s, it's generally been found that users' perception of response times can be roughly categorized into several buckets:[1]

- Up to **0.1 seconds**, users perceive the system's response as **instantaneous.** They feel that they are directly manipulating the user interface and data.

- Up to about **1 second**, users feel that the system is **responsive**. Users will likely notice a delay, but it is small enough for their thought processes to stay uninterrupted.

- Up to about **10 seconds**, users clearly notice that the system is slow, and their mind is likely to wander, but they can keep some amount of attention on the application. Providing a progress bar is critical here.

- After about **10 seconds**, you will lose your users' attention. They will wander off and get a cup of coffee or switch to a different application. Ideally, processes that take this long should be conducted offline or in the background, allowing users to continue with other work. In any case, status and progress should be clearly communicated, including estimated time remaining. A cancel mechanism is critical.

Motion, Timing, and Transitions

The first computing device to use motion and animated transitions as core elements of the user experience was the Apple Macintosh. Mac windows sprang open from dragable app and folder icons and collapsed back into them when closed. Menus dropped open when clicked and rolled up again when the mouse button was released. The Switcher facility in early Mac OS allowed you to change the current open application by clicking a control in the menu bar. The control caused the current app's screen to slide horizontally out of view to the left. Another open app's screen slid in from the right like a carousel. (Amusingly, this carousel-like app transition has reappeared on the iPad as an optional four-fingered left/right swipe gesture.)

In later versions of Mac OS and Windows, more animated transitions were added. Dialogs no longer simply appeared; they slid or popped into place. Expandable drawers, palettes, and panels became common idioms, especially in professional software.

However, it was not until the advent of the iPhone that the use of motion and animated transitions became an integral and critical part of the digital product experience. In concert with multitouch gestures, animated transitions allow mobile apps to appear so responsive and immersive that you almost forget that what is being flicked, pinched, twirled, and swiped onscreen is really just pixels providing an illusion of physicality.

Motion is a powerful mechanism for expressing and illustrating the relationships between objects. This mechanism has been particularly successful on mobile devices, where the form factor limits what can be shown onscreen. Animated transitions help users create a strong mental model of how what is presented in one view is related to what was presented in the previous view. It's often used to good effect on the web as well, helping create a spatial aspect to navigation and state transitions.

Although it's tempting to do otherwise, motion and animation must always be used sparingly and judiciously. Not only is an overabundance of motion potentially confusing and irritating, but it also can make some people ill. This fact was reported after the release of Apple's iOS 7, possibly due to its new and somewhat overzealous parallax and app zoom-out/zoom-in animations.

The overarching goal of motion and animated transitions in interaction should be to support and enhance the user's state of flow. As Dan Saffer discusses in his excellent book, *Microinteractions* (O'Reilly, 2013), animations and transitions should help achieve the following:[2]

- Focus user attention in the appropriate place.
- Show relationships between objects and their actions.
- Maintain context during transitions between views or object states.
- Provide the perception of progression or activity (such as through progress bars and spinners).
- Create a virtual space that helps guide users from state to state and function to function.
- Encourage immersion and further engagement.

Furthermore, designers should strive for these qualities when creating interactions involving motion and animation:[3]

- **Short, sweet, and responsive**—Animations should not slow down interactions (and thus interrupt flow). They should last only as long as it takes to accomplish one or more of the goals just listed, and in any case less than a second to retain a feeling of responsiveness.
- **Simple, meaningful, and appropriate**—In iOS7, Apple changed how you "kill" a running app. Previously, you tapped and held the app icon in the multitasking tray, waited for an X icon to appear on it, tapped it, and then pressed the home button to exit a mode. (This was almost the same action you took to delete the app from the product.) Now, you flick a representation of the app's last screen away from you, causing it to scoot off the top of the screen. This is much simpler and more satisfying, and it is appropriate to the function it triggers. (Sadly, it is equally undiscoverable, as shown in Figure 11-10.)
- **Natural and smooth**—Animated transitions, especially those providing feedback to gestural interfaces, should feel almost like real physical interactions, mimicking (if not modeling) motion attributes such as inertia, elasticity, and gravity.
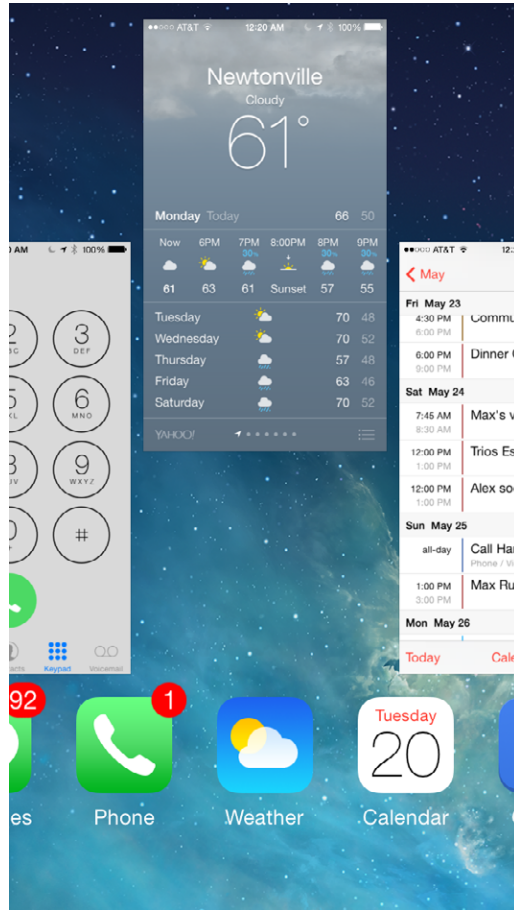
Figure 11-10: In iOS7, to kill an app, you flick a representation of the app's last screen away from you. This is much simpler and more satisfying than the old method—tapping and holding the app icon to put it into a "delete mode."

Motion is most successful when it has a rhythmic quality, in which the timing helps the user anticipate what will be shown next. Changes in timing can be used to cue users about changes in context, state, or mode. This visual feedback can also be reinforced by the use of sounds. Sounds can help direct user interaction (the "tap" of a button in iOS), express the effect of user interaction (the clicking as the selection changes in the PlayStation 3's main horizontal menu), or reinforce a transition (a whoosh that accompanies a swipe gesture).

The Ideal of Effortlessness

Creating a successful product requires more than delivering useful functionality. You must also consider how different functional elements are orchestrated to enable users to achieve a sense of *flow* as they go about their business. The best user interfaces often don't leave users in awe of their beauty, but rather are hardly even noticed because they can be used effortlessly.

Understanding the importance of flow, orchestrating your interface to maximize it, and making judicious use of motion and transitions to ease the user from one state or mode to another can give your apps the aura of effortlessness that helps make them seem to work like magic.

Notes
1. Miller, 1968
2. Saffer, 2012
3. Haase and Guy, 2010

REDUCING WORK AND ELIMINATING EXCISE

Digital products too often contain interactions that are top-heavy, requiring unnecessary work for users. Interacting with an interface always involves some work on the part of the user. The goal of designers (or at least one of the more important ones) is to minimize that work, while at the same time enabling users to achieve their goals. If designers and developers don't pay careful attention to the human actions required to operate their technology, the result can be a taxing experience for the users. They will struggle to relate their mental models of the activities they want to perform to the product interface that has been engineered.

Users perform four types of work when interacting with digital products:

- **Cognitive work**—Comprehending product behaviors, as well as text and organizational structures
- **Memory work**—Recalling product behaviors, commands, passwords, names and locations of data objects and controls, and other relationships between objects
- **Visual work**—Figuring out where the eye should start on the screen, finding one object among many, decoding layouts, and differentiating among visually coded interface elements (such as list items with different colors)
- **Physical work**—Keystrokes, mouse movements, gestures (click, drag, double-click), switching between input modes, and number of clicks required to navigate

When implementation-model thinking is applied to digital products, these four types of work are seldom minimized for users—quite the opposite, in fact. The result is software that, in effect, charges its users a tax, or **excise**, of cognitive and physical effort every time it is used.

In the physical world, mandatory tasks that don't immediately satisfy our goals are sometimes unavoidable. For example, when we get up late on a workday and need to get to the office quickly, we must open the garage door, get in the car, start the motor, back out, and close the garage door before we even begin the forward motion that will take us to our destination. These actions support the physicality of the automobile rather than getting us to the destination faster.

If we had *Star Trek* transporters instead, we'd dial up our destination and teleport there instantaneously—no garages, no motors, no traffic lights. Digital products, much like our fictional transporter, don't necessarily need to have the same kind of roadblocks that stand in the way of our goals in the physical world. But implementation-model design often makes it seem that way to users.

Goal-Directed Tasks versus Excise Tasks

Any large task, such as driving to the office, involves many smaller tasks. Some of these tasks work directly toward achieving the goal; these are tasks like steering down the road toward your office. *Excise tasks*, on the other hand, don't contribute directly to reaching the goal, but instead represent extra work that satisfies either the needs of our tools or those of outside agents as we try to achieve our objectives.

In this example, the excise tasks are pretty clear. Opening the garage door is something we do for the car, not for us, and it doesn't move us toward our destination the way the accelerator pedal and steering wheel do. Stopping at red lights is something imposed on us by our society that, again, doesn't help us achieve our true goal. (In this case, it does help us achieve a related goal of arriving *safely* at our office.) A tune-up helps keep the car running well, but it doesn't get us anywhere quickly while we're doing it.

Software, too, has a pretty clear dividing line between *goal-directed tasks* and excise tasks. Like automobiles, some software excise tasks are trivial, and performing them is no great hardship. On the other hand, other software excise tasks are as obnoxious as fixing a flat tire. Installation leaps to mind here, as do such excise tasks as configuring networks and backing up files.

Types of Excise

The problem with excise tasks is that the effort we expend doing them doesn't go directly toward accomplishing our goals. Where we can eliminate excise tasks, we make people more effective and productive, ultimately creating better usability and a better user experience.

> **DESIGN PRINCIPLE**   *Eliminate excise wherever possible.*

The existence of excise in user interfaces is a primary cause of user dissatisfaction with software-enabled products. It behooves every designer and product manager to be on the lookout for interaction excise in all its forms and to take the time and energy to see that it is eliminated from their products.

Navigational excise

Navigation through the functions or features of a digital product is largely excise. Except in the case of games where the *goal* is to navigate successfully through a maze of obstacles, the work that users are forced to do to get around in software and on websites is seldom aligned with their needs, goals, and desires. (However, well-designed navigation can be an effective way to instruct users about what is available to them, which is better aligned with their goals.)

Unnecessary or difficult navigation is a major frustration to users. In fact, in our opinion, poorly designed navigation presents one of the largest and most common problems in the usability of interactive products—mobile, desktop, web, or otherwise. It is also the place where the developer's implementation model typically is made most apparent to users.

Navigation through software occurs at multiple levels:

- Across multiple windows, views, or pages
- Across multiple panes or frames within a window, view, or page
- Across tools, commands, or menus
- Within information displayed in a pane or frame (such as scrolling, panning, zooming, following links)

We find it useful to think in terms of a broad definition of navigation: *any action that takes the user to a new part of the interface or that requires him or her to locate objects, tools, or data elsewhere in the system*. When we start thinking about such actions as navigation, it becomes clear that they are excise and therefore should be minimized or, if possible, eliminated. The following sections discuss each of these types of navigation in more detail.

Navigation across multiple screens, views, or pages

Moving across multiple application views or pages is perhaps the most disorienting kind of navigation for users. It involves a gross shifting of attention that disrupts the user's flow and forces him into a new context. The act of navigating to another window also often means that the contents of the original window are partly or completely obscured. On the desktop, it means that the user needs to worry about window management, an excise task that further disrupts his flow. If users must constantly shuttle between windows to achieve their goals, their disorientation and frustration levels will rise, they will become distracted from the task at hand, and their effectiveness and productivity will drop.

If the number of windows is large enough, the user will become sufficiently disoriented that he may experience *navigational trauma*: He gets lost in the interface. Sovereign posture applications (discussed in Chapter 9) can avoid this problem by placing all main interactions in a single primary view, which may contain multiple independent panes.

Navigation between panes

Windows or views can contain multiple panes—adjacent to each other and separated by splitters (see Chapter 20) or stacked on top of each other and denoted by tabs. Adjacent panes can solve many navigation problems, because they provide useful supporting functions, links, or data on the screen in close reach of the primary work or display area. This reduces navigation to almost nil. If objects can be dragged between panes, those panes should be adjacent.

Problems arise when adjacent supporting panes become too numerous or are not placed on the screen in a way that matches users' work flows. Too many adjacent panes results in visual clutter and confusion: Users do not know where to go to find what they need. Also, crowding forces scrolling—another navigational hit. Navigation within the single screen thus becomes a problem. Some web portals, trying to be everything to everyone, have such navigational problems.

In some cases, depending on user work flows, tabbed panes can be appropriate. Tabbed panes bring with them a level of navigational excise and potential for user disorientation

because they obscure what was on the screen before the user navigated to them. However, this idiom is appropriate for the main work area when multiple documents or independent views of a document are required (such as in Microsoft Excel; see Figure 12-1).
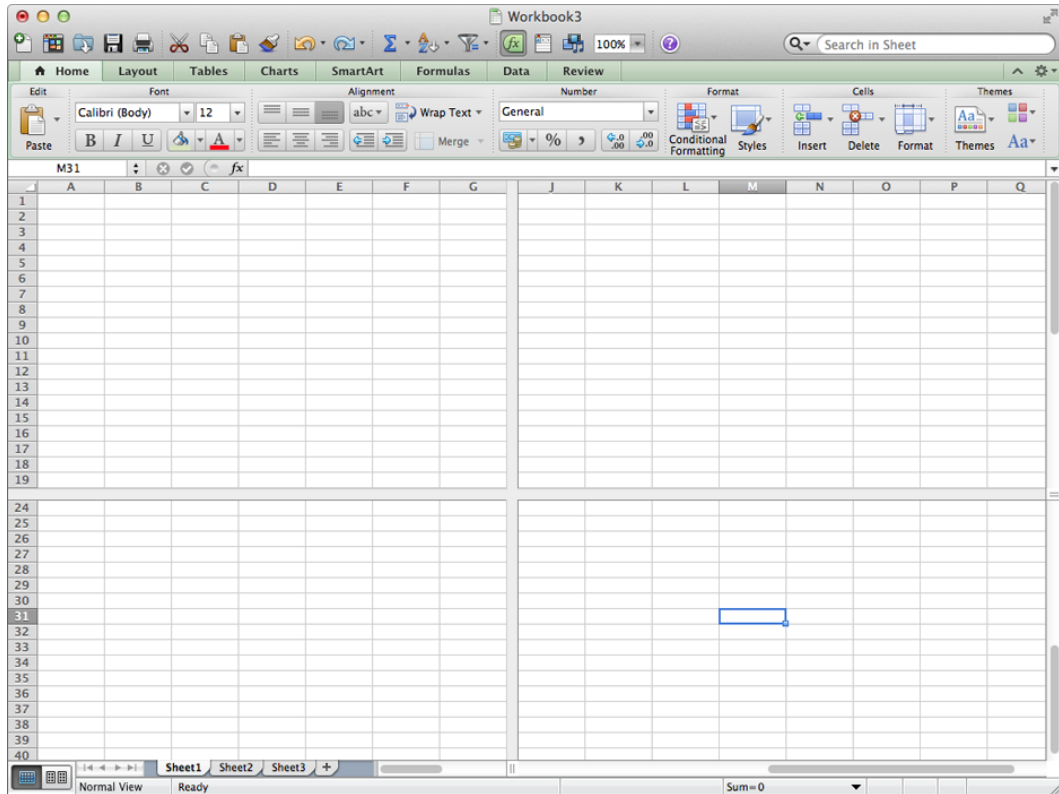
Figure 12-1: Microsoft Excel makes use of tabbed panes (visible in the lower left) to let users navigate between related worksheets. Excel also makes use of splitters to provide adjacent panes for viewing multiple, distant parts of a single spreadsheet without constant scrolling. Both these idioms help reduce navigational excise for Excel users.

Some developers use tabs to break complex product capabilities into smaller chunks. They reason that using these capabilities will somehow become easier if the functionality is cut into bite-sized pieces. Actually, putting parts of a single facility onto separate panes increases excise and decreases users' understanding and orientation.

The use of tabbed screen areas is a space-saving mechanism and is sometimes necessary to fit all the required information and functions into a limited space. (Settings dialogs are a classic example. We don't think anyone is interested in seeing all the settings for a sophisticated application laid bare in a single view.) In most cases, though, the use of tabs creates significant navigational excise. It is rarely possible to accurately describe the contents of a tab with a succinct label (though in a pinch, *rich visual modeless feedback*

on tabs can help—see Chapter 15). Therefore, users must click through each tab to find the tool or piece of information they are looking for.

Tabbed panes can be appropriate when there are multiple supporting panes for a primary work area that are not used at the same time. The support panes can be stacked, and the user can choose the pane suitable for his current tasks, which is only a click away. A classic example involves the color mixer and swatches area in Adobe Illustrator, as shown in Figure 12-2. These two tools are mutually exclusive ways of selecting a drawing color, and users typically know which is appropriate for a given task.
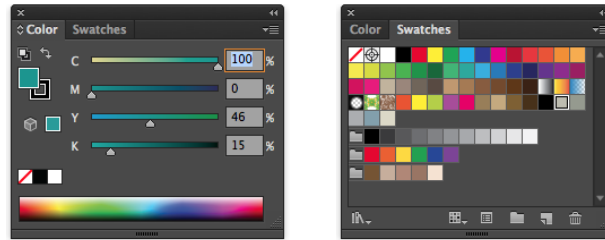
Figure 12-2: Tabbed palettes in Adobe Illustrator allow users to switch between the mixer and swatches, which provide alternative mechanisms for picking a color.

Navigation between tools and menus

Another important and overlooked form of navigation results from a user's need to use different tools, palettes, and functions. Spatial organization of these within a pane or window is critical to minimizing extraneous mouse movements that, at best, could result in user annoyance and fatigue and, at worst, could result in repetitive stress injury. Tools that are used frequently and in conjunction with each other should be grouped spatially and also should be immediately available. Menus require more navigational effort on the part of users because their contents are not visible prior to clicking. Frequently used functions should be provided in toolbars, palettes, or the equivalent. Menu use should be reserved for infrequently accessed commands. (We discuss organizing controls again later in this chapter, and we discuss toolbars in depth in Chapter 18.)

Adobe Photoshop exhibits some undesirable behaviors in how it forces users to navigate between palette controls. For example, the Paint Bucket tool and the Gradient tool each occupy the same location on the tool palette. You must select between them by clicking and holding the visible control, which opens a menu, as shown in Figure 12-3. However, both are fill tools, and if both are used frequently, it would be better to place each of them on the palette next to each other to avoid that frequent, flow-disrupting tool navigation.
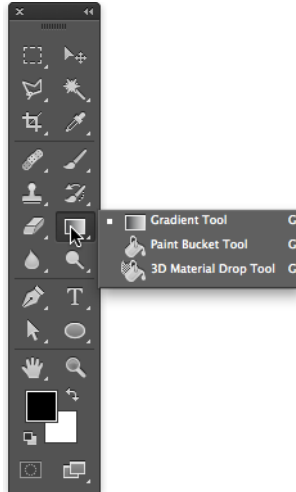
Figure 12-3: In Adobe Photoshop, the Paint Bucket tool is hidden in a combo icon button (see Chapter 21) on its tool palette. Even though users make frequent use of both the Gradient tool and the Paint Bucket tool, they are forced to access this menu anytime they need to switch between these tools.

Navigation of information

Information, or the content of panes or windows, can be navigated using several methods: scrolling (panning), linking (jumping), and zooming. The first two methods are common: Scrolling is ubiquitous in most software, and linking is ubiquitous on the web (although increasingly, linking idioms are being adopted in non-web applications). Zooming is used primarily to visualize 3D and detailed 2D data.

Scrolling is often a necessity, but the need for it should be minimized when possible. Often there is a trade-off between paging and scrolling information: You should understand your users' mental models and work flows to determine what is best for them.

In 2D visualization and drawing applications, vertical and horizontal scrolling are common. These kinds of interfaces benefit from a thumbnail map to ease navigation. We'll discuss this technique as well as other visual signposts later in this chapter.

Linking is the critical navigational paradigm of the web. Because it is a visually dislocating activity, extra care must be taken to provide visual and textual cues that help orient users.

Zooming and *panning* are navigational tools for exploring 2D and 3D information. These methods are appropriate when creating 2D or 3D drawings and models or for exploring

representations of real-world 2D and 3D environments (architectural walkthroughs, or topographic maps, for example). They can fall short when they are used to examine arbitrary or abstract data presented in more than two dimensions. Some information visualization tools use zoom to mean "Display more attribute details about objects"—a logical rather than spatial zoom. As the view of the object enlarges, attributes (often textual) appear superimposed over its graphical representation. This technique works great when the attributes in question are tightly associated with spatial data, such as that employed in Google Maps (see Figure 12-4). But for abstract data spaces, this kind of interaction is almost always better served through an adjacent supporting pane that displays the properties of selected objects in a more standard, readable form.
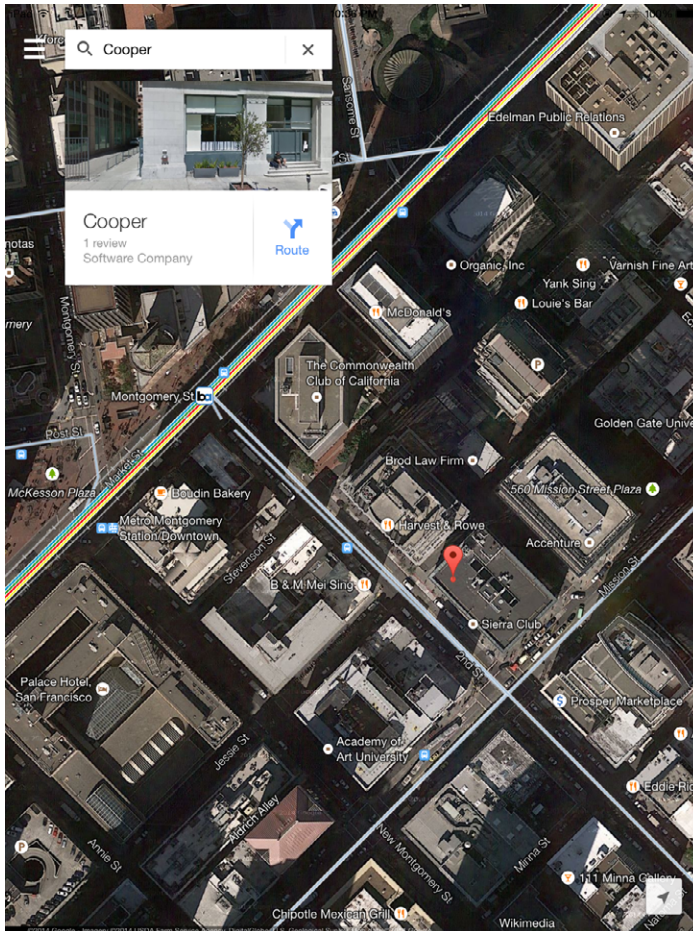
Figure 12-4: The Google Maps app makes excellent use of a combination of spatial and logical zoom. As the user physically zooms in by spreading his fingers apart on the map, location details such as transit lines, traffic congestion, street names, and places of business also come into view. Zoom usually works best when applied to concrete rather than abstract data spaces, such as maps.

Panning and zooming, especially when paired, create navigational difficulties for users. Although this situation is improving due to the prevalence of online maps and easy to grasp gestural interfaces, it is still possible for people to get lost using virtual spaces. Humans are not used to moving in unconstrained 3D environments, and they have difficulty perceiving 3D properly when it is projected on a 2D screen. (See Chapter 18 for more on 3D manipulation.)

Skeuomorphic excise

We are experiencing an incredible transformation from the age of industrial, mechanical artifacts to an age of digital, information objects. It is only natural for us, then, to try to draw on the models and forms of an earlier era that we are comfortable with and use them in this new, less certain one. As the history of the industrial revolution shows, the fruits of new technology can often only be expressed at first with the language of an earlier technology. For example, we called railroad engines iron horses and automobiles horseless carriages. Unfortunately, this imagery and language color our thinking more than we might admit.

Naturally, we tend to use old-style mechanical representations in our new digital environments, a practice called *skeuomorphism*. Sometimes this appropriation of the old is valid because the function is identical, even if the underlying technology is different. For example, when we translate the process of typing with a typewriter into doing word processing on a computer, we are using a mechanical representation of a common task. Typewriters used little metal tabs to rapidly move the carriage several spaces until it came to rest on a particular column. The process, as a natural outgrowth of the technology, was called tabbing or setting tabs. Word processors also have tabs because their function is the same; whether you are working on paper rolled around a platen or on images on a video screen, you need to rapidly slew to a particular margin offset.

More often, however, mechanical representations shouldn't be translated verbatim into the digital world. We encounter problems when we bring our familiar mechanical artifacts into software. These representations result in excise and unnecessarily limit interactions that could be far more efficient than those allowed for by the old models.

Mechanical procedures are usually easier to perform by hand than they are with digital products. Consider a simple contact list. If it is faithfully rendered onscreen like a little bound book, it will be much more complex, inconvenient, and difficult to use than the physical address book. The physical address book, for example, stores names in alphabetical order by last name. But what if you want to find someone by her first name? The mechanical artifact doesn't help you: You have to scan the pages manually. The faithfully replicated digital version wouldn't search by first name either. The difference is that,

on the computer screen, you lose many subtle visual and tangible cues offered by the paper-based book (bent page corners, penciled-in notes). Meanwhile, scrollbars, swipe-to-delete, and navigational drilldowns are harder to use, harder to visualize, and harder to understand than simply flipping pages.

Designers paint themselves into a corner when they rely on slavishly skeuomorphic metaphors. Visual metaphors such as desktops with telephones, copy machines, staplers, and fax machines—or file cabinets with folders in drawers—may make it easy to understand the relationships between interface elements and behaviors. But after users learn these fundamentals, managing the metaphor becomes an exercise in excise. (For more discussion on the limitations of visual metaphors, see Chapter 13.)

Screen real estate consumed by skeuomorphic representations is also excessive, particularly in sovereign posture applications, where maximizing screen space for content rather than UI chrome is of primary importance. The little telephone that so charmingly told us how to dial on that first day long ago is now just a barrier to quick communication.

It's all too easy to fall into the trap of skeuomorphic excise in the name of user friendliness. Apple's iOS veered uncomfortably in the direction of skeuomorphism for versions 4, 5, and 6, but it seems to have finally snapped out of it in iOS 7, as shown in Figure 12-5.

Figure 12-5: In iOS 6 (left), Apple indulged in some excesses of skeuomorphism that seem to have been purged in iOS 7 (right).

Modal excise

The previous chapter introduced the concept of *flow*, whereby a person enters a highly productive mental state by working in harmony with her tools. Flow is a natural state, and people enter it without much prodding. It takes some effort to break into flow after someone has achieved it. Interruptions like a ringing telephone will do it, as will a modal error message or confirmation dialog. Some interruptions are unavoidable, but interrupting the user's flow for no good reason is *stopping the proceedings with idiocy* and is one of the most disruptive forms of excise.

> **DESIGN PRINCIPLE**
>
> *Don't stop the proceedings with idiocy.*

Poorly designed software makes assertions that no self-respecting individual would ever make. It states unequivocally, for example, that a file doesn't exist merely because the software is too stupid to look for the file in the right place, and then it implicitly blames *you* for losing it! An application cheerfully executes an impossible query that hangs up your system until you decide to reboot. Users view such software behavior as idiocy, and with just cause.

Errors, notifiers, and confirmation messages

There are probably no more prevalent excise elements than error message and confirmation message dialogs. These are so ubiquitous that eradicating them takes a lot of work. In Chapter 15, we discuss these issues at length, but for now, suffice it to say that they are high in excise and should be eliminated from your applications whenever possible.

The typical modal error message is unnecessary. It either tells the user something he doesn't care about or demands that he fix a situation that the application can and should usually fix just as well. Figure 12-6 shows an error message box displayed by Adobe Illustrator 6 when the user tries to save a document. We're not exactly sure what it's trying to tell us, but it sounds dire.

The message stops an already annoying and time-consuming procedure, making it take even longer. A user cannot fetch a cup of coffee after telling the application to save his artwork, because he might return only to see the function incomplete and the application mindlessly holding up the process. We discuss how to eliminate these sorts of error messages in Chapter 21.
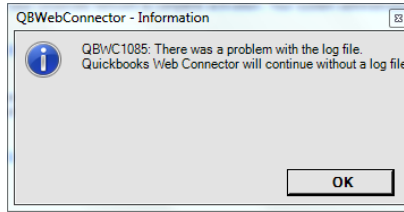
Figure 12-6: This ugly, useless error message box stops the proceedings with idiocy. You can't verify or identify what it tells you, and it gives you no options for responding other than to admit your own culpability by clicking OK. This message comes up only when the application is saving—when you have entrusted it to do something simple and straightforward. The application can't even save a file without help, and it won't tell you what help it needs!

Figure 12-7 shows another frustrating example, this time from Microsoft Outlook.

Figure 12-7: Here is a horrible confirmation box that stops the proceedings with idiocy. If the application is smart enough to detect the difference, why can't it correct the problem itself? The options the dialog offers are scary. It is telling you that you can explode one of two boxes: One contains garbage, and the other contains the family dog—but the application won't say which is which. And if you click Cancel, what does that mean? Will it still explode your dog?

This dialog is asking you to make an irreversible and potentially costly decision based on no information whatsoever! If the dialog occurs just after you changed some rules, doesn't it stand to reason that you want to keep them? And if you don't, wouldn't you like a bit more information, such as exactly what rules are in conflict and which of them are the more recently created? You also don't have a clear idea what happens when you click Cancel. Are you canceling the dialog and leaving the rules mismatched? Are you discarding recent changes that led to the mismatch? The kind of fear and uncertainty that this poorly designed interaction arouses in users is completely unnecessary. We discuss how to improve this kind of situation in Chapter 21.

Making users ask permission

Back in the days of command lines and character-based menus, interfaces indirectly offered services to users. If you wanted to change an item, such as your address, first you had to ask the application for permission to do so. The application would then display a screen where you could change your address. Asking permission is pure excise, and unfortunately things haven't changed much. If you want to change one of your saved addresses on Amazon.com, you have to click a button and go to a different page. If you want to change a displayed value, you should be able to change it right there. You shouldn't have to ask permission or go to a different room.

> **DESIGN PRINCIPLE**
>
> *Don't make users ask permission.*

As in the preceding example, many applications have one place where the values (such as filenames, numeric values, and selected options) are displayed for output and another place where user input to them is accepted. This follows the implementation model, which treats input and output as different processes. A user's mental model, however, doesn't recognize a difference. He thinks, "There is the number. I'll just click it and enter a new value." If the application can't accommodate this impulse, it is needlessly inserting excise into the interface. If the user can modify options, he should be able to do so right where the application displays them.

> **DESIGN PRINCIPLE**
>
> *Allow input wherever you have output.*

The opposite of asking permission can be useful in certain circumstances. Rather than asking the application to launch a dialog, the user tells a dialog to go away and not return. In this way, the user can make an unhelpful dialog stop badgering him, even though the application mistakenly thinks it is helping. Microsoft Windows now makes heavy use of this idiom. (If a beginner inadvertently dismisses a dialog and can't figure out how to get it back, he may benefit from another easy-to-identify safety-net idiom in a prominent place—a Help menu item saying "Bring back all dismissed dialogs," for example.)

Stylistic excise

Users must perform visual work to decode onscreen information, such as finding a single item in a list, figuring out where to begin reading on a screen, or determining which elements on a screen are clickable and which are merely decoration.

A significant source of visual work is the use of overly stylized graphics and interface elements (see Figure 12-8). Visual style can certainly create mood and reinforce brand, but it shouldn't do so at the expense of utility and usability by forcing users to decode visual elements to understand which represent controls and critical information and which are merely ornamental. The use of visual style, at least in apps geared toward productivity rather than entertainment, should support the clear communication of information and interface behavior.
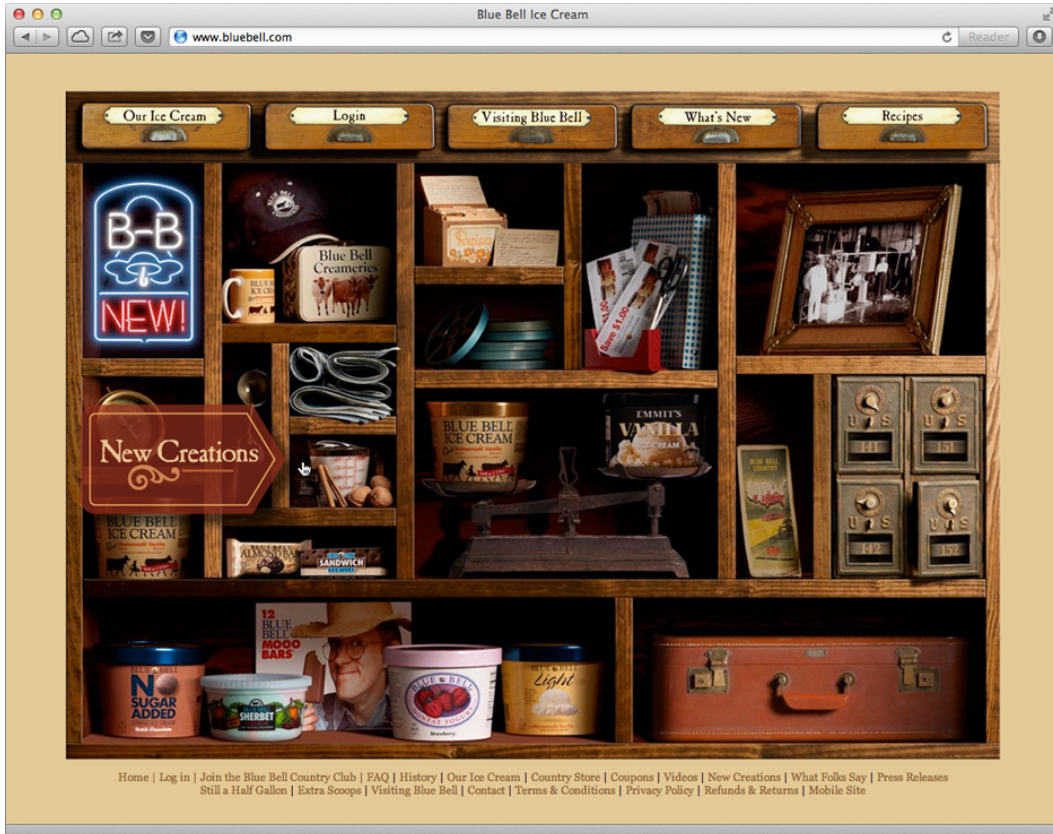
Figure 12-8: The home page of Blue Bell Creameries provides a good example of visual excise. Text is highly stylized and doesn't follow a layout grid. It's difficult for users to differentiate between décor and navigational elements. This requires users to do visual work to interact with the site. This isn't always a bad thing—just the right amount of the right kind of work can be a source of entertainment (as with games and puzzles).

For more discussion on striking the right balance to create effective visual interface designs, see Chapter 17.

Excise Is Contextual

One man's (or persona's) goal-directed task may be another's excise task in a different context. In general, a function or action is excise if it is forced on the user rather than made available at his discretion. An example of this kind of function is window management. The only way to determine whether a function or behavior such as this is excise is by comparing it to personas' goals. If a significant persona needs to see two applications at a time on the screen to compare or transfer information, the ability to configure the main windows of the applications so that they share the screen space is not excise. If your personas don't have this specific goal or need, the work required to configure the main window of either application *is* excise.

Excise may also vary by software posture (see Chapter 9). Users of transient posture applications often require some instruction to use the product effectively. Allocating screen real estate to this effort typically does not contribute to excise in the same way as it does in sovereign posture applications. Transient posture applications aren't used frequently, so their users need more assistance understanding what the application does and remembering how to control it. For sovereign posture applications, however, the slightest excise becomes agonizing over time.

However, some types of actions are almost *always* excise and should be eliminated under all circumstances. These include most hardware-management tasks that the software could handle itself (if a few more design and engineering cycles were spent on it). Any demands for such information should be struck from user interfaces and replaced with more silently intelligent application behavior behind the scenes.

Eliminating Excise

Navigational excise is easily the most prevalent type of excise found in digital products and thus is one of the best places to start eliminating it. There are many ways to begin improving (eliminating, reducing, or speeding up) navigation in your applications, websites, and devices. Here are the most effective:

- Reduce the number of places to go.
- Provide signposts.
- Provide overviews.
- Properly map controls to functions.
- Avoid hierarchies.
- Don't replicate mechanical models.

We'll discuss these in detail next.

Reduce the number of places to go

The most effective method of improving navigation sounds quite obvious: Reduce the number of places to which one must navigate. These "places" include modes, forms, dialogs, pages, windows, and screens. If the number of modes, pages, or screens is kept to a minimum, people's ability to stay oriented increases dramatically. In terms of the four types of navigation presented earlier, this directive means you should do the following:

- Keep the number of windows and views to a minimum. One full-screen window with two or three views is best for many users. Keep dialogs, especially modeless dialogs, to a minimum. Applications, websites, or mobile apps with dozens of distinct types of pages, screens, or forms are difficult to navigate.

- Limit the number of adjacent panes in your interface to the minimum number needed for users to achieve their goals. In sovereign posture applications, three panes is a good thing to shoot for, but there are no absolutes here—in fact, many applications require more. On web pages, anything more than two navigation areas and one content area begins to get busy. On tablet apps, two panes is typical.

- Limit the number of controls to as few as your users really need to meet their goals. Having a good grasp of your users via personas will enable you to avoid functions and controls that your users don't really want or need and that, therefore, only get in their way.

- Minimize scrolling when possible. This means giving supporting panes enough room to display information so that they don't require constant scrolling. Default views of 2D and 3D diagrams and scenes should be such that the user can orient himself without too much panning. Zooming is the most difficult type of navigation for most users (though more straightforward in mobile apps using pinch gestures), so its use should be discretionary, not a requirement.

Many online stores present confusing navigation because the designers have tried to serve everyone with one generic site. If a user buys books but never music from a site, access to the music portion of the site could be deemphasized in the main screen for that user. This makes more room for that user to buy books, and the navigation becomes simpler. Conversely, if he visits his account page frequently, his version of the site should prominently display his account button (or tab).

Provide signposts

In addition to reducing the number of navigable places, another way to enhance users' ability to find their way around is by providing better points of reference—*signposts*. In the same way that sailors navigate by reference to shorelines or stars, users navigate by reference to *persistent objects* placed in a user interface.

Persistent objects, in a desktop world, always include the application's windows. Each application most likely has a main, top-level window. The salient features of that window are also considered persistent objects: menu bars, toolbars, and other palettes or visual features like status bars and rulers. Generally, each window of the interface has a distinctive look that will soon become recognizable.

On the web, similar rules apply. Well-designed websites make careful use of persistent objects that remain constant throughout the shopping experience, especially the top-level navigation bar along the top of the page. Not only do these areas provide clear navigational options, but their consistent presence and layout also help orient customers (see Figure 12-9).
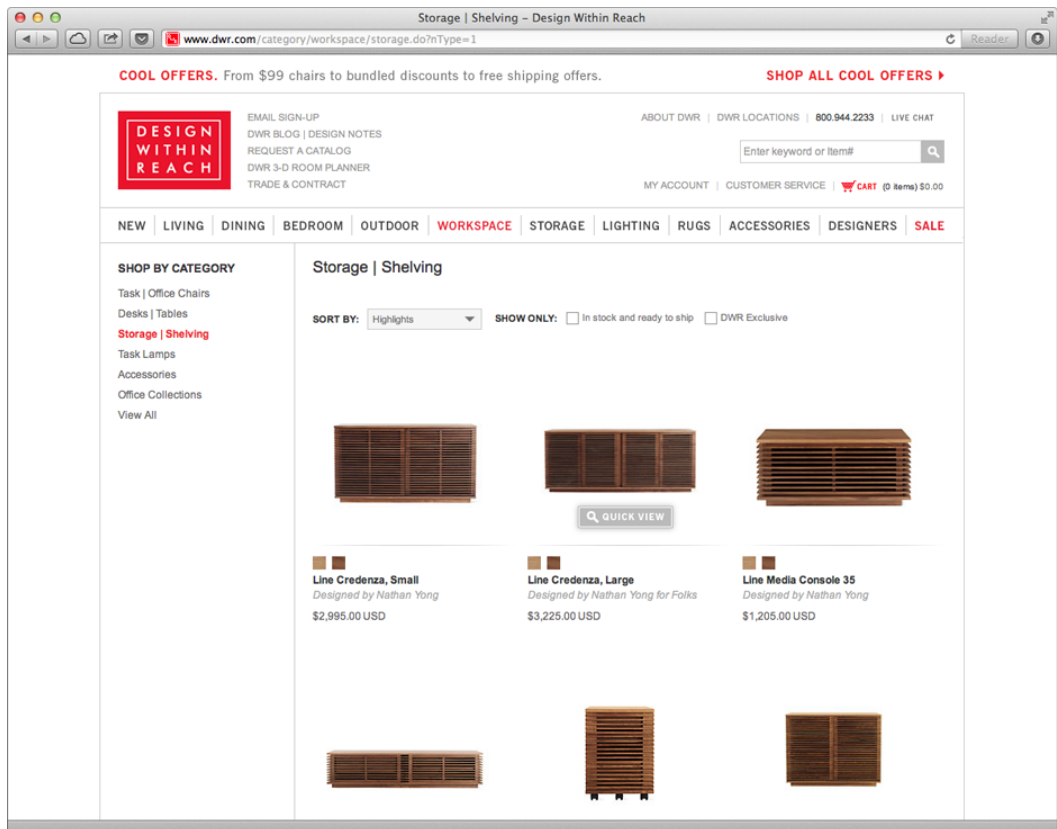
Figure 12-9: The Design Within Reach website makes use of many persistent areas on the majority of its pages, such as the links and search field along the top and the browse tools on the side. These not only help users figure out where they can go but also help keep them oriented.

In devices, similar rules apply to screens, but hardware controls themselves can take on the role of signposts—even more so when they offer visual or tactile feedback about their state. Radio buttons that, for example, light when selected, even a needle's position on a dial, can provide navigational information if integrated appropriately with the software.

Depending on the application, the contents of the application's main window may also be easily recognizable (especially with kiosks and small-screen devices). Some applications may offer a few different views of their data, so the overall aspect of their screens changes depending on the view chosen. A desktop application's distinctive look, however, usually comes from its unique combination of menus, palettes, and toolbars. This means that menus and toolbars must be considered aids to navigation. You don't need a lot of signposts to navigate successfully; they just need to be visible. Needless to say, signposts can't aid navigation if they are removed, so it is best if they are permanent fixtures of the interface (some iOS browsers break this rule slightly by allowing controls to scroll up as the user moves down the page; however, they immediately scroll back down when the user reverses direction—a clever means of bringing controls back into focus as soon as they are needed).

Making each page on a website look just like every other one may maintain visual consistency, but it can, if carried too far, be disorienting. You should use common elements consistently on each page, but by making different rooms look distinctive, you will help orient your users better.

Menus

The most prominent permanent object in a desktop application is the main window and its title and menu bars. Part of the benefit of the menu comes from its reliability and consistency. Unexpected changes to an application's menus can deeply reduce users' trust in them. This is true for menu items as well as for individual menus.

Toolbars

If the application has a toolbar, it should also be considered a recognizable signpost. Because toolbars are idioms for perpetual intermediates rather than for beginners, the strictures against changing menu items don't apply quite as strongly to individual toolbar controls. Removing the toolbar itself is certainly a dislocating change to a persistent object. Although the ability to do so should exist, it shouldn't be offered casually, and users should be protected from accidentally triggering it. Some applications put a control on the toolbar that makes the toolbar disappear! This is a completely inappropriate ejector seat lever.

Other interface signposts

Tool palettes and fixed areas of the screen where data is displayed or edited should also be considered persistent objects that add to the interface's navigational ease. Judicious use of white space and legible fonts is important so that these signposts remain clearly evident and distinctive.

Provide overviews

Overviews serve a purpose similar to signposts in an interface: They help orient users. The difference is that overviews help orient users within the content rather than within the application as a whole. Because of this, the overview area should itself be persistent; its content is dependent on the data being navigated.

Overviews can be graphical or textual, depending on the nature of the content. An excellent example of a graphical overview is the aptly named Navigator palette in Adobe Photoshop, shown in Figure 12-10.
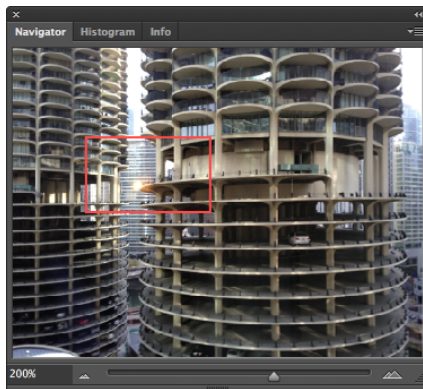
Figure 12-10: On the left, Adobe makes use of an excellent overview idiom in Photoshop: the Navigator palette, which provides a thumbnail view of a large image with an outlined box that represents the portion of the image currently visible in the main display. Not only does the palette provide navigational context, but it can be used to pan and zoom the main display as well. A similar idiom is employed on the right in the Google Finance charting tool, in which the small graph on the bottom provides a big-picture view and context for the zoomed-in view on top.

In the web world, the most common form of overview area is textual: the ubiquitous breadcrumb display (see Figure 12-11). Again, most breadcrumbs provide a navigational aid as well as a navigational control: Not only do they show where in the data structure a visitor is, but they also give him or her tools to move to different nodes in the structure in the form of links. This idiom has lost some popularity as websites have moved from being strictly hierarchical organizations to more associative organizations, which don't lend themselves as neatly to breadcrumbs.
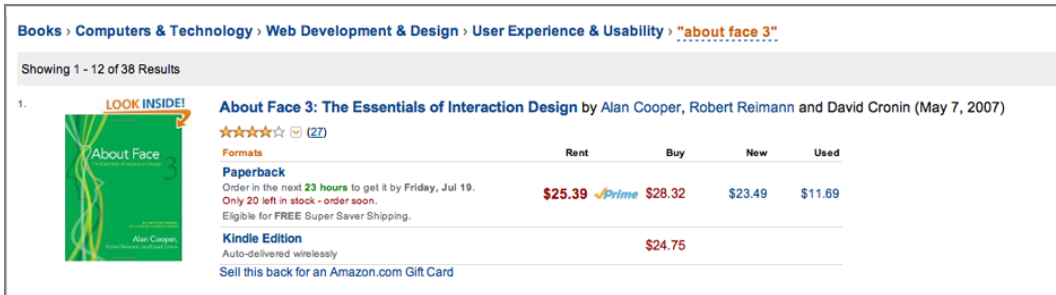
Figure 12-11: A typical breadcrumb display from Amazon.com. Users see where they've been and can click anywhere in the breadcrumb trail to navigate to that link.

A final interesting example of an overview tool is the *annotated scrollbar*, which is most useful for scrolling through text. They make clever use of the linear nature of both scrollbars and textual information to provide location information about the locations of selections, highlights, and potentially many other attributes of formatted or unformatted text. Hints about the locations of these items appear in the "track" that the thumb of the scrollbar moves in, at the appropriate location. When the thumb is over the annotation, the annotated feature of the text is visible in the display. Microsoft Word uses a variation of the annotated scrollbar; it shows the page number and nearest header in a ToolTip that remains active during the scroll, as shown in Figure 12-12.
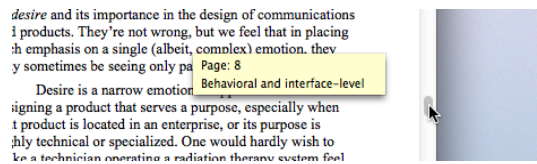
Figure 12-12: An annotated scrollbar from Microsoft Word provides useful context for the user as he or she navigates through a document.

Properly map controls to functions

Mapping describes the relationship between a control, the thing it affects, and the intended result. Poor mapping is evident when a control does not relate visually, spatially, or symbolically to the object it affects. Poor mapping requires users to stop and think about the relationship, breaking flow. Poor mapping of controls to functions increases the cognitive load for users and can result in potentially serious user errors.

Donald Norman provides an excellent example of mapping problems from the non-digital world in *The Design of Everday Things* (Basic Books, 2002). Almost anyone who cooks has run into the annoyance of a stovetop whose burner knobs do not map appropriately to the burners they control. The typical stovetop, such as the one shown in Figure 12-13, features four burners arranged in a flat square with a burner in each corner. However, the knobs that operate those burners are laid out in a straight line on the front of the unit.
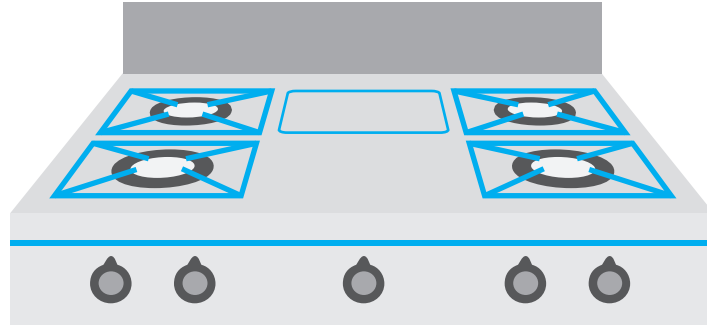
Figure 12-13: A stovetop with poor physical mapping of controls. Does the knob on the far left control the left-front or left-rear burner? Users must figure out the mapping anew each time they use the stovetop.

In this case, we have a *physical mapping* problem. The *result* of using the control is reasonably clear: A burner will heat up when you turn a knob. However, the *target* of the control—which burner will get warm—is unclear. Does twisting the leftmost knob turn on the left-front burner, or does it turn on the left-rear burner? Users must find out by trial and error or by referring to the tiny icons next to the knobs. The unnaturalness of the mapping compels users to figure out this relationship anew every time they use the stove. This cognitive work may become habituated over time, but it still exists, making users prone to error if they are rushed or distracted (as people often are while preparing meals). In the best-case scenario, users feel stupid because they've twisted the wrong knob, and their food doesn't get hot until they notice the error. In the worst-case scenario, they might accidentally burn themselves or set fire to the kitchen.

The solution requires moving the stovetop knobs so that they better suggest which burners they control. The knobs don't have to be laid out in exactly the same pattern as the burners, but they should be positioned so that the target of each knob is clear. The stovetop shown in Figure 12-14 is a good example of an effective mapping of controls.

In this layout, it's clear that the upper-left knob controls the upper-left burner. The placement of each knob visually suggests which burner it will turn on. Norman calls this more intuitive layout "natural mapping."

Figure 12-15 shows another example of poor mapping—of a different type. In this case, it is the *logical mapping* of concepts to actions that is unclear.
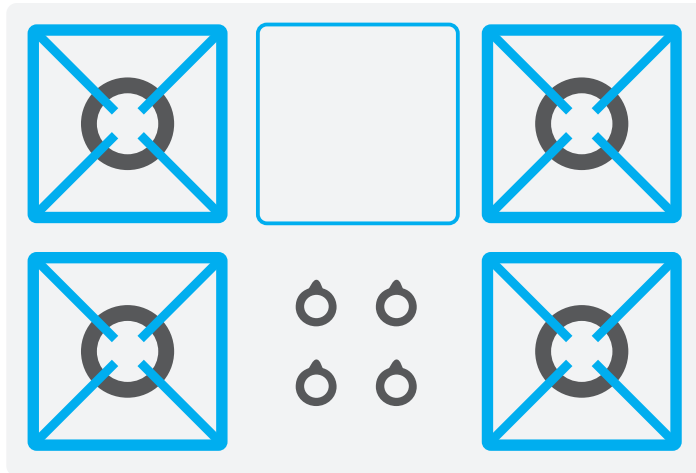
Figure 12-14: Clear spatial mapping. On this stovetop, it is clear which knob maps to which burner, because the spatial arrangement of knobs clearly associates each knob with a burner.
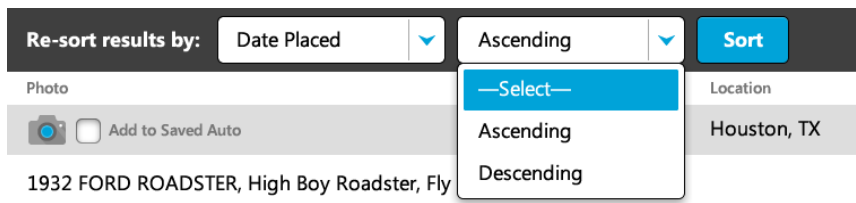
Figure 12-15: An example of a logical mapping problem. If the user wants to see the most recent items first, does she choose Ascending or Descending? These terms don't map well to how users conceive of time.

This website uses a pair of drop-down menus to sort a list of search results by date. The selection in the first drop-down determines the choices present in the second. When **Re-sort results by: Date Placed** is selected in the first menu, the second drop-down presents the options **Ascending** and **Descending**.

Unlike the poorly mapped stovetop knobs, the *target* of this control is clear—the drop-down menu selections affect the list below them. However, the *result* of using the control is unclear: Which sort order will the user get if she chooses Ascending?

The terms chosen to communicate the date-sorting options make it unclear what users should choose if they want to see the most recent items first in the list. **Ascending** and **Descending** do not map well to most users' mental model of time. People don't think of dates as ascending or descending; rather, they think of dates and events as being recent or ancient. A quick fix to this problem is to change the wording of the options to **Most recent first** and **Oldest first**, as shown in Figure 12-16.
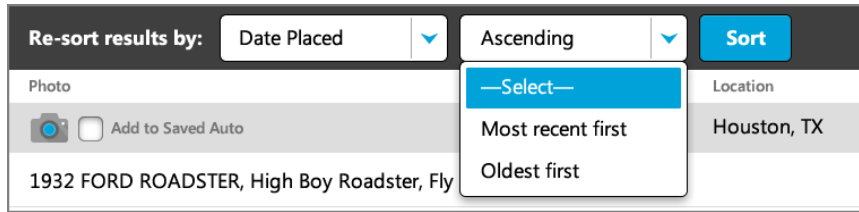
Figure 12-16: Clear, logical mapping. "Most recent" and "Oldest" are terms that users can easily map to time-based sorting.

Whether you make appliances, mobile apps, desktop applications, or websites, your product may have mapping problems. Mapping is an area where attention to detail pays off. You can measurably improve a product by seeking out and fixing mapping problems, even if you have very little time to make changes. The result is a product that is easier to understand and more pleasurable to use.

Avoid hierarchies

Hierarchies are one of the developer's most durable tools. Much of the data inside applications, along with much of the code that manipulates it, is in hierarchical form. For this reason, many developers present hierarchies (the implementation model) in user interfaces. Early menus, as we've seen, were hierarchical. But abstract hierarchies are very difficult for users to successfully navigate, except where they're based on user mental models and the categories are truly mutually exclusive. This truth is often difficult for developers to grasp because they themselves are so comfortable with hierarchies.

Most humans are familiar with hierarchies in their business and family relationships, but hierarchies are not natural concepts for most people when it comes to storing and retrieving arbitrary information. Most mechanical storage systems are simple, composed of either a single sequence of stored objects (like a bookshelf) or a series of sequences, one level deep (like a file cabinet). This method of organizing things into a single layer of groups is extremely common and can be found everywhere in your home and office. Because it never exceeds a single level of nesting, we call this storage paradigm *monocline grouping*.

Developers are comfortable with nested systems, in which an instance of an object is stored in another instance of the same object. Most other humans have a difficult time with this idea. In the mechanical world, complex storage systems, by necessity, use different mechanical form factors at each level. In a file cabinet, you never see folders inside folders or file drawers inside file drawers. Even the dissimilar nesting of folder-inside-drawer-inside-cabinet rarely exceeds two levels of nesting. In the current desktop metaphor used by most window systems, you can nest folder within folder ad infinitum. It's no wonder most computer neophytes get confused when confronted with this paradigm.

Most people store their papers (and other items) in a series of stacks or piles based on some common characteristic: The Acme papers go here; the Project M papers go there; personal stuff goes in the drawer. Donald Norman (1994) calls this a *pile cabinet*. Only inside computers do people put the Project M documents inside the Active Clients folder, which in turn is stored inside the Clients folder, stored inside the Business folder.

Computer science gives us hierarchical structures as tools to solve the very real problems of managing massive quantities of data. But when this implementation model is reflected in the represented model presented to users (as discussed in Chapter 1), they get confused, because it conflicts with their mental model of storage systems. Monocline grouping is the mental model people typically bring to software. Monocline grouping is so dominant outside the computer that interaction designers violate this model at their peril.

Monocline grouping is an inadequate system for physically managing the large quantities of data commonly found on computers, but that doesn't mean it isn't useful as a *represented model*. The solution to this conundrum is to render the structure as the user imagines it—as monocline grouping—but to provide the search and access tools that only a deep hierarchical organization can offer. In other words, rather than forcing users to navigate deep, complex tree structures, give them tools to *bring appropriate information to themselves*. We'll discuss some design solutions that help make this happen in Chapter 14.

Don't replicate Mechanical-Age models

As already discussed, skeuomorphic excise—resulting from an unreflective replication of Mechanical-Age actions in digital interfaces—adds excise, navigational and otherwise.

It makes sense to spend some time rethinking products and features that are translated from the pre-digital world. How can the new, digital version be streamlined and adapted to take full advantage of the digital environment? How can excise be eliminated and smarts brought to bear?

Take the desk calendar. In the non-digital world, calendars are made of paper and are usually divided into a one-month-per-page format. This is a reasonable compromise based on the size of paper, file folders, briefcases, and desk drawers.

Digital products with representations of calendars are quite common, and they almost always display one month at a time. Even if they can show more than one month, as Outlook does, they almost always display days in discrete one-month chunks. Why?

Paper calendars show a single month because they are limited by the size of the paper, and a month is a convenient breaking point. High-resolution digital displays are not so constrained, but most designers copy the mechanical artifact faithfully, as shown in Figure 12-17.
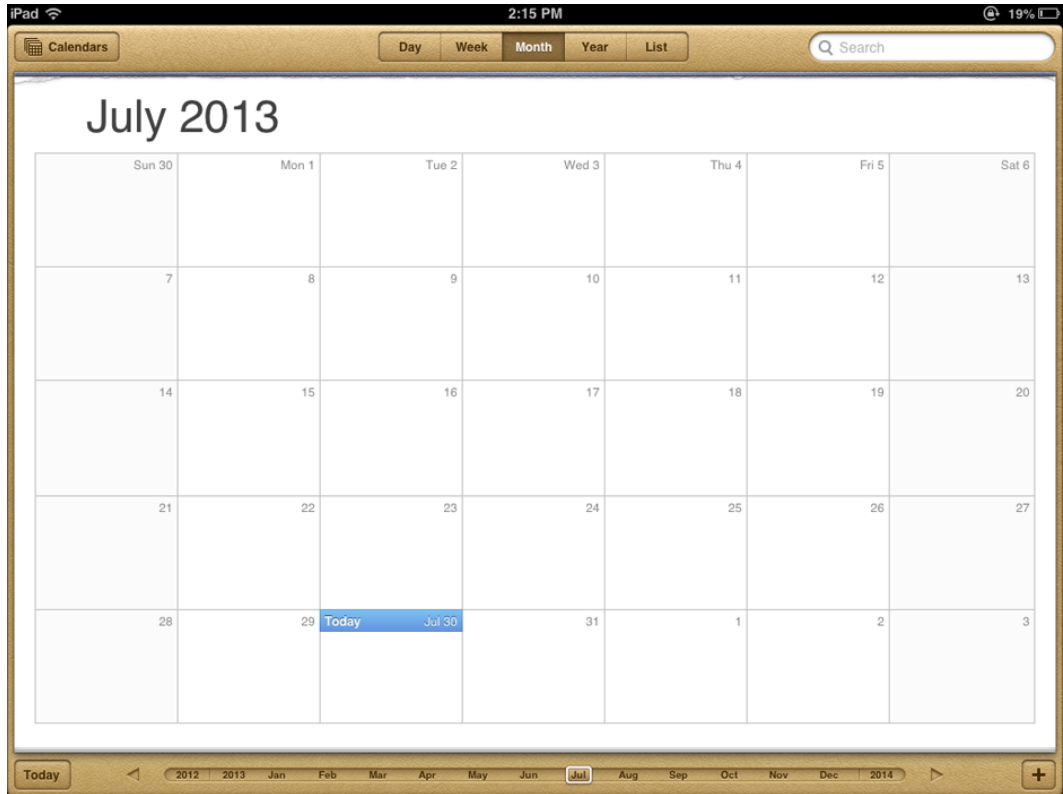
Figure 12-17: The ubiquitous calendar is so familiar that we rarely stop to apply Information-Age sensibilities to its design on the screen. Calendars were originally designed to fit on stacked sheets of paper, not interactive digital displays. How would you redesign a digital calendar? Which of its aspects are artifacts of its old, Mechanical-Age platform?

On an interactive screen, the calendar could easily be a continuously scrolling sequence of days, weeks, or months, as shown in Figure 12-18. Scheduling something from August 28 to September 4 would be simple if weeks were contiguous instead of broken up by the arbitrary monthly division.

Similarly, the grid pattern in digital calendars is almost universally a fixed size. Why can't the width of columns of days or the height of rows of weeks be adjustable like a spreadsheet? Certainly you'd want to adjust the sizes of your weekends to reflect their relative importance in relation to your weekdays. If you're a businessperson, your working-week calendar would demand more space than a vacation week. The adjustable grid interface idiom is well known—every spreadsheet in the world uses it—but the mechanical representations of calendars are so firmly entrenched that we rarely see apps that deviate from them.
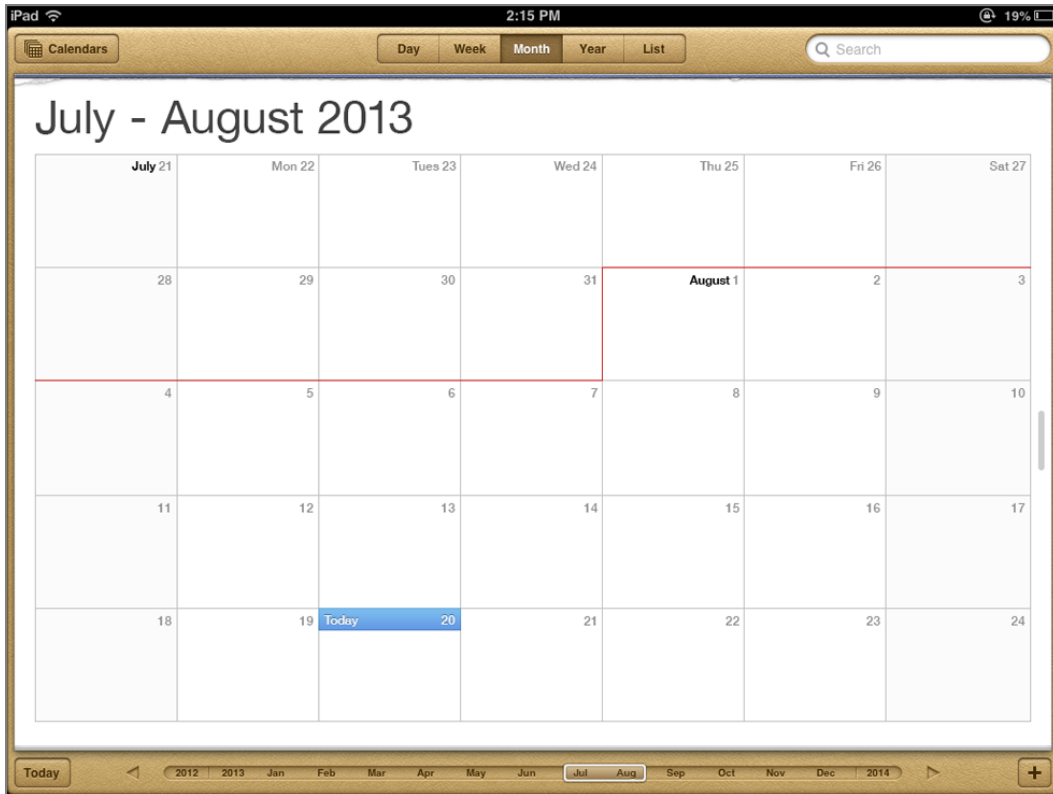
Figure 12-18: Scrolling is a familiar task to computer users. Why not replace the page-oriented calendar with a scrolling one to make it better? This perpetual calendar can do everything the old one can, and it also solves the mechanical-representation problem of scheduling across monthly boundaries. Don't drag old limitations onto new platforms out of habit. What other improvements can you think of?

The designer of the software shown in Figure 12-17 probably thought of calendars as canonical objects that couldn't be altered from the familiar. Surprisingly, most time-management software handles time internally—in its implementation model—as a continuum, and renders it as discrete months only in its user interface—its represented model!

Some might argue that the one-month-per-page calendar is better because it is easily recognizable and familiar to users. However, the new digital model isn't all that different from the old paper model, except that it permits the users to do something they couldn't easily do before—schedule across monthly boundaries. People don't find it difficult to adapt to new representations *if they offer a significant improvement.*

Apple flubbed an opportunity to take this approach with their redesigned iOS7 Calendar app. It features a continuous vertically scrolling calendar in month view... but the designers chose to line break at month boundaries, and didn't support a drag gesture to specify a multi-day event. So close, and yet so far.

Paper-style calendars in mobile devices and desktops are mute testimony to how our Mechanical-Age modes of thinking influence our designs. If we don't inform our assumptions about product use with an analysis of user goals, we will end up building excise-ridden software that remains in the Mechanical Age. Better software is based on Information-Age thinking.

Other Common Excise Traps

You should be vigilant in finding and rooting out each small item of excise in your interface. These myriad little extra unnecessary steps can add up to a lot of extra work for users. This list should help you spot excise transgressions:

- Don't force users to go to another window to perform a function that affects the current window.

- Don't force users to remember where they put things in the hierarchical file system.

- Don't force users to resize windows unnecessarily. When a child window pops up on the screen, the application should size it appropriately for its contents. Don't make it big and empty or so small that it requires constant scrolling.

- Don't force users to move windows. If there is open space on the desktop, put the application there instead of directly over some other already open application.

- Don't force users to reenter their personal settings. If the user has ever set a font, color, indentation, or sound, make sure that she doesn't have to do so again unless she wants a change.

- Don't force users to fill in fields to satisfy some arbitrary measure of completeness. If the user wants to omit some details from the transaction entry screen, don't force him to enter them. Assume that he has a good reason for not doing so. In most instances, the completeness of the database isn't worth badgering users over.

- Don't force users to ask permission. This is frequently a symptom of not allowing input in the same place as output.

- Don't ask users to confirm their actions. This requires a robust Undo facility.

- Don't let the user's actions result in an error.

Excise represents the most common and most pernicious barriers to usability and user satisfaction in digital products. Don't let it rear its ugly head in your designs or applications!

METAPHORS, IDIOMS, AND AFFORDANCES

When the first edition of this book was published, interface designers often spoke of finding the right visual and behavioral metaphors on which to base their interface designs. In that decade or two following the introduction of the Apple Macintosh, it was widely believed that filling interfaces with visual representations of familiar objects from the real world would give users a pipeline to easy learning. As a result, designers created interfaces resembling offices filled with desks, file cabinets, telephones, and address books, or pads of paper, or a street with signs and buildings.

With the advent of Android, Windows Phone, and iOS 7, we have officially passed into a post-metaphorical era of interaction design. Gone are the skeuomorphisms and overwrought visual metaphors from the early days of desktop software and handheld devices. Modern device user interfaces (UIs) (and, increasingly, desktop UIs as well) are properly content- and data-centric, minimizing the cognitive footprint of UI controls almost to a fault.

This recent shift away from metaphor was long overdue, and for good reason: Strict adherence to metaphors ties interfaces unnecessarily tightly to the workings of the physical world. One of the most fantastic things about digital products is that the working model presented to users need not be bound by the limitations of physics or the inherent clumsiness of mechanical systems and 3D real-world objects mapped to 2D control surfaces.

User interfaces based on metaphors have a host of other problems as well. There aren't enough good metaphors to go around, they don't scale well, and the users' ability to recognize them is often questionable, especially across cultural boundaries. Metaphors, especially physical and spatial metaphors, have a limited place in the design of most digital products. In this chapter, we discuss the reasons for this, as well as the modern replacements for design based on metaphors.

Interface Paradigms

The three dominant paradigms in the conceptual and visual design of user interfaces are *implementation-centric*, *metaphoric*, and *idiomatic*. The implementation-centric interfaces are based on *understanding* how things actually work under the hood—a difficult proposition. Metaphoric interfaces are based on *intuiting* how things work—a risky method. Idiomatic interfaces, however, are based on *learning* how to accomplish things—a natural, human process.

Historically, the field of interaction design has progressed from a heavy focus on technology (implementation), to an equally heavy focus on metaphor, and, most recently, to a more idiomatic focus. Although many examples of all three types of interface paradigms still are in use today, the most modern, information-centric interface designs in common use on computers, phones, tablets, and other devices are primarily idiomatic in nature.

Implementation-centric interfaces

Implementation-centric user interfaces are still widespread, especially in enterprise, medical, and scientific software. Implementation-centric software shows us, without any hint of shame, precisely how it is built. There is one button per function and one dialog per module of code, and the commands and processes precisely echo the internal data structures and algorithms. The side effect of this is that we must first learn how the software works internally to successfully understand and use the interface. Following the implementation-centric paradigm means user-interface design that is based exclusively on the implementation model.

Clearly, implementation-centric interfaces are the easiest to build. Every time a developer writes a function, he slaps on a bit of user interface to test that function. It's easy to debug, and when something doesn't behave properly, it's easy to troubleshoot. Furthermore, engineers like to know how things work, so the implementation-centric paradigm is very satisfying to them. Engineers prefer to see the virtual gears and levers and valves because this helps them understand what is going on inside the machine. But those artifacts needlessly complicate things for users. Engineers may want to understand the inner workings, but most users don't have either the time or desire. They'd much rather be successful than knowledgeable, a preference that is often hard for engineers to understand.

Most people would rather be successful than knowledgeable.

A close relative of the implementation-centric interface worth mentioning is the "org-chart-centric" interface. This is a common situation in which a product or, most typically, a website is not organized according to how users are likely to think about information. Instead, it is organized by which part of the company or organization owns whatever piece of information the user is looking to access. Such a site typically has a tab or area for each corporate division, and there is a lack of cohesion between these areas. Usually there is no coordinated design between intracorporate fiefdoms in these situations. Similar to the implementation-centric product interface, an org-chart-centric website requires users to understand how a corporation is structured so that they can find the information they are interested in, and that information is often unavailable to those same users.

Metaphoric interfaces

Metaphoric interfaces rely on the real-world connections users make between the visual cues in an interface and its function. Since there was less of a need to learn the mechanics of the software, metaphoric interfaces were a step forward from implementation-centric interfaces. However, the power and utility of heavily metaphoric interfaces were, at least for a time, inflated to unrealistic proportions.

When we talk about a metaphor in the context of user interface and interaction design, we really mean a visual metaphor that signals a function: a picture used to represent the purpose or attributes of a thing. Users recognize the metaphor's imagery. By extension, it is presumed that they can understand the purpose of the thing. Metaphors can range from tiny icons on toolbar buttons to the entire screen on some applications—from a tiny pair of scissors on a button, indicating Cut, to a full-size checkbook in Quicken.

Instinct, intuition, and learning

In the computer industry, and particularly in the user-interface design community, the word *intuitive* is often used to mean *easy to use* or *easy to understand*. This term has become closely associated with metaphorical interfaces.

We do understand metaphors intuitively, but what does that really mean? Webster's Dictionary defines *intuition* like this:

> **in·tu·i·tion** \\ˌin-tü-ˈi-shən\\ *n 1 : quick and ready insight* **2 a** *: immediate apprehension or cognition* **b :** *knowledge or conviction gained by intuition* **c :** *the power or faculty of attaining to direct knowledge or cognition without evident rational thought and inference*

This definition doesn't say much about *how* we intuit something. In reality, no magical quality of "intuitiveness" makes things easy to use. Instead, there are concrete reasons why people grasp some interfaces and not others.

Certain sounds, smells, and images make us respond without any previous conscious learning. When a child encounters an angry dog, she *instinctively* knows that bared teeth signal danger, even without any previous learning. Instinct is a hardwired response that involves no conscious thought.

Examples of instinct in human-computer interaction include how we are startled by unexpected changes in the image on our computer screen, how we find our eyes drawn to a flashing advertisement on a web page, and how we react to sudden noises from our computer or the haptic vibrations of our video-game controller.

Intuition, unlike instinct, works by *inference*, in which we see connections between disparate subjects and learn from these similarities while not being distracted by their differences. We grasp the meaning of the metaphoric elements of an interface because we mentally connect them with *other things we have previously learned in the world*.

You intuit how to use a wastebasket icon, for example, because you once learned how a real wastebasket works, thereby preparing your mind to make the connection years later. You didn't *intuit* how to use the original wastebasket. It was just an easy thing to learn.

Metaphorical interfaces are an efficient way to take advantage of the awesome power of the human mind to make inferences. However, this approach also depends on the idiosyncratic minds of users, which may not have the requisite language, learned experiences, or inferential power necessary to make those connections. Furthermore, metaphorical approaches to interface design have other serious problems, as we shall soon see.

The tyranny of the global metaphor

The most significant problem with metaphors is that they tie our interfaces to Mechanical Age artifacts. An extreme example of this was Magic Cap, the operating system for a handheld communicator. It was introduced by a company called General Magic, founded by Macintosh software gurus Andy Hertzfeld and Bill Atkinson. It was ahead of its time in overall concept, with its remarkably usable touchscreen keyboard and address book nearly 15 years before the iPhone.

Unfortunately, it relied on metaphors for almost every aspect of its interface. You accessed your messages from an inbox or a notebook on a desk. You walked (virtually) down a hallway lined with doors representing secondary functions. You went outside to

access third-party services, which, as shown in Figure 13-1, were represented by buildings on a street. You entered a building to configure a service, and so on.

Relying heavily on a metaphor such as this means that you can intuit the software's basic functions. But the downside is that, after you understand its function, the metaphor adds significantly to the overhead of navigation. You *must* go back out onto the street to configure another service. You *must* go down the hallway and into the game room to play Solitaire. This may be normal in the physical world, but there is no reason for it in the world of software. Why not abandon this slavish devotion to metaphor and give the user *easy* access to functions? It turns out that a General Magic developer later created a bookmarking shortcut facility as a kludgy add-on, but alas, it was too little, too late.
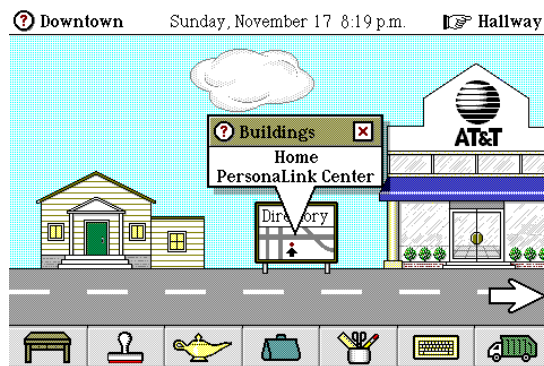
Figure 13-1: The Magic Cap interface from General Magic was used in products from Sony and Motorola in the mid-1990s. It is a tour de force of metaphoric design. All the navigation in the interface, and most other interactions as well, were subordinated to the maintenance of spatial and physical metaphors. It was probably fun to design but was not particularly convenient to use after you became an intermediate. This was a shame, because some of the lower-level, nonmetaphoric data-entry interactions were quite sophisticated, well designed, and ahead of their time.

General Magic's interface relied on what is called a *global metaphor*. This is a single, overarching metaphor that provides a framework for all the other metaphors in the system. It might work for a video game, but much less so for anything where efficiency is a concern.

The hidden problem of global metaphors is the mistaken belief that other lower-level metaphors consistent with them enjoy cognitive benefits by association. It's impossible to resist stretching the metaphor beyond simple function recognition: That software

telephone also lets us dial with buttons just like those on our desktop telephone. We see software that has an address book of phone numbers just like those in our pocket and purse. Wouldn't it be better to go beyond these confining, Industrial Age technologies and deliver some of the computer's real power? Why shouldn't our communications software allow multiple connections or make connections by organization or affiliation, or just hide the use of phone numbers?

Alexander Graham Bell would have been ecstatic if he could have created a phone that let you call your friends just by pointing to pictures of them. He couldn't do so because he was restricted by the dreary realities of electrical circuits and Bakelite moldings. On the other hand, today we have the luxury of rendering our communications interfaces in any way we please. Showing pictures of our friends is completely reasonable. In fact, it's what modern phone interfaces like the iPhone do.

For another example of the problematic nature of extending metaphors, we need look no further than the file system and its folder metaphor. As a mechanism for organizing documents, it is quite easy to learn and understand because of its similarity to a physical file folder in a file cabinet. Unfortunately, as is the case with most metaphoric user interfaces, it functions a bit differently than its real-world analog, which has the potential to create cognitive friction on the part of users. For example, in the world of paper, no one nests folders 10 layers deep. This fact makes it difficult for novice computer users to come to terms with the navigational structures of an operating system.

Implementing this mechanism also has limiting consequences. In the world of paper, it is impossible for a document to be located in two different places in a filing cabinet. As a result, filing is executed with a single organization scheme (such as alphabetically by name or numerically by account number). Our digital products are not intrinsically bound by such limitations. But blind adherence to an interface metaphor has drastically limited our ability to file a single document according to multiple organization schemes.

> **DESIGN PRINCIPLE** *Never bend your interface to fit a metaphor.*

As Brenda Laurel described in *Computers* as Theatre (Addison-Wesley, 2013), "Interface metaphors rumble along like Rube Goldberg machines, patched and wired together every time they break, they are so encrusted with the artifacts of repair that we can no longer interpret them or recognize their referents." Of all the misconceptions to emerge from Xerox PARC, the desirability of global metaphors is perhaps the most debilitating and unfortunate.

Other limitations of metaphors

Metaphors have many other limitations when applied to modern Information Age systems. For one thing, metaphors don't scale very well. A metaphor that works well for a simple process in a simple application often fails to work well as that process grows in size or complexity. Large desktop file icons as a means of accessing and manipulating files were a good idea when computers had floppy drives or 20 MB hard drives with only a couple of hundred files. But in these days of terabyte hard drives and tens of thousands of files, file icons become too clumsy to use effectively by themselves as a means for moving files around.

Next, while it may be easy to discover visual metaphors for physical objects like printers and documents, it can be difficult or impossible to find metaphors for processes, relationships, services, and transformations—the most frequent uses of software. It can be daunting to find a useful visual metaphor for changing channels, purchasing an item, finding a reference, setting a format, changing a photograph's resolution, or performing statistical analysis. Yet these operations are the types of processes we use software to perform most frequently.

Metaphors also rely on associations perceived in similar ways by both the designer and the user. If the user doesn't have the same cultural background as the designer, metaphors can fail. Even in the same or similar cultures, significant misunderstandings can occur. Does a picture of an airplane in an airline app mean "Check flight arrival information" or "Make a reservation"?

Finally, although a metaphor is easier for first-time users to understand, it exacts a tremendous cost after they become intermediates. By reflecting the physical world of mechanisms, most metaphors firmly nail our conceptual feet to the ground, forever limiting the power of our software. It is almost always better to design idiomatically, using metaphors only when a truly appropriate and powerful one falls in our lap.

Exceptions to the rule

Although metaphorical and skeuomorphic user interfaces should, generally speaking, be avoided, there are always exceptions to the rule. Video games often employ diegetic interfaces to keep players in the game world. Simulation software, such as flight simulators, intentionally use controls resembling their real-world counterparts. Another genre of software that makes heavy use of metaphoric interfaces is music creation software. While simulating piano keys, drum pads, synthesizer knobs and sliders, or even the frets and strings of a guitar may seem a bit silly in a mouse-driven desktop interface, it feels quite different on a multitouch iPad screen. There the expressiveness of a virtual instrument can begin to match the expressiveness of a real-world one, as shown in Figure 13-2.

Figure 13-2: Sunrizer is an iPad synthesizer that closely resembles its hardware brethren. On a touchscreen, simulated knobs and sliders make sense if your users are accustomed to hardware user interfaces, since the interaction with them is so similar to the real-world interactions. However, the creators of Sunrizer have not become slaves to the global metaphor, but have rather improved on the real world as only a digital interface can. Swiping left or right on the keyboard slides higher or lower octave keys into view, effectively removing the limitations of screen width.

On the other hand, digital musical instruments don't require the use of metaphor to be successful or expressive. TC-11 is an iPad synthesizer that uses an abstract, idiomatic interface that is both unique and extremely expressive, as shown in Figure 13-3.

As people increasingly make use of multi-touch displays in place of hardware-controlled gadgets, tools, and instruments, it's reasonable to expect that real-world metaphors will eventually fade and be supplanted by idiomatic interfaces more optimized to expressive gestures. We'll discuss what it means to create idiomatic interfaces in the next section.
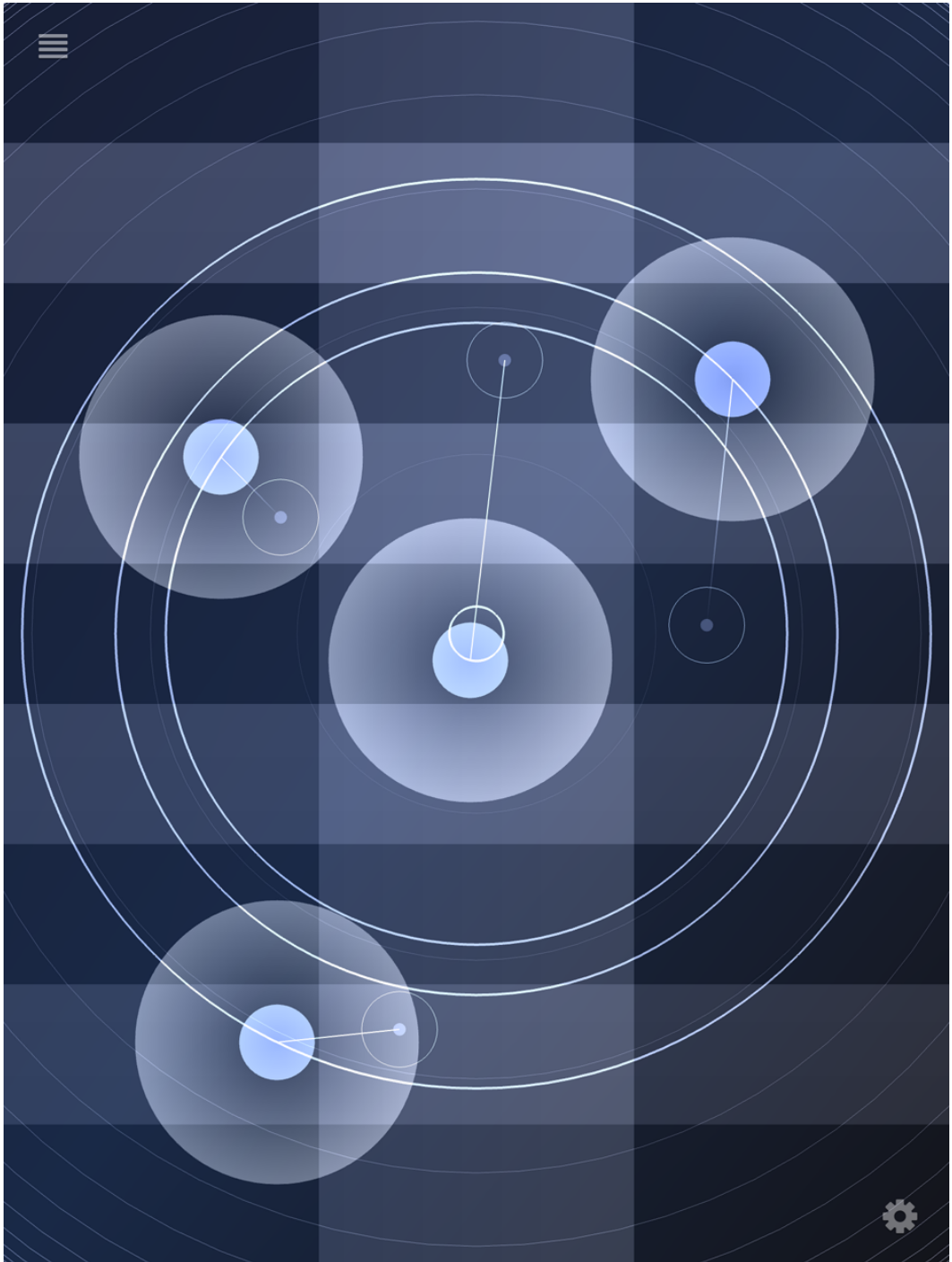
Figure 13-3: TC-11 takes a completely different approach to creating an expressive digital instrument. It sports a unique, abstract, and completely idiomatic user interface in which the user must learn by exploring the tonal and visual effects created by touching and gesturing. It even includes a sophisticated patch editor for building new sounds and interactions.

Idiomatic interfaces

Idiomatic design, what Ted Nelson has called "the design of principles," is based on how we learn and use idioms—figures of speech like "beat around the bush" and "cool." Idiomatic user interfaces solve the problems of the previous two interface types by focusing not on technical knowledge or intuition of function, but rather on the learning of simple, non-metaphorical visual and behavioral idioms to accomplish goals and tasks.

Idiomatic expressions don't provoke associative connections like metaphors do. There is no bush, and nobody is beating anything. Idiomatically speaking, something can be both cool and hot and be equally desirable. We understand the idiom simply because we have learned it and because it is distinctive, not because it makes subliminal connections in our minds. Yet we all can rapidly memorize and use such idioms: We do so almost without realizing it.

If you cannot intuit an idiom, neither can you reason it out. Our language is filled with idioms that, if you haven't been taught them, make no sense. If someone says "Uncle Joe kicked the bucket," you know what he means even though no bucket or kicking is involved. You can't figure it out by thinking through the various permutations of smacking pails with your feet. You can learn the meaning of this expression only from context in something you read or by being explicitly taught it. You remember this obscure connection between buckets, kicking, and dying only because humans are good at remembering things like this.

The human mind has a truly amazing capacity to learn and remember large numbers of idioms quickly and easily, without relying on comparisons to known situations or an understanding of how or why they work. This is a necessity, because most idioms don't have metaphoric meaning, and the stories behind most others were lost ages ago.

Graphical interfaces are largely idiomatic

It turns out that most of the elements of intuitive graphical interfaces are actually visual idioms. Windows, title bars, close boxes, screen splitters, hyperlinks, and drop-downs are things we learn idiomatically rather than intuit metaphorically. OS X's use of a trash can to unmount an external FireWire disk before removing it is purely idiomatic (and many designers consider it a poor idiom), despite the visual metaphor of the trash can.

The mouse input devices used by most personal computers are not metaphoric of anything, but rather are learned idiomatically. Nothing about the mouse's physical appearance indicates its purpose or use, nor is it comparable to anything else in our experience, so learning it is not intuitive. (Even the name "mouse" is rather unhelpful in that regard.)

In a scene from the movie *Star Trek IV: The Voyage Home*, Scotty (one of the best engineers from the 23rd century) comes to 20th-century Earth and tries to use a computer.

He picks up the mouse, holds it to his mouth, and speaks into it. This scene is funny and believable: The mouse has no visual affordance that it is a pointing device. However, as soon as you slide the mouse around on your desktop, you see a visual symbol, the cursor, move around on the computer screen in the same way. Move the mouse left, and the cursor moves left; move the mouse forward, and the cursor moves up. As you first use the mouse, you immediately get the sensation that the mouse and cursor are connected. This sensation is extremely easy to learn and equally hard to forget. That is idiomatic learning.

Modern multi-touch user interfaces, seen on most smartphones and tablets, also are idiomatic. (Although touching objects on the screen to activate them is intuitive, the gestural idioms must all be learned.) This is becoming even more true as the skeuomorphisms once popularized by Apple have increasingly been replaced with flatter and graphically simpler layouts and controls. Touch gestures, when designed correctly, can be learned even more easily than mouse movements. This is because you have more direct ability to manipulate objects on-screen with your fingers rather than via the virtual proxy of a mouse cursor.

Ironically, many of the familiar graphical UI elements that have been historically thought of as metaphoric are actually idiomatic. Artifacts like resizable windows and endlessly nested file folders are not really metaphoric, because they have no parallel in the real world. They derive their strength only from their easy idiomatic learnability.

Good idioms must be learned only once

We are inclined to think that learning interfaces is hard because of our conditioning based on experience with implementation-centric software. These interfaces are very hard to learn because you need to understand how the software works internally to use them effectively. Most of what we know we learn *without* understanding: things like faces, social interactions, attitudes, melodies, brand names, the arrangement of rooms and furniture in our house and office. We don't *understand* why someone's face is composed the way it is, but we *know* that face. We recognize it because we have looked at it and have automatically (and easily) memorized it.

> **DESIGN PRINCIPLE**   *All idioms must be learned; good idioms need to be learned only once.*

The key observation about idioms is that although they must be learned, they are very easy to learn, and good ones need to be learned only once. It is quite easy to learn idioms like "neat" or "politically correct" or "the lights are on but nobody's home" or "in a pickle" or "take the red-eye" or "grunge." The human mind can pick up idioms like these from a single hearing. It is similarly easy to learn idioms like radio buttons, close boxes, drop-down menus, and combo boxes.

Branding and idioms

Marketing and advertising professionals understand well the idea of taking a simple action or symbol and imbuing it with meaning. After all, synthesizing idioms is the essence of product branding, in which a company takes a product or company name and imbues it with a desired meaning. The example of an idiomatic symbol shown in Figure 13-4 illustrates its power.

Figure 13-4: This idiomatic symbol has been imbued with meaning from its use, rather than by any connection to other objects. For anyone who grew up in the 1950s and 1960s, this otherwise meaningless symbol has the power to evoke fear because it represents nuclear radiation. Visual idioms, such as the American flag, can be just as powerful as metaphors, if not more so. The power comes from how we use them and what we associate with them, rather than from any innate connection to real-world objects.

Building Idioms

When graphical user interfaces were invented, they were so clearly superior that many observers credited their success to the interfaces' graphical nature. This was a natural, but incorrect, assumption. The first graphical UIs, such as the original Mac OS, were better primarily because the graphical nature of their interfaces required a restriction of the range of vocabulary by which the user interacted with the system. In particular, the input they could accept from the user went from an unrestricted command line to

a tightly restricted set of mouse-based actions. In a command-line interface, users can enter any combination of characters in the language—a virtually infinite number. For a user's entry to be correct, he needs to know exactly what the application expects. He must remember the letters and symbols with exacting precision. The sequence can be important. Sometimes even capitalization matters.

In modern desktop UIs, users can point to images or words on the screen with the mouse cursor. Most of these choices migrated from the users' heads to the screen, eliminating any need to memorize them. Using the mouse buttons, users can click, double-click, or click and drag. The keyboard is used for data entry, but typically not for command entry or navigation. The number of atomic elements in users' input vocabulary has dropped from dozens to just three. This is true even though the range of tasks that can be performed by modern software apps isn't any more restricted than that of command-line systems.

The more atomic elements an interaction vocabulary has, the more time-consuming and difficult the learning process is. Restricting the number of elements in our interaction vocabulary reduces its expressiveness at the atomic level. However, more-complex interactions can easily be built from the atomic ones, much like letters can be combined to form words, and words to form sentences.

A properly formed interaction vocabulary can be represented by an inverted pyramid. All easy-to-learn communications systems obey the pattern shown in Figure 13-5. The bottom layer contains *primitives*, the atomic elements of which everything in the language is composed. In modern desktop graphical UIs, these primitives consist of positioning the mouse, clicking, and tapping a key on the keyboard. In touch-gesture systems they consist of tapping and dragging.

Input delete, create, draw — Idioms: Application specific commands and feedback — Output, scrolling, sorting, dialogs

Double-click, buttonclick, selection — Compounds: Generic input and output actions and symbols — Edit fields, checkboxes, Highlighting

Point, click, drag, keypress — Primitives: Indivisible actions and feedback mechanisms — Cursor, text
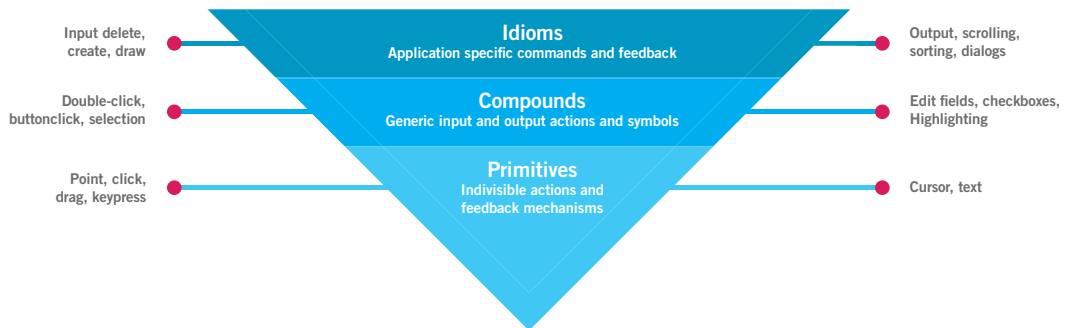
Figure 13-5: One of the primary reasons that graphical UIs are easy to use is that they enforce a restricted interaction vocabulary that builds complex idioms from a very small set of primitives: pointing, clicking, and dragging. These primitives can build a larger set of simple compounds. These in turn can be assembled into a wide variety of complex, domain-specific idioms, all of which are based on the same small set of easily learned actions.

The middle layer contains *compounds*. These are more complex constructs created by combining one or more of the primitives. They include simple visual objects such as text display; actions such as double-clicking, dragging, swiping, and pinching; and manipulable objects like buttons, check boxes in a form, links, and resize handles.

The uppermost layer contains *idioms*. Idioms combine and structure compounds using *domain knowledge* of the problem under consideration: information related to the user's work patterns and goals, not specifically to the computerized solution. The set of idioms opens the vocabulary to information about the particular problem the application is trying to address. In a graphical UI, it includes things like labeled buttons and fields, navigation bars, list boxes, icons, and even groups of fields and controls, or entire panes and dialogs.

Any language that does not follow this form will be very hard to learn. Many effective communications systems outside the computer world use similar vocabularies. Street signs in the U.S. follow a simple pattern of shapes and colors: Yellow triangles are cautionary, red octagons are imperatives, and green rectangles are informative.

Manual Affordances

In his seminal book *The Design of Everyday Things* (Basic Books, 2002), Donald Norman gives us the term *affordance*, which he defines as "the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used."

This concept is essential to the practice of interface design. But for our purposes, the definition omits a key connection: *How* do we know what those properties offer us? If you look at something and understand how to use it—you comprehend its affordances—you must be using some method to make the mental connection.

Therefore, we propose altering Norman's definition by omitting the phrase "and actual." When we do this, affordance becomes a purely cognitive concept, referring to what we *think* the object can do rather than what it actually can do. If a pushbutton is placed next to the front door of a residence, its affordances are 100 percent doorbell. If, when we push it, it causes a trapdoor to open and we fall into it, it turns out that it wasn't a doorbell, but that doesn't change its affordance as one.

So how do we know it's a doorbell? Because we have learned about doorbells, door etiquette, and pushbuttons from our complex and lengthy socialization process. We have learned about this class of pushable objects by being exposed to electrical and electronic devices in our environs and because—years ago—we stood on doorsteps with our parents, learning how to approach another person's home.

But another force is at work here too. If we see a pushbutton in an unlikely place such as the hood of a car, we cannot imagine what its purpose is, but we do recognize it as a pushable object. How do we know this? Undoubtedly, we recognize it because of our tool-manipulating instincts. When we see round, slightly concave, finger-sized objects within reach, we develop an urge to push them. (You can easily observe this behavior in any two-year-old.) We see objects that are long and rounded, and we wrap our fingers around them and grasp them like handles. This is what Norman is getting at with his term *affordance*. For clarity, however, we'll rename this instinctive understanding of how objects are manipulated with our hands *manual affordance*. When artifacts are clearly shaped to fit our hands or body, we recognize that they can be manipulated directly and require no written instructions. In fact, this act of understanding how to use a tool based on the relationship of its shape to our hands is a clear example of *intuiting* an interface.

Norman discusses at length how manual affordances are much more compelling than written instructions. A typical example he uses is a door that must be pushed open using a metal bar for a handle. The bar is just the right shape and height and is in the right position to be grasped by the human hand. The door's manual affordances scream, "Pull me!" No matter how often someone uses this diabolical door, he will always attempt to pull it open, because the affordances are strong enough to drown out any number of signs affixed to the door saying Push.

There are only a few manual affordances. We grip handle-shaped things with our hands; if they are small, we pinch or push them with our fingers. We pull along lines and turn around axes. We push flat plates with our hands or fingers. If they are on the floor, we push them with our feet. We rotate round things, using our fingers for small things—like dials—and both hands on larger things, like steering wheels. Such manual affordances are the basis for much of our visual user interface design.

The design of widgets for older operating system interfaces like Windows 7 and OS X relied on shading, highlighting, and shadows to make screen images appear more dimensional. These so-called skeuomorphic clues have fallen out of fashion with Android Kitkat, Windows 8, and OS Mavericks, but where they appear they offer *virtual* manual affordances in the form of button-like images that say "push me" or "slide me" to our tool-manipulating brains. Recent trends in flattened, visually minimal user interfaces threaten ease of use by removing these virtual manual affordances in the service of visual simplification.

Semantics of manual affordances

What's missing from an unadorned, virtual manual affordance is any idea of what function it performs. We can see that it looks like a button, but how do we know what it will accomplish when we press it? Unlike mechanical objects, you can't figure out a virtual lever's function just by tracing its connections to other mechanisms. Software can't be

casually inspected in this manner. Instead, we must rely on either supplementary text and images or, most often, our previous learning and experience. The Windows 7 scrollbar's affordance clearly shows that it can be manipulated. But the only things about it that tells us what it does are the arrows (frequently missing in mobile apps), which hint at its directionality. To know that a scrollbar controls our position in a document, we either have to be taught or learn through experimentation.

Controls must have text or iconic labels on them to make sense. If the answer isn't suggested by the control, we can only learn what it does by one of two methods: experimentation or training. We either read about it somewhere, ask someone, or try it and see what happens. We get no help from our instinct or intuition. We can only rely on the empirical.

Fulfilling expectations of affordances

In the real world, an object does what it does as a result of its physical form and its connections with other physical objects. A saw can cut wood because its serrations are sharp, its blade is flat, and it has a handle. A knob can open a door because it is connected to a latch. However, in the digital world, an object does what it does because a developer imbued it with the power to do something. We can discover a great deal about how a saw or a knob works by physical inspection, and we can't easily be fooled by what we see. On a computer screen, though, we can see a raised three-dimensional rectangle that clearly wants to be pushed like a button, but this doesn't necessarily mean that it *should* be pushed. It could do almost anything. We can be fooled because there is no natural connection—as there is in the real world—between what we see on the screen and what lies behind it. In other words, we may not know how to work a saw, and we may even be frustrated by our inability to manipulate it effectively, but we will never be fooled by it. It makes no representations that it doesn't live up to. On computer screens, false impressions are very easy to create inadvertently.

When we render a button on the screen, we are making a contract with the user that that button will change visually when she pushes it: It will appear to be actuated when tapped or when the user clicks while the mouse cursor hovers over it. Furthermore, the contract states that the button will perform some reasonable work that is accurately described by its legend. This may sound obvious, but it is astonishing how many applications offer bait-and-switch manual affordances. This is relatively rare for pushbuttons but is all too common for other controls, especially on websites where the lack of affordances can make it difficult to differentiate between controls, content, and ornamentation. Make sure that your application delivers on the expectations it sets via the use of manual affordances.

Direct Manipulation and Pliancy

Modern graphical user interfaces are founded on the concept of *direct manipulation* of graphical objects on the screen: buttons, sliders, menus, and other function controls, as well as icons and other representations of data objects. The ability to select and modify objects on the screen is fundamental to the interfaces we design today. But what is direct manipulation, exactly?

In 1974, Ben Shneiderman coined the term "direct manipulation" to describe an interface design strategy consisting of three important components:

- Visual representation of the data objects that an application is concerned with
- Visible and gestural mechanisms for acting on these objects (as opposed to free-form text commands)
- Immediately visible results of these actions

It's worth noting that two of his three points concern the visual feedback the application offers to users. It might be more accurate to call it "visual manipulation" because of the importance of what users see during the process. Virtual manual affordances and rich visual feedback are both key elements in the design of direct manipulation interfaces.

> **DESIGN PRINCIPLE**
>
> *Rich visual feedback is the key to successful direct manipulation.*

Interactions that are not implemented with adequate visual feedback will fail to effectively create the experience of direct manipulation.

Uses of direct manipulation

Using direct manipulation, we can point to what we want. If we want to move an object from A to B, we click or tap it and drag it there. As a general rule, the better, more flow-inducing interfaces are those with plentiful and sophisticated direct manipulation idioms.

Authoring tools do this pretty well. For example, most word processors let you set tabs and indentations by dragging a marker on a ruler. Someone can say, in effect, "Here is where I want the paragraph to start." The application then calculates that this is precisely 1.347 inches from the left margin instead of forcing the user to enter 1.347 in some text box somewhere.

Similarly, most art and design tools (such as Adobe's Creative Suite) provide a high degree of direct manipulation of objects (although many parameters of the click-and-type variety remain). Google's Snapseed photo editor is a great example of a consumer-focused multi-touch app that employs direct manipulation to good effect. Gestures control image processing parameters for digital photo editing instead of cumbersome sliders or numeric text fields.

Figure 13-6 shows how multiple control points can be placed or selected by tapping. These points can be moved by dragging; and pinching on the currently selected point adjusts the diameter of the applied filter, with feedback to the user in the form of a circle and red tint to show the extent of the filter's application. Swiping horizontally controls the intensity of the filter, tracked both by the green meter surrounding the point, and the numeric scale at the bottom of the screen. Vertical swiping selects between brightness, contrast, and saturation. While this is a lot of functionality to build into gestures, it becomes second nature after a few uses due to the rich visual modeless feedback and fluidity with which images can be adjusted.
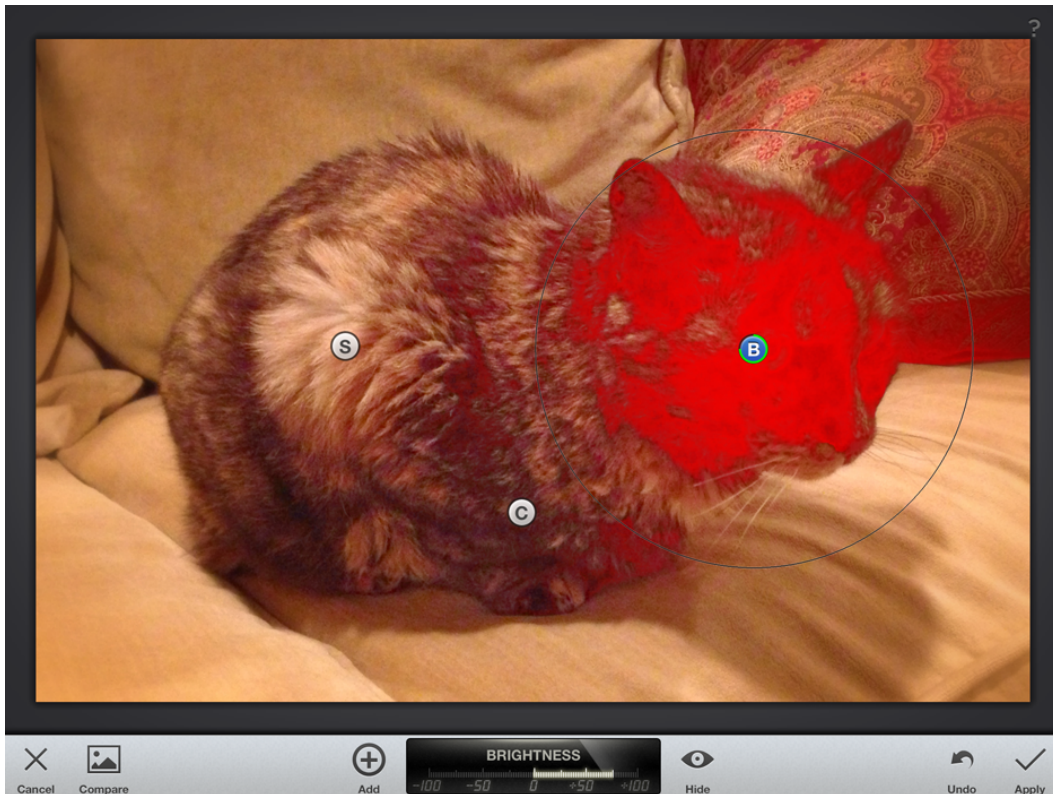
Figure 13-6: Google's Snapseed photo editor for the iPad uses gestural controls to position and manipulate visual effects parameters via tapping, pinching, twirling, and swiping. Numeric feedback is provided in addition to real-time previewing, but no textual numeric entry is required or, in fact, even allowed—not that it is missed.

The principle of direct manipulation applies in a variety of situations. When items in a list need to be reordered, the user may want them ordered alphabetically, but he also may want them in order of personal preference—something no algorithm can offer. A user should be able to drag the items into the desired order directly, without an algorithm's interfering with this fundamental operation.

Drag and drop can also save the user from tiresome and repetitive use of dialogs and menus. In the Sonos Desktop Controller, shown in Figure 13-7, users can drag songs directly to any location in the current play queue or instead drag them to any speaker in the house for immediate playback. They don't have to open a menu and choose options.

You seldom see direct manipulation interfaces for entering complex numeric data. Usually you're given numeric entry fields or sliders. A great example of direct manipulation of graphically presented numeric data is the Addictive Synth app for the iPad, shown in Figure 13-8. It allows you to sketch waveforms and effect parameters for a music synthesizer with your finger and then play the results immediately on the onscreen piano keyboard.
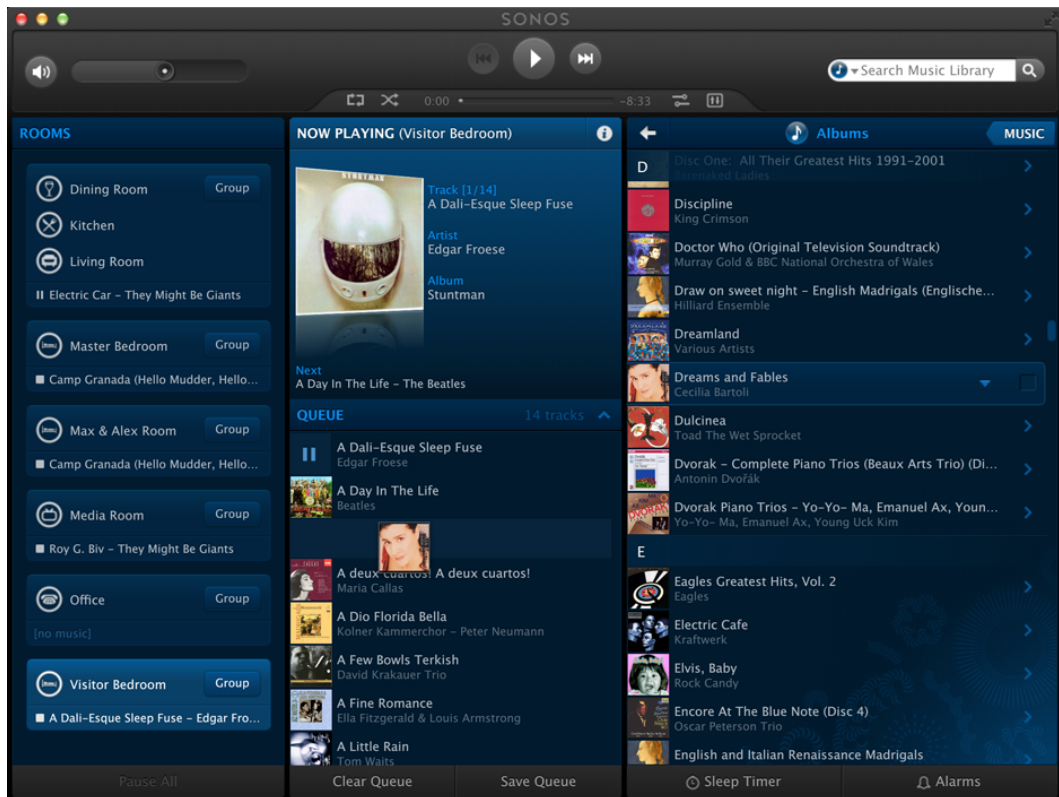
Figure 13-7: The Sonos Desktop Controller lets you drag songs and albums from search-and-browse results to anywhere in the playback queue, to the Now Playing area for immediate playback, or to any room in the house. You do so with a single drag-and-drop gesture. Tablet versions of the app also allow similar direct manipulation of music.

Figure 13-8: Addictive Synth is an iPad music synthesizer that allows users to draw their own waveforms and audio effects curves with their finger and then hear the results in real time. There's a reason for the app's name: The experience is immersive and satisfying.

Direct manipulation is simple, straightforward, easy to use, and easy to remember. However, as we have discussed, direct-manipulation idioms—like most other idioms—must first be learned. Luckily, because the visible and direct nature of these interactions bears a close resemblance to interactions with objects in the physical world, learning the idioms usually is easy. After you learn them, you seldom forget them.

Direct manipulation isn't always appropriate

Apple's *Human Interface Style Guide* has this to say about direct manipulation: "Users want to feel that they are in charge of the computer's activities." iOS user interfaces make it clear that Apple believes in direct manipulation as a fundamental tenet of interaction design. On the other hand, user-centered design expert Don Norman (2002) says, "Direct manipulation, first-person systems have their drawbacks. Although they are often easy to use, fun, and entertaining, it is often difficult to do a really good job with them. They require the user to do the task directly, and the user may not be very good at it." Whom should we believe?

The answer, of course, is both. Direct manipulation is a powerful tool, but it can require skill development for users to become effective at complex tasks (such as designing an airplane using a CAD system). Many direct-manipulation idioms, even relatively mundane ones, require motor coordination and a sense of purpose. For example, even moving files between folders in Windows Explorer can be a complicated task requiring dexterity and foresight. Keep these challenges in mind as you design direct-manipulation idioms. Some amount of direct manipulation is usually a good thing, but depending on your personas' skills and usage contexts, it's also possible to go overboard. You should always consider what your personas need to manipulate manually, and what the application can assist them with, either via guides and hints or automatically.

Pliancy and hinting

Returning to Norman's concept of affordance, it's critical to visually communicate to users *how* interface elements can be directly manipulated. We use the term *pliant* to refer to objects or screen areas that react to input and that the user can manipulate. For example, a button control is pliant because it can be "pushed" by a finger or a mouse cursor. Any object that can be dragged is pliant, and every cell in a spreadsheet and every character in a word processor document is pliant.

In most cases, the fact that an object is pliant should be communicated visually to users. The only situation where this isn't true is when you are concerned with presenting rich, complex functionality solely to expert users with no concern about their ability to learn and use the application. In these cases, the screen real estate and visual attention that would otherwise be devoted to communicating pliancy may be more appropriately used elsewhere. Do not make the decision to take this route lightly.

> **DESIGN PRINCIPLE**   *Visually communicate pliancy whenever possible.*

There are three basic ways to communicate—or hint at—the pliancy of an object to users:

- Create static visual affordances as part of the object itself.
- Dynamically change the object's visual affordances in reaction to change in input focus or other system events.
- In the case of desktop pointer-driven interfaces, change the cursor's visual affordance as it passes over and interacts with the object.

Static hinting

Static hinting is when an object's pliancy is communicated by the static rendering of the object itself. For example, the faux three-dimensional sculpting of a button control is static visual hinting because it provides (virtual) manual affordance for pushing.

For complex desktop interfaces with a lot of objects and controls, static object hinting can sometimes require an impractical number of rendered screen elements. If everything has a three-dimensional feel to provide affordance, your interface can start to look like a sculpture garden. Dynamic hinting, as we'll discuss in a minute, provides a solution to this problem.

However, static hinting is well-suited for mobile user interfaces. Typically fewer objects are on the screen at any given time, and they must, by necessity, be large enough to manipulate with fingers, leaving ample room for the necessary visual cues of affordance.

Ironically, the current trend in mobile UIs is toward flattening and visually simplifying elements to the point where text, flat monochrome icons, and rectilinear flat buttons and cards are the only visual elements available. This creates many challenges for designers, both from a standpoint of creating visual hierarchy and for indicating pliancy and affordance. The end result is that mobile interfaces are becoming more difficult to learn, even as they are becoming visually simpler.

Dynamic hinting

Dynamic hinting is most often used in desktop user interfaces. It works like this: When the cursor passes over a pliant object, the object temporarily changes its appearance, as shown in Figure 13-9. This action occurs before any mouse buttons are clicked and is triggered by cursor flyover only. It is commonly called a "rollover." A good example of this is the behavior of icon buttons (see Chapter 21) on toolbars: Although it has no persistent button-like affordance, passing the cursor over any single icon button causes the affordance to appear. The result is a powerful hint that the control has the behavior of a button, and eliminating the persistent affordance dramatically reduces visual clutter on the toolbar.
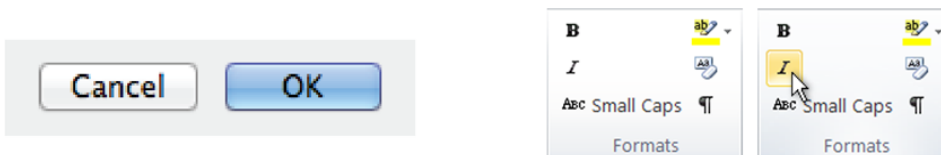
Figure 13-9: The buttons on the left are an example of static visual hinting: Their "clickability" is suggested by the dimensional rendering. The toolbar icon buttons on the right demonstrate dynamic visual hinting: While the Bold toggle doesn't appear to be a button at first glance, passing the mouse cursor over it causes it to change, thereby creating affordance.

Alas, touchscreen devices have no real equivalent to dynamic object hinting; designers and users must make do with whatever static hinting at pliancy is available to them.

Pliant response hinting

Desktop pliant response hinting should occur if the mouse cursor is clicked but not released or while a finger is pressed on a control. The control must show that it is poised to undergo a state change (more on this in Chapter 18). This action is important and is often neglected by developers who create their own controls.

A pushbutton needs to change from a visually raised state to a visually indented state; a check box should highlight its box but not show a check just yet. Pliant response is an important feedback mechanism for any control that either invokes an action or changes its state. It lets the user know that some action is forthcoming if she releases the mouse button. Pliant response is also an important part of the cancel mechanism. When the user clicks a button, that button responds by becoming indented. If the user moves the mouse or his finger away from the button while still holding it down, the onscreen button returns to its quiescent, raised state. If the user then releases the mouse button or lifts his finger, the onscreen button is not activated (consistent with the lack of pliant response).

Cursor hinting

Cursor hinting is another desktop pliancy hinting approach. It communicates pliancy by changing the cursor's appearance as it passes over an object or screen area. For example, when the cursor passes over a window's frame, the cursor changes to a double-headed arrow showing the axis in which the window edge can be stretched. This is the only visual affordance indicating that the frame can be stretched.

Cursor hinting should first and foremost make it clear to users that an otherwise unadorned object is pliant. It is also often useful to indicate what type of direct-manipulation action is possible with an object (such as in the window frame example just mentioned).

Generally speaking, controls should offer static or dynamic visual hinting, whereas pliant (manipulable) data more frequently should offer cursor hinting. For example, it is difficult to make dense tabular data visually hint at pliancy without disturbing its clear representation, so cursor hinting is the most effective method. Some controls are small and difficult for users to spot as readily as a button, and cursor hinting is vital for the success of such controls. The column dividers and screen splitters in Microsoft Excel are good examples, as shown in Figure 13-10.
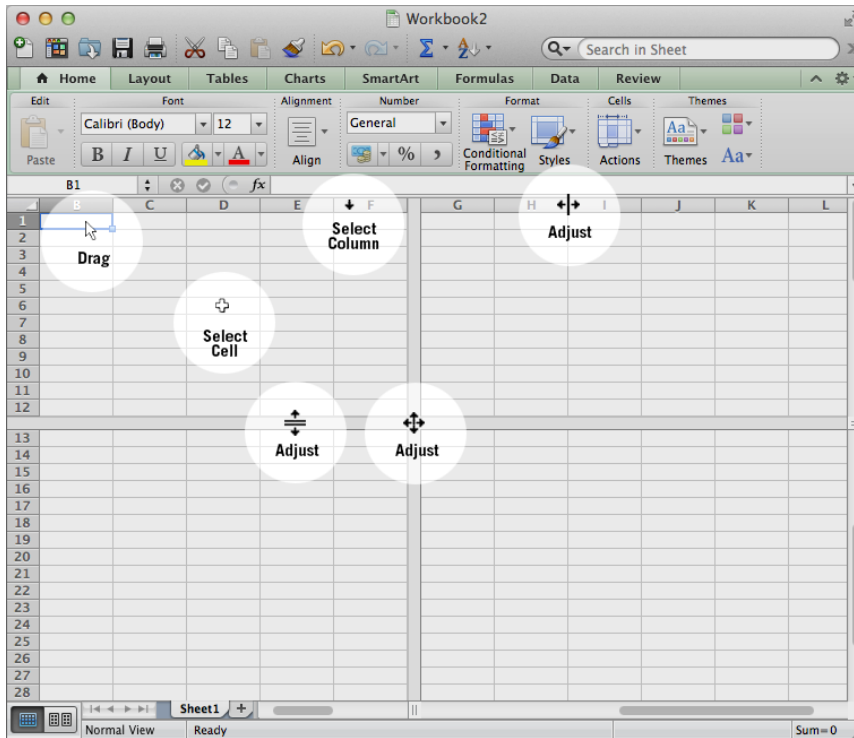
Figure 13-10: Excel uses cursor hinting to highlight several controls that are not obviously pliant by themselves. You can set column width and row height by dragging the short vertical lines between each pair of columns or rows. The cursor changes to a two-headed horizontal arrow that both hints at the pliancy and indicates the permissible drag direction. The same is true for the screen-splitter controls. When the mouse is over an unselected editable cell, it shows the plus cursor, and when it is over a selected cell, it shows the drag cursor.

Again, touchscreen users and designers are out of luck with using this kind of pliancy hinting. As we'll discuss in more detail in Chapter 19, other strategies must be employed in the design of touchscreen apps to ensure that users know what objects they can manipulate, when they can do so, and what actions and gestures are supported.

Escape the Grip of Metaphor

As you build your app, you may be tempted to look backwards to comfortable visual metaphors in which to immerse your users. Avoid that temptation. Reserve global visual metaphors for those few, special contexts where the metaphor is truly an integral part of the experience. Don't use metaphors as a crutch to support rapid learnability or short-term comfort.

Instead, create memorable and appropriate idioms that make your users more effective and efficient, that are imbued with rich pliant feedback, and that allow users to focus on the content and functionality of your app rather than the confines of outmoded Mechanical Age metaphors and interactions. You'll be doing them a favor as they become the intermediate users they—and you—ultimately hope they will be.

RETHINKING DATA ENTRY, STORAGE, AND RETRIEVAL

In the world of digital technology, the place where implementation-model thinking is most strikingly apparent is in data management: entering data, storing it, and finding it again.

How often have you typed information into a set of forms, only to be told by an unpleasant or confusing error dialog that you entered it wrong? Maybe you typed hyphens in a telephone number, or perhaps you entered both your first and last name into the field that was intended to only hold your first name. Or maybe you accidentally typed text into a field meant for only numeric input. Quick, call the data police!

The problems don't end with data entry. If you have ever tried to teach your mother how to use a computer, you will know that the word "difficult" doesn't really do the problem justice. Things go well at first: You start the word processor and type a couple of sentences. She's with you all the way—it's like typing on paper. But when you click the Close button, up pops a dialog asking "Do you want to save changes?" You and Mom hit a wall together. She looks at you and asks, "What does this mean? Is everything okay?" And once you've talked her down and she performs the save, she has one more pressing and difficult question: "Where did it go? And how will I ever be able to find it again?"

Or it might even be on your own smartphone. You want to show an awesome photo to a friend, but given the long tapestry of thumbnails, have to spend more time hunting than the moment was really worth.

These problems are caused by software that forces people to think like computers by unnecessarily confronting them with the internal mechanisms of data entry, storage, and retrieval. This isn't just a problem for your mother; even sophisticated users can easily become confused or make mistakes. We spend thousands of dollars on hardware and software, only to be asked impertinent questions from desktop applications like "Do you really want to save this document?" Well, I've been working on it all afternoon, so yes, oddly enough, I do. Yes, Microsoft Office, we're looking at you.

This chapter provides a different set of approaches for tackling the problems of data entry, files and saving, and retrieval—ones that are more in harmony with the mental models of the people who use your products. Thankfully, we are not the only ones thinking in this way.

Rethinking Data Entry

In Chapter 8, we discussed how interactive products should be designed to behave like considerate and intelligent people. One of the ways in which products are least capable in this regard is when the user is required to enter data. Some unfortunate artifacts of implementation-model thinking prevent people from working in the way they find most natural. In this chapter, we'll discuss problems with existing ways of dealing with data entry and some possible strategies for making this process more focused on human needs and less focused on the needs of the database.

Data integrity versus data immunity

One of the most critical requirements for properly functioning software is clean data. As the aphorism says, "garbage in, garbage out." As a result, developers typically operate according to a simple imperative regarding data entry and data processing: Never allow tainted, unclean data to touch an application. Developers thus erect barriers in user interfaces so that bad data can never enter the system and compromise what is commonly called *data integrity*.

The imperative of data integrity posits that a world of chaotic information is out there, and before any of it gets inside the computer, it must be filtered and cleaned up. The software must maintain a vigilant watch for bad data, like a customs official at a border crossing. All data is validated at its point of entry. Anything on the outside is assumed to be suspect, and after it has run the gauntlet and been allowed inside, it is assumed to be pristine. The advantage is that once data is inside the database, the code doesn't have to bother with successive, repetitive checks of its validity or appropriateness.

The problem with this approach is that it places the needs of the database before those of its users, subjecting them to the equivalent of a shakedown every time they enter a

scrap of data into the system. You don't come across this problem often with most mobile or personal productivity software: PowerPoint doesn't know or care if you've formatted your presentation correctly. But as soon as you deal with a large corporation—whether you are a clerk performing data entry for an enterprise management system or a web surfer buying DVDs online—you come face to face with the border patrol.

People who fill out lots of forms every day as part of their job know that data typically isn't provided to them in the pristine form that their software demands. It is often incomplete and sometimes wrong. Furthermore, they may break from a form's strict demands to expedite this data processing to make their customers happy. But when confronted with a system that is entirely inflexible in such matters, these people must either grind to a halt or find some way to subvert the system to get things done. If the software recognized these facts of human existence and addressed them directly with an appropriate user interface, everyone would benefit.

Efficiency aside, this problem has a more insidious aspect: When software shakes down data at the point of entry, it makes a clear statement that the user is insignificant and the application is omnipotent—that the user works for the good of the application, not vice versa. Clearly, this is not the kind of world we want to create with our technological inventions. We want people to feel empowered and to make it clear that computers work for us. We must return to the ideal division of digital labor: The computer does the work, and the human makes the decisions.

Happily, there's more than one way to protect software from bad data. Instead of keeping it out of the system, the developer needs to make the system *immune* to inconsistencies and gaps in the information. This method involves creating much smarter, more sophisticated applications that can handle all permutations of data, giving the application a kind of *data immunity*.

To implement this concept of data immunity, our applications must be built to look before they leap and to ask for help when they need it. Most software blindly performs arithmetic on numbers without actually examining them first. The application assumes that a number field must contain a number, because data integrity tells it so. If the user enters the word "nine" instead of the number "9," the application barfs. But a human reading the form wouldn't even blink. If the application simply looked at the data before it acted, it would see that a simple math function wouldn't do the trick.

We must design our applications to believe that the user will enter what he means to enter, and if he wants to correct things, he will do so without paranoid insistence. But applications can look elsewhere in the computer for assistance. Is there a module that knows how to make numeric sense of alphabetic text? Is there a history of corrections that might shed some light on a user's intent?

If all else fails, an application must add annotations to the data so that when—and if—the user examines the problem, he finds accurate and complete notes that describe what happened and what steps the application took.

Yes, if the user enters "asdf" instead of "9.38," the application will be unable to achieve satisfactory results. But stopping the application to resolve this *right now* also is unsatisfactory; the entry process is just as important as the end report. If a user interface is designed correctly, the application provides visual feedback when the user enters "asdf," so it's very unlikely that the user will enter hundreds of bad records. Generally, users act stupidly only when applications treat them stupidly.

When the user enters incorrect data, it is often *close* to being correct; applications should be designed to provide as much assistance as possible in correcting the situation. For example, if the user erroneously enters "TZ" for a two-letter state code, and also enters "Dallas" for a city name, it doesn't take a lot of intelligence or computational resources to figure out how to correct the problem.

Handling missing data

It is clearly counter to users' goals—and to the system's utility—if crucial data is omitted. The data-entry clerk who fails to key in something as important as an invoice amount creates a real problem. However, it isn't necessarily appropriate for the application to stop the clerk and point out this failure. Think of your application as being like a car. Your users won't take kindly to having the steering wheel lock up because the car discovered it was low on windshield-washer fluid.

Instead, applications should provide more flexibility. Users may not immediately have access to data for all the required fields, and their workflow may be such that they first enter all the information they have on hand and then return when they have the information needed to fill in the other fields. Of course, we still want our users to be *aware* of any required fields that are missing information, but we can communicate this to them through *rich modeless feedback*, rather than stopping everything to let them know something they may be well aware of.

Take the example of a purchasing clerk keying invoices into a system. Our clerk does this for a living and has spent thousands of hours using the application. He has a sixth sense for what is happening on the screen and wants to know if he has entered bad data. He will be most effective if the application notifies him of data-entry errors by using subtle visual and audible cues.

The application should also help him: Data items, such as part numbers, that *must* be valid shouldn't be entered into free text fields, but instead should be entered via type-ahead (auto-completion) fields or bounded controls such as drop-downs. Addresses and

phone numbers should be entered more naturally into smart text fields that can parse the data. The application should provide unobtrusive modeless feedback on the status of the clerk's work. This will enable him to take control of the situation and will ultimately require less policing by the application.

Most of our information-processing systems *do* tolerate missing information. A missing name, code, number, or price can almost always be reconstructed from other data in the record. If not, the data can always be reconstructed by asking the various parties involved in the transaction. The cost is high, but not as high as the cost of lost productivity or technical support centers. Our information-processing systems can work just fine with missing data. Some developers who build these systems may not like all the extra work involved in dealing with missing data, so they invoke data integrity as an unbreakable law. As a result, thousands of clerks must interact with rigid, overbearing software under the false rubric of keeping databases from crashing.

It is obviously counterproductive to treat workers like idiots to protect against those few who are. It lowers everyone's productivity; encourages rapid, expensive, and error-causing turnover; and decreases morale, which increases the unintentional error rate of the clerks who want to do well. It is a self-fulfilling prophecy to assume that your information workers are untrustworthy.

The stereotypical role of the data-entry clerk mindlessly keypunching from stacks of paper forms while sitting in a boiler room among hundreds of identical clerks doing identical jobs is rapidly evaporating. The task of data entry is becoming less a mass-production job and more a productivity job—one performed by intelligent, capable professionals and, with the popularization of e-commerce, directly by customers. In other words, the population interacting with data-entry software is increasingly less tolerant of being treated like unambitious, uneducated, unintelligent peons. Users won't tolerate stupid software that insults them, not when they can push a button and surf for another few seconds until they find another vendor that presents an interface that treats them with respect.

Data entry and fudgeability

If a system is too rigid, it can't model real-world behaviors. A system that rejects the reality of its users is not helpful, even if the net result is that all its fields are valid. Which is more important, the database or the business it is trying to support? The people who manage the database and create the data-entry applications that feed it often serve only the CPU. This is a significant conflict of interest that good interaction design can help resolve.

Fudgeability can be difficult to build into a computer system because it demands a considerably more capable interface. Our clerk cannot move a document to the top of the queue unless the queue, the document, and its position in the queue can be easily seen. The tools for pulling a document out of the electronic stack and placing it on the

top must also be present and obvious in their functions. Fudgeability also requires facilities to hold records in suspense, but an Undo facility has similar requirements. A more significant problem is that fudging allows possible abuse.

The best strategy to avoid abuse is using the computer's ability to record the user's actions for later examination, if warranted. The principle is simple: Let users do what they want, but keep detailed records of those actions so that full accountability and recovery is possible.

Auditing versus editing

Many developers believe it is their duty to inform users when they make errors entering data. It is certainly an application's duty to inform *other applications* when they make an error, but this rule shouldn't necessarily extend to users. The customer is always right, so an application must accept what the user tells it, regardless of what it does or doesn't know. This is similar to the concept of data immunity, because whatever the user enters should be acceptable, regardless of how incorrect the application believes it to be.

This doesn't mean that the application can throw up its hands and say, "All right, he doesn't want a life preserver, so I'll just let him drown." Just because the application must act as though the user is always right doesn't mean that a user actually *is* always right. Humans often make mistakes, and your users are no exception. User errors may not be your application's fault, but they are its responsibility. How will it fix them?

> **DESIGN PRINCIPLE**
>
> *An error may not be your application's fault, but it is its responsibility.*

Applications can provide warnings—as long as they don't stop the proceedings with idiocy. But if the user chooses to do something suspect, the application can do nothing but accept that fact and work to protect the user from harm. Like a faithful guide, it must follow its client into the jungle, making sure to bring along a rope and plenty of water.

Warnings should clearly and modelessly tell users what they have done, much as the speedometer silently reports our speed violations. It is unreasonable, however, for the application to stop the proceedings, just as it is not right for the speedometer to cut the gas when we edge above 65 miles per hour. Instead of an error dialog, for example, data-entry fields can highlight any user input the application evaluates as suspect.

When the user does something that the application thinks is wrong, the best way to protect him (unless disaster is imminent) is to make it clear that there may be a problem. This should be done in an unobtrusive way that ultimately relies on the user's intelligence to figure out the best solution. If the application jumps in and tries to fix it, it may

be wrong and end up subverting the user's intent. Furthermore, this approach fails to give the user the benefit of learning from the situation, ultimately compromising his ability to avoid the situation in the future. Our applications should, however, remember each of the user's actions and ensure that each action can be cleanly reversed, that no collateral information is lost, and that the user can figure out where the application thinks the problems might be. Essentially, we maintain a clear audit trail of his actions. Thus the principle "Audit, don't edit."

DESIGN PRINCIPLE *Audit, don't edit.*

Microsoft Word has an excellent example of auditing, as well as a nasty counterexample. This excellent example is how it handles real-time spell checking. As you type, red wavy underlines identify words that the application doesn't recognize, as shown in Figure 14-1. Right-clicking these words pops up a menu of alternatives you can choose from. But you don't have to change anything, and you are not interrupted by dialogs or other forms of modal idiocy.
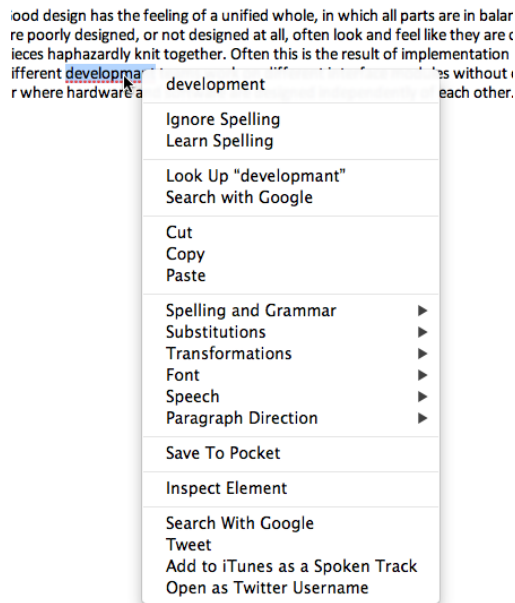
Figure 14-1: Microsoft Word's automatic spell checker audits misspelled words with a wavy red underline, giving users modeless feedback. Right-clicking an underlined word pops up a menu of possible alternatives to choose from. This design idiom has been widely copied by both desktop and mobile apps.

Word's AutoCorrect feature, on the other hand, can be a bit disturbing at first. As you type, it silently changes words it thinks are misspelled. It turns out that this feature is incredibly useful for fixing minor typos as you go. However, the corrections leave no obvious audit trail, so the user often doesn't realize that what he typed has been changed. It would be better if Word could provide some kind of mark that indicates it has made a correction on the off chance that it has miscorrected something. (This possibility becomes much more likely if, for instance, you are writing a technical paper heavy in specialized terminology and acronyms.)

More irksome is Word's AutoFormat feature, which tries to interpret user behaviors like use of asterisks and numbers in text to automatically format numbered lists and other paragraph formats. When this works, it seems magical. But frequently the application does the wrong thing, and once it does so, there is not always a straightforward way to undo the action. AutoFormat tries to be just a bit too smart; it should really leave the thinking to humans. Luckily, this feature can be turned off, and Word provides a special in-place menu that allows users to adjust AutoFormat assumptions.

In the real world, humans accept partially and incorrectly filled-in documents from each other all the time. We make a note to fix the problems later, and we usually do. If we forget, we fix the omission when we eventually discover it. Even if we never fix it, we somehow survive. It's certainly reasonable to use software to improve the efficiency of our data collection efforts, and in many cases it is consistent with human goals to do so. (No one wants to enter the wrong shipping address for an expensive online purchase.) However, our applications can be designed to better accommodate how humans think about such things. The technical goal of data integrity should not be our users' problem to solve.

Rethinking Data Storage

In our experience, people find computer file systems—the facilities that store application and data files on disk—difficult to use and understand. This is one of the most critical components of computers, and errors here have significant consequences. The difference between main memory and longer-term storage is unclear to most people. Unfortunately, how we've historically designed software forces users—even your mom— to know the difference and to think about their documents in terms of how a computer is constructed.

The popularization of web applications and other database-driven software has been a great opportunity to abandon this baggage of computer file system implementation-model thinking. As mentioned before, Google has led the charge with cloud-based web apps that auto-save, sparing users the worry and confusion.

Mobile operating systems like iOS try to address the problem of storage by tightly associating documents with the application that created them. You need to open the application to access the set of documents you created using it. Documents are saved automatically and also are retrieved from within the application. This makes things a lot simpler, once you get used to the app-centric paradigm—until you need to work on your document with a different application. iOS breaks this tight association rule with only a few document types—photos, for example—and then you are back to hunting for the one you need in a set of containers.

The problems with data storage

The roots of the interaction problems with data storage lie, as you'd expect, in implementation models. Technically speaking, every running app really exists in two places at once: in memory and on disk (or flash storage on mobile devices). The same is true of every open file. For the time being, this is a necessary state of affairs. Our technology has different mechanisms for accessing data in a responsive way (dynamic RAM memory) and storing that data for future use (disks/flash memory). However, this is not what most people think is going on. Most of our mental models (aside from developers) are of a single document that we are directly creating and making changes to. Unfortunately, most software presents us with a confusing representation of the implementation model of data storage.

Saving changes

When a Save Changes dialog like the one shown in Figure 14-2 opens, users suppress a twinge of fear and confusion and click the Save button out of habit. A dialog that is always answered the same way is redundant and should be eliminated.
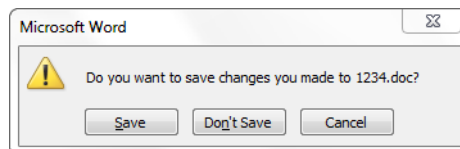
Figure 14-2: This is the question Word asks when you close a file after you have edited it. This dialog is a result of the developer's inflicting the implementation model of the disk file system on the hapless user. This dialog is so unexpected by new users that they often choose Don't Save inadvertently.

The application launches the Save Changes dialog when the user requests Close or Quit because that is when it has to reconcile the differences between the copy of the

document in memory and the copy on the disk. But in most cases, querying the user is simply unnecessary: A yes can be assumed.

The Save Changes dialog is based on a poor assumption: that saving and not saving are equally probable behaviors. The dialog gives equal weight to these two options even though the Save button is clicked orders of magnitude more frequently than the Don't Save button. As we discussed in Chapter 11, this is a case of confusing possibility and probability. The user *might* occasionally say Don't Save, but the user almost always *will* say Save. Mom is thinking, "If I didn't want those changes, why would I have closed the document with them in there?" To her, the question is absurd.

In reality, many applications need not concern themselves with document or file management. Apple's iPhoto and iTunes both provide rich and easy-to-use functionality that allows a typical user to ignore the fact that a file even exists. In iTunes, a playlist can be created, modified, shared, put onto an iPod, and persist for years, despite the fact that the user has never explicitly saved it. Similarly, in iPhoto, image files are sucked out of a camera into the application and can be organized, shown, e-mailed, and printed, all without users ever thinking about the file system. And mobile devices running iOS and Android have largely eliminated the concept of explicit saving.

Closing documents without saving

If you've been using computers for a long time, you've been conditioned to think that the document Close function is the appropriate way to abandon unwanted changes if you make an error or are simply noodling around. This is incorrect; the proper time to reject changes is when the changes are made. We even have a well-established idiom to support this: the Undo function. What's missing is a good way to perform a session-level undo (such as the Revert function, which only a few applications, like Adobe Photoshop, support) without resorting to closing the document without saving.

Experienced users also learn to use the Save Changes dialog for similar purposes. Since there is no easy way to undo massive changes in most documents, we (mis)use the Save Changes dialog by choosing Don't Save. If you discover yourself making big changes to the wrong file, you use this dialog as a kind of escape valve to return things to the status quo. This is handy, but it's also a hack: As we just mentioned, you have more discoverable ways to address these problems.

Save As

When you save a document for the first time or choose the Save As command from the File menu, many applications display the Save As dialog, shown in Figure 14-3.
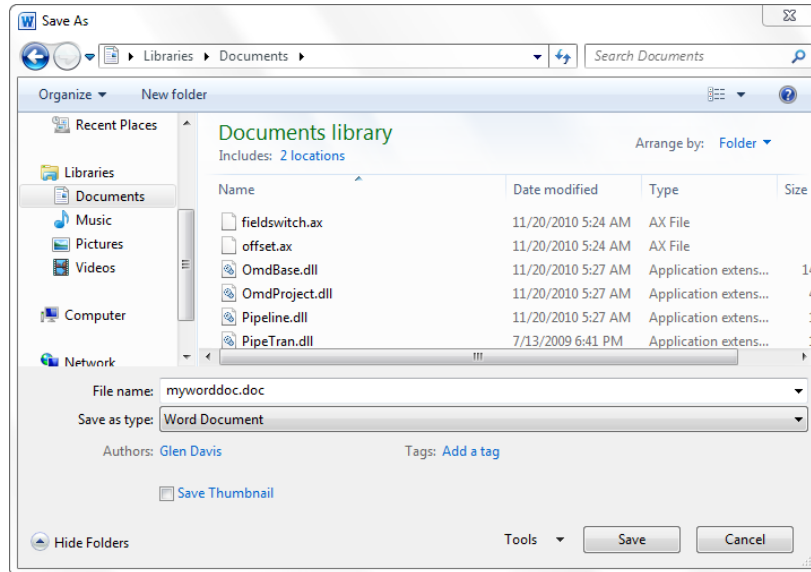
Figure 14-3: The Save As dialog provides two functions: It lets you name your file, and it lets you place it in a directory you choose. Users, however, don't have a clear concept of saving, so the title of the dialog does not match their mental models of the function. Furthermore, if a dialog allows you to name and place a document, you might expect it would allow you to rename and replace it as well. Unfortunately, our expectations are confounded by poor design.

Functionally, this dialog offers two things: It lets users name a file, and it lets them choose which directory to place it in. Both of these functions demand that users have intimate knowledge of the file system and a fair amount of foresight into how they'll need to retrieve the file later. Users must know how to formulate an acceptable and memorable filename and understand the hierarchical file directory. Many users who master the name portion give up on trying to understand the directory tree. They put their documents on their Desktop or in the directory that the application chooses as the default. Occasionally, some action causes the application to forget its default directory, and these users must call in an expert to find their files.

The Save As dialog needs to decide what its purpose truly is. If it is to name and place files, it does a very poor job. After the user has named and placed a file for the first time, he cannot change its name or directory without creating a new document—at least not with this dialog, which purports to offer naming and placing functions. Nor can he do so with any other tool in the application itself. In fact, in Windows 7, he can rename other files using this dialog, but not the ones he is currently working on. Huh? Beginners are out of luck, but experienced users learn the hard way that they can close the document, launch Windows Explorer, rename the file, return to the application, summon the Open dialog from the File menu, and reopen the document.

Forcing the user to go to Explorer to rename the document is a minor hardship, but therein lies a hidden trap. The bait is that Windows easily supports several applications running simultaneously. Attracted by this feature, the user tries to rename the file in the Explorer without first closing the document in the application. This very reasonable action triggers the trap, and the steel jaws clamp down hard on his leg. He is rebuffed with the rude error message box shown in Figure 14-4. Trying to rename an open file is a sharing violation, and the operating system rejects it with a patronizing error message.
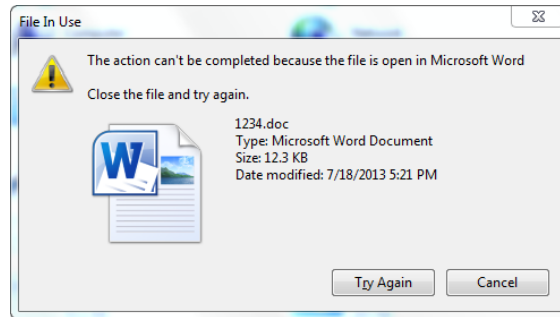
Figure 14-4: If the user attempts to rename a file using Explorer while Word is still editing it, Explorer is too stupid to get around the problem. It is also too rude to be nice about it and puts up this patronizing error message. Rebuffed by both the editing application and the OS, a new user might conclude that a document cannot be renamed.

The innocent user is merely trying to rename his document, and he finds himself lost in operating system arcana. Ironically, the one entity that has both the authority and the responsibility to change the document's name while it is still open—the application itself—refuses to even try.

Archiving

There is no explicit function for making a copy of, or archiving, a document. Users must accomplish this with the Save As dialog, and doing so is as clear as mud. If the user has already saved the file as "Alpha," she must explicitly call up the Save As dialog and change the name. Alpha is closed and put away on disk, and New Alpha is left open for editing. This action makes very little sense from a single-document viewpoint of the world, and it also presents a nasty trap for the user.

Here is a completely reasonable scenario that leads to trouble: Suppose our user has been editing Alpha for the last 20 minutes and now wants to make an archival copy of it on disk so that she can make some big but experimental changes to the original. She calls up the Save As dialog and changes the filename to "New Alpha." The application puts away Alpha on disk, leaving her to edit New Alpha. But Alpha was never saved, so

it gets written to disk without any of the changes she made in the last 20 minutes! Those changes only exist in the New Alpha copy that is currently in memory—in the application. As she begins cutting and pasting in New Alpha, trusting that her handiwork is backed up by Alpha, she is actually modifying the sole copy of this information.

Everybody knows that you can use a hammer to drive a screw or pliers to bash in a nail, but any skilled craftsperson knows that using the wrong tool for the job will eventually catch up with you. The tool will break or the work will be ruined. The Save As dialog is the wrong tool for making and managing copies, and it is the user who will eventually have to pick up the pieces.

Fixing data storage: a unified file model

Properly designed software should treat a document as a single thing, never as a copy on disk and a copy in memory. In this *unified file model*, users should never be forced to confront the computer's internal mechanisms. It is the file system's job to manage writing data between the disks and memory.

The established standard suite of file management for most applications includes Open, Save, and Close commands, and the related Save As, Save Changes, and Open dialogs. Collectively, these dialogs, as we've shown, are confusing for some tasks and are completely incapable of performing other tasks. The following sections describe a different approach to document management that better supports most users' mental models. The user may need to perform several goal-directed tasks on a document; each one should have its own corresponding function:

- Automatic save
- Creating a copy
- Naming and renaming
- Placing and repositioning in the file system
- Specifying the file type
- Reversing changes
- Discarding all changes
- Creating a version
- Communicating status

Automatic save

One of the most important functions every computer user must learn is how to save a document. Invoking this function means taking whatever changes the user has made to the copy in computer memory and writing them to the disk copy of the document. In

the unified model, we abolish all user interface recognition of the two copies, so the Save function disappears from the mainstream interface. That *doesn't* mean it disappears from the application; it is still a necessary operation.

Applications should automatically save documents. For starters, when the user is done with a document and requests the Close function, the application should go ahead and write the changes to disk without stopping to ask for confirmation with the Save Changes dialog.

In a perfect world, this would be sufficient, but computers and software can crash, power can fail, and other unpredictable, catastrophic events can conspire to erase your work. If the power fails before you save, all your changes are lost as the memory containing them scrambles. The original copy on disk will be all right, but hours of work can still be lost. To prevent this from happening, the application must also save the document at intervals during the user's session. Ideally, the application will save every single change as soon as the user makes it—in other words, after each keystroke. For most applications, this is feasible. Another approach is to keep track of small changes in memory and write them to the disk at reasonable intervals.

It's important that this automatic save function does not affect the responsiveness of the user interface. Saving should be either a background process or performed when the user has stopped interacting with the application. Nobody types continuously. Everybody stops to gather his thoughts, or flip a page, or take a sip of coffee. All the application needs to do is wait until the user stops typing for a couple of seconds and then save.

Automatic save will be adequate for almost everybody. However, people who have been using computers for a long time are so paranoid about crashes and data loss that they habitually press Ctrl+S after every paragraph, and sometimes after every sentence. Applications serving these users should have manual save controls, but users should not be *required* to invoke manual saves.

Creating a copy

There should be an explicit function called Create a Copy. The copy will be identical to the original, but it won't be tied to the original in any way. That is, subsequent changes to the original will have no effect on the copy. The new copy of a file named "Alpha" should automatically be given a name with a standard form like "Alpha Copy." If an existing document already has that name, the new copy should be named "Alpha Copy 2." The

copy should be placed in the same directory as the original. A nice option might be to add a time or date stamp at the end of the filename.

It is tempting to envision a dialog that accompanies this command, but there should be no such interruption. The application should take its action quietly, efficiently, and sensibly, without badgering the user with silly dialogs like "Are you sure you want to make a copy?" In the user's mind it is a simple command. If there are any anomalies, the application should make a constructive decision on its own authority.

Naming and renaming

In most applications, when you save a document for the first time, you can choose a name for it. But almost no application lets you rename that file. Sure, you can Save As under another name, but that just makes another file under the new name, leaving the old file untouched under the old name.

The document's name should be shown on the application's title bar. If the user decides to rename the document, he should be able to click the title to edit it in place. What could be simpler and more direct than that? Omnigraffle on OS X is one of the few applications supporting Rename as described here (see Figure 14-5).
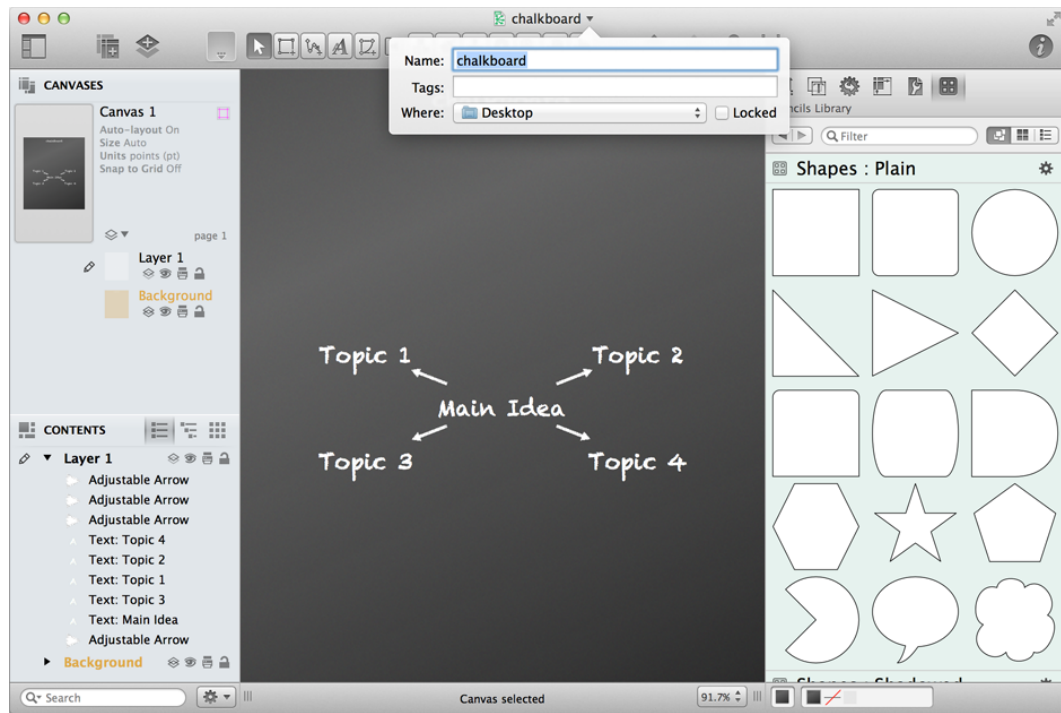
Figure 14-5: Omnigraffle on OS X supports Rename. Clicking on the name of the file in the title bar of the document window opens a pop-up that lets you both rename *and* move the file.

Placing and positioning in the file system

Most often when someone uses an application to edit a document, that document already exists. Documents are typically opened rather than created from scratch. This means that their position in the file system is already established. Although we think of establishing the home directory for a document at the moment of creation or when we first save it, neither of these events is meaningful outside of the implementation model. The new file should be put somewhere reasonable where the user can find it again (such as the Desktop).

DESIGN PRINCIPLE

Put files where users can find them.

The specific appropriate location should depend on your users and the posture of the product you are designing. For complex sovereign applications that most people use daily, it is sometimes appropriate to define an application-specific document location. But for transient applications or sovereign applications that are used less frequently, don't hide your users' files in your own special corner of the file system.

If the user wants to place the document somewhere else, he can request this function from the menu. A Move dialog would then appear with the current document highlighted. In this dialog (an appropriately named relative of the Save As dialog), the user could move the file to any location. The application thus would place all files automatically, and this dialog would be used only to move them elsewhere.

Specifying the file type

At the bottom of the current Save As dialog, shown in Figure 14-3, a combo box allows the user to specify a file type. This function should not be located here. When the type is tied to the act of saving, additional, unnecessary complexity is added to saving. In Word, if the user innocently changes the type, both the save function and any subsequent close action are accompanied by a frightening and unexpected confirmation box. Overriding a file's type is a relatively rare occurrence. Saving a file is a common occurrence. These two functions should not be combined.

From the user's point of view, the document's type—rich text, plain text, or Word, for example—is a characteristic of the document rather than of the disk file. Specifying the type shouldn't be associated with the act of saving the file to disk. It belongs more properly in a Document Properties dialog, accessible near the display of the document's

340   Part II: Making Well-Behaved Products

filename. This dialog should have significant cautions built into its interface to make it clear to the user that the function could involve significant data loss.

In the case of some drawing applications, where saving image files as multiple types is desirable, an Export dialog (which most drawing applications already support) is appropriate for this function.

Reversing changes

If the user inadvertently makes changes to the document that must be reversed, a tool already exists for correcting these actions: Undo (see Chapter 15 for more on Undo behaviors). The file system should not be called in as a surrogate for Undo. The file system may be the mechanism that supports the function, but that doesn't mean it should be rendered to users in those terms. The concept of going directly to the file system to undo changes undermines the Undo function.

The version function, described in the section after the next one, shows how a file-centric vision of Undo can be implemented so that it works well with the unified file model.

Discarding all changes

While it's not the most common of tasks, we certainly want to allow the user to discard all the changes she has made after opening or creating a document, so this action should be explicitly supported. Rather than forcing the user to understand the file system to achieve her goal, a simple Discard Changes function on the main menu would suffice. A similarly useful way to express this concept is Revert to Version, which is based on a version system described in the next section. Because Discard Changes involves significant data loss, the user should be protected with clear warning signs. Making this function undoable also would be relatively easy to implement and highly desirable.

Creating a version

Creating a *version* is very similar to using the Copy command. The difference is that this copy is managed by the application and presented to users as the single document instance after it is made. It should also be clear to users that they can return to the state of the document at each point that a version was made. Users should be able to see a list of versions along with various statistics about them, like the time each was recorded and its size or length. With a click, the user can select a version. By doing so, he also immediately selects it as the active document. The document that was current at the time of the version selection will be created as a version itself. Also, since disk space is hardly a

scarce resource these days, it makes sense to create versions regularly, in case it doesn't occur to your users.

A new File menu

Our new File menu now looks like the one shown in Figure 14-6. It functions as follows:

- New and Open work as before.
- Close closes the document without a dialog or any other fuss after automatically saving changes.
- Rename/Move brings up a dialog that lets the user rename the current file or move it to another directory.
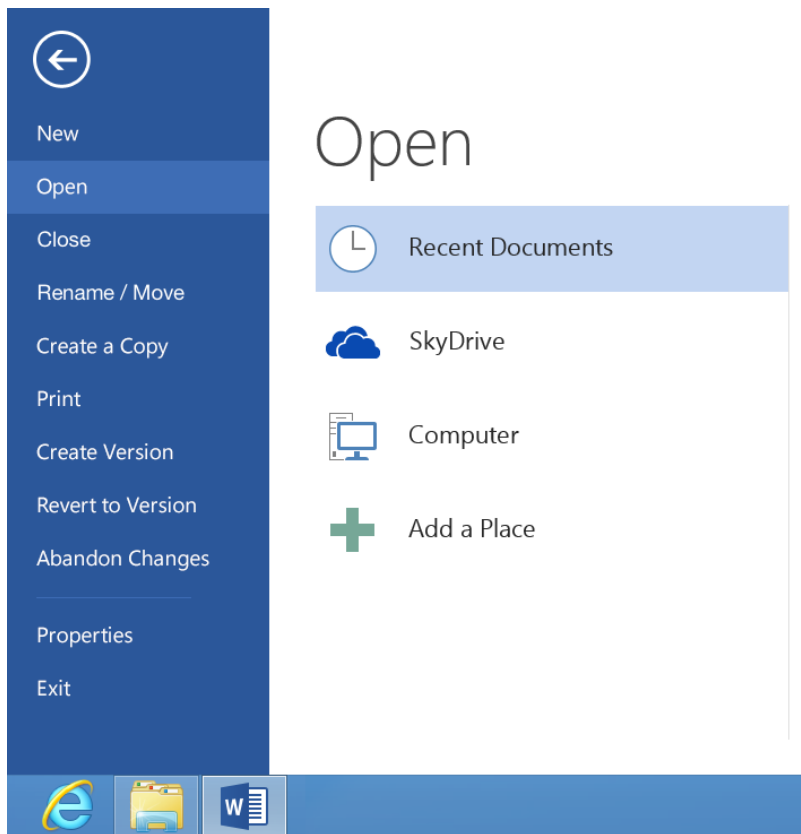
Figure 14-6: The revised File menu now better reflects the user's mental model, rather than the developer's implementation model. There is only one file, and the user owns it. If she wants, she can make tracked or one-off copies of it, rename it, discard any changes she's made, or change the file type. She no longer needs to understand or worry about the copy in RAM versus the copy on disk.

- Create a Copy creates a new file that is a copy of the current document.

- Print collects all printer-related controls in a single dialog.

- Create Version is similar to Copy, except that the application manages these copies by way of a dialog summoned by the Revert to Version menu item.

- Abandon Changes discards all changes made to the document since it was opened or created.

- Properties opens a dialog that lets the user change the document's type.

- Exit behaves as it does now, closing the document and the application.

A new name for the File menu

Now that we are presenting a unified model of storage instead of the bifurcated implementation model of disk and RAM, we no longer need to call the leftmost application menu the File menu—a reflection of the implementation model, not the user's model. There are two reasonable alternatives.

We could label the menu according to the type of documents the application processes. For example, a spreadsheet application might label its leftmost menu Sheet. An invoicing application might label it Invoice.

Alternatively, we could give the leftmost menu a more generic label, such as Document. This is a reasonable choice for applications like word processors, spreadsheets, and drawing applications, but it may be less appropriate for more specialized niche applications.

Conversely, the few applications that do represent the contents of disks as files—generally operating system shells and utilities—should have a File menu because they are addressing files as files.

Communicating status

If the file system needs to show the user a file that cannot be changed because it is in use by another application, the file system should indicate this to the user. Showing the filename in red or with a special symbol next to it, along with a ToolTip explaining the situation, would be sufficient. A new user might still get an error message, as in Figure 14-4, but at least some visual and textual clues would show the reason the error cropped up.

Not only are there two copies of all data files in the current model, but when they are running, there also are two copies of all applications. When the user goes to the Windows taskbar's Start menu and launches his word processor, a button corresponding to Word appears on the taskbar. But if he returns to the Start menu, Word is still there! From the

user's point of view, he has pulled his hammer out of his toolbox only to find that a hammer is still in there.

This should probably not be changed; after all, one of the strengths of the computer is its capability to have multiple copies of software running simultaneously. But the software should help users understand this unintuitive action. The Start menu could, for example, make some reference to the already running application.

Time for a change

If you're a developer, you might be squirming a little in your seat. You might be thinking we are treading on holy ground: A pristine copy on disk is critical, and we'd better not advocate getting rid of it. Relax! There is nothing terribly wrong with the implementation of our file systems. We simply need to hide its existence from users. We can still offer users all the advantages of that extra copy on disk without exploding their mental model.

If we remake the file system's *represented model* according to users' mental models, we can achieve several significant advantages. First, all users will become more effective. If they aren't forced to spend effort and mental energy managing their computer's file system, they'll be more focused on the task at hand. And, of course, they won't have to redo hours of work lost to a mistake in the complex chess game of versioning in contemporary operating systems.

Second, we can teach Mom how to really use computers well. We won't have to answer her pointed questions about the interface's inexplicable behavior. We can show her applications and explain how they allow her to work on the document. Upon completion, she can store the document on the disk as though it were a journal on a shelf. Our sensible explanation won't be interrupted by that Save Changes? dialog. Mom represents the mass market of digital product consumers, who may own and use computers and other devices but who don't like them, trust them, or use them effectively.

The last advantage is that interaction designers won't have to incorporate clumsy file system awareness into their products. We can structure the commands in our applications according to users' goals instead of the operating system's needs.

There will certainly be an initial cost as experienced users get used to the new idioms, but it will be far less than you might suppose. This is because these power users have already shown their ability and tolerance by learning the implementation model. For them, learning the better model will be no problem, and there will be no loss of functionality. The advantages for new users will be immediate and significant. We computer professionals forget how tall the mountain is after we've climbed it, but every day newcomers approach the base of this Everest of computer literacy and are severely discouraged.

Anything we can do to lower the heights they must scale will make a big difference, and this step will tame some of the most perilous peaks.

Rethinking Data Retrieval

One of the most amazing aspects of the modern world is the sheer quantity of information and media we can access within our applications, on our laptops and mobile devices, and via networks and the Internet. But accompanying the boundless possibilities of infinite data access is a difficult design problem: How do we make it easy for people to find what they're looking for and, more importantly, find what they need?

Luckily, great strides have been made in this area by Google, with its various search engines, and Apple, with its highly effective Spotlight functionality in OS X (more on these later). But although these solutions point to some effective interactions, they really just scratch the surface. Google search may be very useful for finding textual, image, or video content on the web, but that doesn't necessarily mean that the same interaction patterns are appropriate for a more targeted retrieval scenario.

As with almost every other problem in interaction design, we've found that crafting an appropriate solution must start with a good understanding of users' mental models and usage contexts. With this information, we can structure storage and retrieval systems that accommodate specific purposes. This chapter discusses methods of data retrieval from an interaction standpoint and presents some human-centered approaches to the problem of finding useful information.

Storage versus retrieval

A *storage system* is a method of safely keeping things in a repository. It is composed of a container and the tools necessary to put objects in and take them back out again. A *retrieval system* is a method of finding things in a repository according to some associated value, such as name, position, or some other attribute of the contents.

In the physical world, storing and retrieving are inextricably linked; putting an item on a shelf (storing it) also gives us the means to find it later (retrieving it). In the digital world, the only thing linking these two concepts is our faulty thinking. Computers enable remarkably sophisticated retrieval techniques—if only we can break our thinking out of its traditional box.

Digital storage and retrieval mechanisms have traditionally been based on the concept of "folders" or "directories." It's certainly true that the folder metaphor has provided a useful way to approach a computer's storage and retrieval systems (much as

one would approach them for a physical object). But as we discussed in Chapter 13, the metaphoric nature of this interaction pattern is limiting. Ultimately, the sole use of folders or directories as a retrieval mechanism requires that users know where an item has been stored in order to locate it. This is unfortunate, since digital systems can give us significantly better methods of finding information than those physically possible using mechanical systems. But before we talk about how to improve retrieval, let's briefly discuss how it works.

Retrieval in the physical world

We can own a book or hammer without giving it a name or permanent place of residence in our house. A book can be identified by characteristics other than a name—a color or shape, for example. However, after we accumulate a large number of items that we need to find and use, it helps to be a bit more organized.

Retrieval by location

It is important that our books and hammers have a proper place, because that is how we find them when we need them. We can't just whistle and expect them to find us; we must know where they are and then go there and fetch them. In the physical world, the actual location of a thing is the means of finding it. Remembering where we put something—its address—is vital to both finding it and putting it away so it can be found again. When we want to find a spoon, for example, we go to the place where we keep our spoons. We don't find the spoon by referring to any inherent characteristic of the spoon itself. Similarly, when we look for a book, we either go to where we left the book, or we guess that it is stored with other books. We don't find the book by association. That is, we don't find the book by referring to its contents.

In this model, the storage system is the same as the retrieval system: Both are based on remembering locations. They are coupled storage and retrieval systems.

Indexed retrieval

Retrieval by location sounds pretty good, but it has a flaw: It's limited in scale by human memory. Although it works for the books, hammers, and spoons in your house, it doesn't work for all the volumes stored in the Library of Congress, for example.

In the world of books and paper on library shelves, we make use of a classification system to help us find things. Using the Dewey Decimal System (or its international offshoot, the Universal Decimal Classification system), every book is assigned a unique "call number" based on its subject. Books are then arranged numerically (and then alphabetically by author's last name), resulting in a library organized by subject.

The only remaining issue is how to discover the number for a given book. Certainly nobody could be expected to remember every number. The solution is an *index*, or a collection of records that allows you to find an item's *location* by looking up an *attribute* of the item, such as its name.

Traditional library card catalogs provide lookup by three attributes: author, subject, and title. When the book is entered into the library system and assigned a number, three index cards are created for the book, including all particulars and the Dewey Decimal number. Each card is headed by the author's name, subject, or title. These cards are then placed in their respective indices in alphabetical order. When you want to find a book, you look it up in one of the indices and find its number. You then find the row of shelves that contains books with numbers in the same range as your target by examining signs. You search those particular shelves, narrowing your view by the lexical order of the numbers until you find the one you want.

You *physically* retrieve the book by participating in the system of storage, but you *logically* find the book you want by participating in a system of retrieval. The shelves and numbers are the storage system. The card indices are the retrieval system. You identify the desired book with one and fetch it with the other. In a typical university or professional library, customers are not allowed into the stacks. As a customer, you identify the book you want by using only the retrieval system. The librarian then fetches the book for you by participating only in the storage system. The book's unique number is the bridge between these two interdependent systems. In the physical world, both the retrieval system and the storage system may be labor-intensive. Particularly in older, noncomputerized libraries, they are both inflexible. Adding a fourth index based on acquisition date, for example, would be prohibitively difficult for the library.

Retrieval in the digital world

Unlike in the physical world of books, stacks, and cards, it's not very hard to add an index in the computer. Ironically, in a system where easily implementing dynamic, associative retrieval mechanisms is at last possible, we often don't implement *any* retrieval system other than the storage system. If you want to find a file on disk, you need to know its name and place. It's as if we went into the library, burned the card catalog, and told the patrons they can easily find what they want by just remembering the little numbers on the spines of the books. We have put 100 percent of the burden of file retrieval on the user's memory while the CPU just sits there idling, twiddling its digital thumbs on billions of NOP (no-operation) instructions.

Although our desktop computers can handle hundreds of different indices, we ignore this capability and frequently have no indices at all pointing into the files stored on our disk. Instead, we have to remember where we put our files and what we called them in order to find them again. This omission is one of the most destructive, backward steps

in modern software design. This failure can be attributed to the interdependence of files and the organizational systems in which they exist—an interdependence that doesn't exist in the mechanical world.

There is nothing wrong with the disk file storage systems that we have created for ourselves. The only problem is that we have failed to create adequate disk file *retrieval* systems. Instead, we hand the user the storage system and call it a retrieval system. This is like handing him a bag of groceries and calling it a gourmet dinner. There is no reason to change our file storage systems. The UNIX model is fine. Our applications can easily remember the names and locations of the files they have worked on, so they aren't the ones who need a retrieval system: It's for us human users.

Digital retrieval methods

There are three fundamental ways to find a document on a digital system. You can find it by remembering where you left it in the file structure—*positional retrieval*. You also can find it by remembering its identifying name—*identity retrieval*. (It should be noted that these two methods typically are used in conjunction.) The third method, *associative* or *attribute-based retrieval*, is based on the ability to search for a document based on some inherent quality of the document itself. For example, if you want to find a book with a red cover, or one that discusses light rail transit systems, or one that contains photographs of steam locomotives, or one that mentions Theodore Judah, you must use an associative method.

The combination of position and identity provides the basis for most digital storage systems. However, most digital systems do not provide an associative method for storage. By ignoring associative methods, we deny ourselves any attribute-based searching, so we must depend on human memory to recall the position and identity of our documents. Users must know the title of the document they want *and* where it is stored to find it. For example, to find a spreadsheet in which you calculated the amortization of your home loan, you need to remember that you stored it in the directory called "Home" and that the file was named "amort1." If you can't remember either of these facts, finding the document is difficult.

Attribute-based retrieval systems

For early graphical interfaces like the original Macintosh, a positional retrieval system almost made sense: The desktop metaphor dictated it (you don't use an index to look up papers on your desk), and precious few documents could be stored on a 144KB floppy disk. However, our current desktop systems can easily hold five million times as many documents (and that's not to mention what even a meager local network can provide access to)! Yet we still use the same old metaphors and retrieval models to manage our

data. We continue to render our software's retrieval systems in strict adherence to the storage system's implementation model of the storage system. We ignore the power and ease of use of a system for *finding* files that is distinct from the system for *keeping* files.

An attribute-based retrieval system enables users to find documents by their contents and meaningful properties (such as when they were last edited). The purpose of such a system is to provide a mechanism for users to express what they're looking for according to how they think about it. For example, a saleswoman looking for a proposal she recently sent to a client named "Widgetco" could effectively express herself by saying "Show me the Word documents related to 'Widgetco' that I modified and printed yesterday."

A well-crafted attribute-based retrieval system also enables users to find what they're looking for by synonyms or related topics or by assigning attributes to individual documents. The user can then dynamically define sets of documents having these overlapping attributes. Returning to our saleswoman example, each potential client is sent a proposal letter. Each of these letters is different and is naturally grouped with the files pertinent to that client. However, there is a definite relationship between each of these letters, because they all serve the same function: proposing a business relationship. It would be convenient if the saleswoman could find and gather all such proposal letters while allowing each one to retain its uniqueness and association with its particular client. A file system based on place—on its single storage location—must necessarily store each document by a single attribute (client or document type) rather than by multiple characteristics.

A retrieval system can learn a lot about each document just by keeping its eyes and ears open. If it remembers some of this information, much of the burden on users is made unnecessary. For example, it can easily remember certain things:

- The user that created or the users that contributed to the document
- The device that created the document
- The application that created the document
- The document's contents and type
- Which application last opened the document
- The document's size and whether it is exceptionally large or small
- Whether the document has been untouched for a long time
- How long the document was last open
- How much information was added or deleted during the last edit
- If the document was created from scratch or cloned from another
- If the document is frequently edited

- If the document is frequently viewed but rarely edited

- Whether the document has been printed and where

- How often the document has been printed, and whether changes were made to it each time immediately before printing

- Whether the document has been faxed, and to whom

- Whether the document has been e-mailed, and to whom

Spotlight, the search function in Apple's OS X, provides effective attribute-based retrieval, as shown in Figure 14-7. Not only can the user look for documents according to meaningful properties, but he can save these searches as "Smart Folders." Doing so enables him to see documents related to a given client in one place and all proposals in a different place. (However, he would have to put some effort into identifying each proposal as such, because Spotlight can't recognize this.) It should be noted that one of the most important factors contributing to Spotlight's usefulness is the speed with which results are returned. This is a significant differentiating factor between it and the Windows search functionality. It was achieved through purposeful technical design that indexes content during idle time.
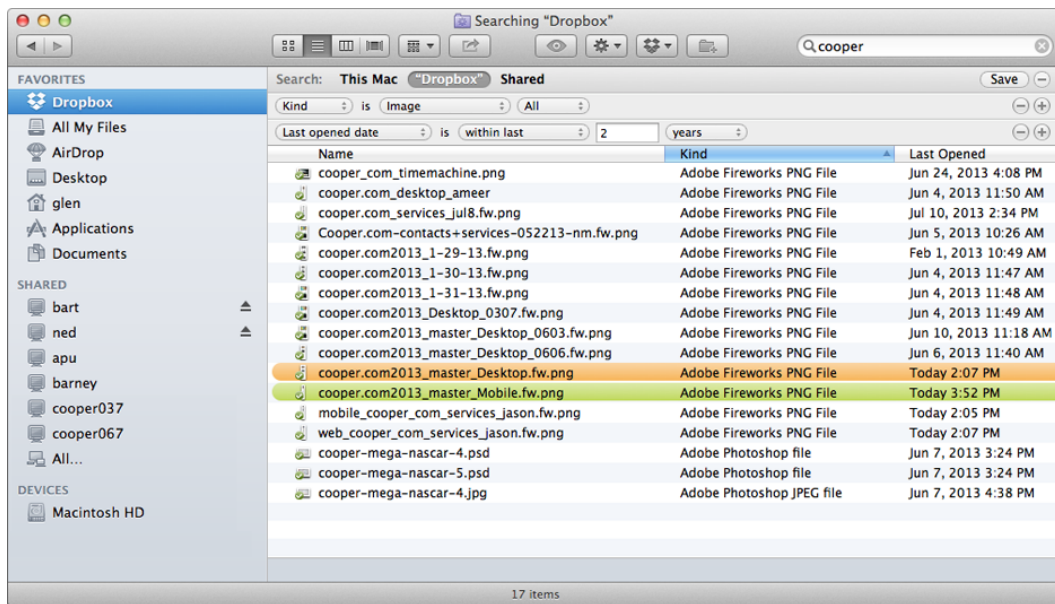
Figure 14-7: Spotlight, the search capability in Apple's OS X, allows users to find a document based on meaningful attributes such as the name, type of document, and when it was last opened.

An attribute-based retrieval system can find documents for users without users ever having to explicitly organize documents in advance. But there is also considerable value in allowing users to *tag* or manually specify attributes about documents. This allows

users to fill in the gaps where technology can't identify all the meaningful attributes. It also allows people to define de facto organizational schemes based on how they discuss and use information. The retrieval mechanism achieved by such tagging is often called a "folksonomy," a term credited to information architect Thomas Vander Wal. Folksonomies can be especially useful in social and collaborative situations. There they can provide an alternative to a globally defined taxonomy if it is undesirable or impractical to force everyone to adhere to and think in terms of a controlled vocabulary. Good examples of the use of tagging to facilitate information retrieval include Flickr, del.icio.us, and the highly popular social sharing app, Twitter (see Figure 14-8).

Figure 14-8: Twitter, whose hashtags have become part of mainstream culture, is the classic example of a folksonomy that has achieved widespread adoption.

Relational databases versus digital soup

Software that uses database technology typically makes two simple demands of its users. First, users must define the form of the data in advance. Second, users must then conform to that definition. There are also two facts about human users of software. First, they rarely can express in advance what they will want. Second, even if they could express their specific needs, more often than not they change their minds.

Organizing the unorganizable

Living in the Internet age, we find ourselves more and more frequently confronting information systems that fail the relational database litmus: We can neither define information in advance nor reliably stick to any definition we might conjure up. In particular, the two most common components of the Internet exemplify this dilemma.

First, let's consider e-mail. Whereas a record in a database has a specific identity, and thus belongs in a table of objects of the same type, an e-mail message doesn't fit this paradigm very well. We can divide our e-mail into incoming and outgoing, but that doesn't help us much. For example, if you receive a piece of e-mail from Jerry about Sally, regarding the Ajax Project and how it relates to Jones Consulting and your joint presentation at the board meeting, you can file this in the "Jerry" folder, or the "Sally" folder, or the "Ajax" folder, but what you really want is to file it in all of them. In six months, you might try to find this message for any number of unpredictable reasons, and you'll want to be able to find it, regardless of your reason.

Second, let's consider the web. Like an infinite, chaotic, redundant, unsupervised hard drive, the web defies structure. Enormous quantities of information are available on the web, but its sheer size and heterogeneity almost guarantee that no regular system could ever be imposed on it. Even if the web could be organized, the method would likely have to exist on the outside, because its contents are owned by millions of individuals, none of whom are subject to any authority. Unlike records in a database, we cannot expect to find a predictable identifying mark in a record on the web.

Problems with databases

Databases have a further problem: All database records are of a single, predefined type, and all instances of a record type are grouped. A record may represent an invoice or a customer, but it never represents an invoice *and* a customer. Similarly, a field within a record may be a name or a social security number, but it is never a name *and* a social security number. This fundamental concept underlies all databases. It serves the vital purpose of allowing us to impose order on our storage system. Unfortunately, it fails miserably to address the realities of retrieval for our e-mail problem. It is not enough that the e-mail from Jerry is a record of type "e-mail." Somehow, we must also identify it as a record of type "Jerry," type "Sally," type "Ajax," type "Jones Consulting," and type "Board Meeting." We must also be able to add and change its identity at will, even after the record has been stored. What's more, a record of type "Ajax" may refer to documents other than e-mail messages, such as a project plan. Because the record format is unpredictable, the value that identifies the record as pertaining to Ajax cannot be stored reliably within the record itself. This directly contradicts how databases work.

Databases do provide retrieval tools that can do a bit more than just match simple record types. They allow us to find and fetch a record by examining its contents and matching

them against search criteria. For example, we search for invoice number "77329" or for the customer with the identifying string "Goodyear Tire and Rubber." Yet this *still* fails for our e-mail problem. If we allow users to enter the keywords "Jerry," "Sally," "Ajax," "Jones Consulting," and "Board Meeting" into the message record, we must define such fields in advance. But as we've said, defining things in advance doesn't guarantee that the user will follow that definition later. He may now be looking for messages about the company picnic, for example. Besides, adding a series of keyword fields leads you into one of the most fundamental and universal conundrums of data processing: If you give users 10 fields, someone is bound to want 11.

The attribute-based alternative

So if relational database technology isn't right, what is? If users find it hard to define their information in advance, as databases require, is there an alternative storage and retrieval system that might work well for them?

Again, the key is separating the storage and retrieval systems. If an *index* were used as the retrieval system, the storage technique could remain a database. We can imagine the storage facility as a sort of *digital soup* where we could put our records. This soup would accept any record we dumped into it, regardless of its size, length, type, or contents. Whenever a record was entered, the application would return a token that could be used to retrieve the record. All we would have to do is give it back that token, and the soup would instantly return our record. This is just our storage system, however; we still need a retrieval system that manages all these tokens for us.

Attribute-based retrieval thus comes to our rescue: We can create an index that stores a key value along with a copy of the token. The real magic, though, is that we can create an infinite number of indices, each one representing its own key and containing a copy of the token. For example, if our digital soup contained all our e-mail messages, we could establish an index for each of our old friends: "Jerry," "Sally," "Ajax," "Jones Consulting," and "Board Meeting." Now, when we need to find e-mail pertinent to the board meeting, we don't have to paw manually and tediously through dozens of folders. Instead, a single query brings us everything we are looking for.

Of course, someone or something must fill those indices, but that is a more mundane exercise in interaction design. Two components must be considered. First, the system must be able to read e-mail messages and automatically extract and index information such as proper names, Internet addresses, street addresses, phone numbers, and other significant data. Second, the system must make it very easy for the user to add ad hoc pointers to messages. He should be able to specify that a given e-mail message pertains to a certain value, whether or not that value is quoted verbatim in the message. Typing is okay, but selecting from picklists, clicking-and-dragging, and other more-advanced user interface idioms can make the task almost painless.

Important advantages arise when the storage system is reduced in importance and the retrieval system is separated from it and significantly enhanced. Some form of digital soup will help us get control of the unpredictable information that is beginning to make up more and more of our everyday information universe. We can offer users powerful information-management tools without demanding that they configure their information in advance or that they conform to that configuration in the future. After all, they can't do it. So why insist?

Constrained natural-language output

This chapter has discussed the merits of attribute-based retrieval. This kind of system, to be truly successful, requires a front end that allows users to easily make sense of what could be complex and interrelated sets of attributes.

One alternative is to use natural-language processing, in which the user can key in his request in English. The problem with this method is that it is still not possible for today's run-of-the-mill computers to effectively understand natural-language queries in most commercial situations. It might work reasonably in the laboratory under tightly controlled conditions, or in the real world for specific domains with tightly controlled vocabulary and syntax, but not in the consumer world, where language is subject to whim, dialect, colloquialism, and ambiguity. In any case, the programming of a natural-language recognition engine is beyond the capabilities and budget of your average development team.

A better approach, which we've used successfully on numerous projects, is a technique we call *constrained natural-language output*. Using this technique, the application provides an array of bounded controls for users to choose from. The controls line up so that they can be read like an English sentence. The user chooses from a list of valid alternatives, so the design is in essence a self-documenting, bounded query facility. Figure 14-9 shows how this works.
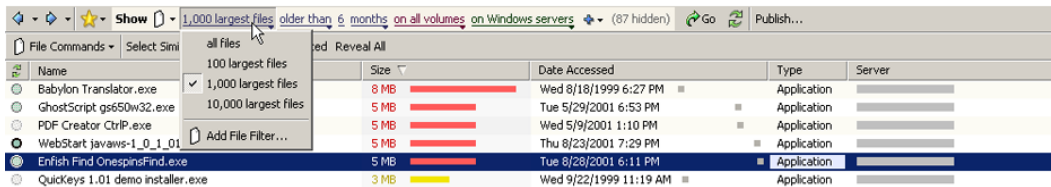
Figure 14-9: An example of a constrained natural-language output interface to an attribute-based retrieval engine, part of a Cooper design created for Softek's Storage Manager. These controls produce natural language as output, rather than attempting to accept natural language as input, for database queries. Each underlined phrase, when clicked, provides a drop-down menu with a list of selectable options. The user constructs a sentence from a dynamic series of choices that always guarantees a valid result.

A natural-language output interface also is helpful for expressing everything from queries to plain old relational databases. Querying a database in the usual fashion is very hard for most people because it calls for Boolean notation and arcane database syntax, à la SQL.

English isn't Boolean, so the English clauses aren't joined with AND and OR, but rather with English phrases like "All of the following apply" or "Not all of the following apply." Users find that choosing from among these phrases is easy because they are clear and bounded, and users can read the phrase like a sentence to check its validity.

The trickiest part of natural-language output from a development perspective is that choosing from controls on the left may, in many circumstances, change the content of the choices in controls to the right of them, in a cascading fashion. This means that to effectively implement natural-language output, the grammar of the choices needs to be well mapped out in advance. Also, the controls need to be dynamically changeable or hideable, depending on what is selected in other controls. Finally, the controls themselves must be able to display or, at least, load data dynamically.

The other concern is localization. If you are designing for multiple languages, those with very different word orders (for example, German and English) may require different grammar mappings.

Both attribute-based retrieval engines and natural-language output interfaces require significant design and programming effort, but users will reap tremendous benefits in terms of power and flexibility in managing their data. Because the amount of data we all must manage is growing at an exponential rate, it makes sense to invest now in these more powerful, goal-directed tools wherever data must be managed.

PREVENTING ERRORS AND INFORMING DECISIONS

In the early days of the digital revolution, a significant portion of a software application's graphical interface was taken up by dialoges and messages telling users what they did wrong, or warning them about what their computer or software was unable to handle due to real or presumed technical limitations. The first edition of *About Face* was released during that time, and was, as you can probably imagine, quite critical of this state of affairs.

Today, the second of these two categories of error messages has dropped by the wayside as computational, storage, and communication speeds have all increased by several orders of magnitude and as the sophistication of programming tools and techniques has similarly advanced.

The first type of error message—scolding users for their mistakes—has begun to disappear as well (at least in consumer and mobile applications). Designers have discovered better ways to eliminate errors before they happen, allow users to reverse their actions, and provide them with the almost magical ability to see the results of their actions before they even take them. These three strategies of preventing errors and informing decisions are the subject of this chapter.

Using Rich Modeless Feedback

Most computers (and, increasingly, many devices) come with high-resolution displays and high-quality audio systems. Yet very few applications (outside of games) even scratch the surface of using these facilities to provide useful information about the app's status, the users' tasks, and the system and its peripherals in general. An entire toolbox is available to supply information to users. But until quite recently, most designers and developers have used the same blunt instrument—the dialog—to communicate information (usually after it is truly useful) to users. We'll discuss in detail the reasoning behind why certain dialogs—errors, alerts, and confirmations—are not appropriate ways to communicate in Chapter 21.

Unfortunately, this means that subtle status information is simply never communicated to users, because most designers know you don't want dialogs to pop up constantly. But constant feedback—especially positive feedback—is exactly what users need. It's simply the channel of communication that needs to be different.

In this section, we'll discuss how visual information, when displayed in a modeless fashion in the main views of an application, won't stop the user's flow and can all but eliminate those pesky dialogs.

Rich visual modeless feedback

Perhaps the most important type of modeless feedback is *rich visual modeless feedback* (RVMF). This type of feedback is rich in terms of giving in-depth information about the status or attributes of a process or object in the current application. It is visual in that it makes idiomatic use of pixels on the screen (often dynamically). It is modeless in that this information is always readily displayed, requiring no special action or mode shift on the user's part to view and make sense of the feedback.

For example, in Microsoft Outlook 2013, an icon next to an e-mail sender's name visually indicates whether that person is available for a chat session or phone call. This is handy when a real-time conversation is preferable to an e-mail exchange. This icon (as well as the ability to start a chat session from a right-click menu) means that users don't have to open their chat client and find the sender's name to see if that person is available. This is so easy and convenient that the user doesn't have to think about it. Another example of the strategy, as designed for a Cooper client, is shown in Figure 15-1.

Here's another example, this time from iOS: When you download an app from the App Store, the downloading file appears on the Home screen as an icon with a small, dynamically updating progress indicator, showing visually how far along the app is in the download and install process (see Figure 15-2).
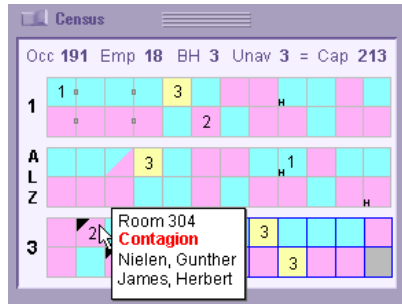
Figure 15-1: This pane from a Cooper design for a long-term health-care information system is a good example of RVMF. The diagram represents all the rooms in the facility. Color coding indicates male, female, empty, or mixed-gender rooms; numbers indicate empty beds; tiny boxes between rooms indicate shared bathrooms. Black triangles indicate health issues, and a tiny H means a held bed. This RVMF is supplemented with ToolTips, which show room numbers and occupant names and highlight any important notices about the room or its residents. A numeric summary of rooms, beds, and employees is given at the top. This display has a short learning curve. After mastering it, nurses and facility managers can understand their facility's status at a glance.
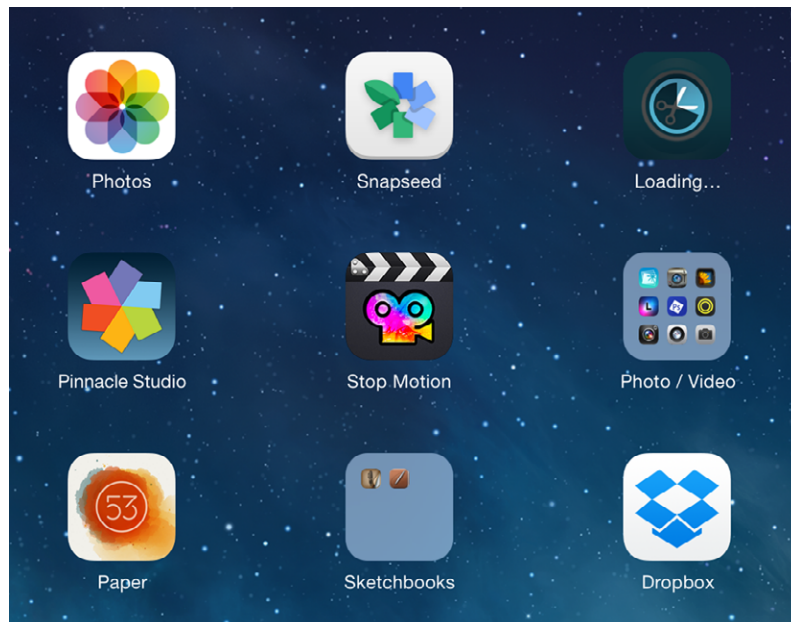
Figure 15-2: When apps are purchased from Apple's App Store, the app icon appears on the Home screen of the iPad or iPhone (upper right). A dynamically updating circular indicator on the icon marks the progress of the download-and-install process.

A final example of RVMF is from the computer gaming world: Sid Meier's Civilization (see Figure 15-3). This game provides dozens of examples of RVMF in its main interface, which is a map of the historical world. You are the leader of an evolving civilization that you are trying to build. Civilization uses RVMF to indicate half a dozen dynamically changing attributes of a city, all represented visually. If a city is more advanced, its architecture is more modern. If it is larger, the icon is larger and more embellished. If it is plagued by civil unrest, smoke rises from the city. Individual troop and civilian units also show status visually, by way of tiny meters indicating unit health and strength. Even the landscape has RVMF: Dotted lines marking spheres of influence shift as units move and cities grow. Terrain changes as roads are laid, forests are cleared, and mountains are mined. Although dialogs exist in the game, much of the information needed to understand what is going on is communicated clearly with no words or dialogs whatsoever.

Figure 15-3: Civilization is a game in which you chart the course of civilization. Its interface provides dozens of examples of rich visual modeless feedback.

Imagine if all the objects that had pertinent status information on your desktop or in your application could display their status in this manner. Printer icons could show how close the printer is to completing your print job. Icons for hard drives and removable media could show how full these items are. When an object is selected for drag and drop, all the places that could receive it would become highlighted to announce their receptiveness.

Think about the objects in your application and their attributes—especially dynamically changing ones—and what kind of status information is critical for your users.

Figure out how to create a representation of this. After the user notices and learns this representation, it tells him what is going on at a glance. (There should also be a way to get fully detailed information if the user requests it.) Put this information into main application windows in the form of RVMF and see how many dialogs you can eliminate from routine use!

One important point needs to be made about rich modeless visual feedback: It isn't for beginners. Even if you add ToolTips to textually describe the details of any visual cues you add (which you should), a ToolTip requires users to perform work to discover it and decode its meaning. RVMF is something that users will begin to use over time. When they do, they'll think it's amazing, but in the meantime, they will need the support of menus and dialogs to find what they're looking for. This means that RVMF used to replace alerts and warnings of serious trouble must be extraordinarily clear to users. Make sure that this kind of status is visually emphasized over less critical, more informational RVMF.

Audible feedback

In data-entry environments, clerks sit for hours in front of computer screens entering data. These users may well be examining source documents and typing by touch instead of looking at the screen. If a clerk enters something erroneous, he needs to be informed of it via both auditory and visual feedback. The clerk can then use his sense of hearing to monitor the success of his inputs while he keeps his eyes on the document.

The kind of auditory feedback we're proposing is not the same as the beep that accompanies an error message box. In fact, it isn't a beep at all. The auditory indicator we propose as feedback for a problem is silence. The problem with much current audible feedback is the still-prevalent idea that, rather than positive audible feedback, negative feedback is desirable.

Avoid negative audible feedback

People frequently counter the idea of audible feedback with arguments that users don't like it. Users are offended by the sounds that computers make, and they don't want their computer beeping at them. Despite the fact that Microsoft and Apple have tried to improve the quality of alert sounds by hiring sound designers (including the legendary Brian Eno for Windows 95), warm ambience doesn't change the fact that sounds are used to convey negative, often insulting messages.

Emitting noise when something bad happens is called negative audible feedback. On most systems, error dialogs normally are accompanied by a shrill beep, so audible feedback has become strongly associated with them. That beep is a public announcement of the user's failure. It explains to all within earshot that you have done something stupid.

It is such a hateful idiom that most software developers now have an unquestioned belief that audible feedback is inappropriate to interface design. Nothing could be further from the truth. It is the negative aspect of the feedback that presents problems, not the audible aspect.

Negative audible feedback has several things working against it. Because the negative feedback is issued when a problem is discovered, it naturally takes on the characteristics of a home alarm. Home alarms are designed to be purposefully loud, discordant, and disturbing. They are supposed to wake sound sleepers from their slumbers when their house is on fire and their lives are at stake. Unfortunately, users are constantly doing things that cause apps to generate error messages, so these noises have become part of the normal course of interaction. Alarms have no place in this normal relationship, just as we don't expect our car alarms to go off if we accidentally change lanes without using the turn signal. Perhaps the most damning aspect of negative audible feedback is the implication that success must be greeted with silence. Humans like to know when they are doing well. They need to know when they are doing poorly, but that doesn't mean that they like to hear about it. Negative feedback systems are simply appreciated less than positive feedback systems.

Given the choice of no noise versus noise for negative feedback, people will choose the former. Given the choice of no noise versus soft and pleasant noises for positive feedback, however, many people will choose the latter. We have never given our users a chance by putting high-quality, positive audible feedback in our apps, so it's no wonder that people associate sound with bad interfaces.

Provide positive audible feedback

Almost every object and system outside the world of software offers sound to indicate success rather than failure. When we close the door, we know that it is latched when we hear the click, but silence tells us that it is not yet secure. When we converse with someone and she says "Yes" or "Uh-huh," we know that, at least minimally, she registered what was said. When she is silent, however, we have reason to believe that something is amiss. When we turn the key in our car's ignition and get silence, we know we have a problem. When we flip the switch on the copier and it stays coldly silent instead of humming, we know that we have trouble. Even most equipment that we consider silent makes some noise: Turning on the stovetop returns a hiss of gas and a gratifying "whoomp" as the pilot ignites the burner. Electric ranges are inherently less friendly and harder to use because they lack that sound—they require indicator lights to tell us their status.

When success with our tools yields a sound, it is called positive audible feedback. Our software tools are mostly silent; all we hear is the quiet clicks of the keyboard. Hey! That's positive audible feedback. Every time you press a key, you hear a faint but positive sound. Keyboard manufacturers could make perfectly silent keyboards, but they don't because

we depend on audible feedback to tell us how we are doing. This is one of the reasons why tablet computers like the iPad provide audible feedback for their touchscreen keyboards by default. The feedback doesn't have to be sophisticated—those clicks don't tell us much—but they must be consistent. If we ever detect silence, we know that we have failed to press the key. The true value of positive audible feedback is that its absence is an extremely effective problem indicator.

The effectiveness of positive audible feedback originates in human sensitivity. Nobody likes to be told that they have failed. Error message boxes are negative feedback, telling the user that he has done something wrong. Silence can ensure that the user knows this without actually being told of the failure. It is remarkably effective, because the software doesn't have to insult the user to accomplish its ends.

Our software should give us constant, small, audible cues just like our keyboards. Our applications would be much friendlier and easier to use if they issued barely audible but easily identifiable sounds when user actions are correct. The app could issue a reassuring click every time the user enters valid input into a field, and an affirming tone when a form has been successfully completed. If an application doesn't understand some input, it should remain silent, subtly informing the user of the problem, allowing her to correct the input without embarrassment or ego bruising. When the user drags and drops an object appropriately, he might be rewarded with a soft, cheerful "plonk" from the speakers for success or with silence (and a visual hop back to its origin point) if the drop was not meaningful.

As with visual feedback, computer games tend to excel at positive audio feedback. Apple's OS X also does a good job with subtle positive audio feedback for activities like document saves and drag and drop. Of course, the audible feedback must be at the right volume for the situation. Windows and the Mac offer a standard volume control, so one obstacle to beneficial audible feedback has been overcome, but audible feedback also should not overpower music playing on the computer.

Rich modeless feedback is one of the greatest tools at the disposal of interaction designers. Replacing annoying, useless dialogs with subtle and powerful modeless communication can make the difference between an app users will despise and one they will love. Think of all the ways you might improve your own applications and prevent user errors with RVMF and other mechanisms of modeless feedback!

Undo, Redo, and Reversible Histories

Undo is the remarkable facility that lets us reverse a previous action, painlessly turning back the clock on our mistakes. Simple and elegant in theory, the feature is of obvious value. Yet when we examine current implementations and uses of Undo from a Goal-Directed point of view, we see considerable variation in purpose and method.

Undo is critically important for users, but it's not quite as simple as it may appear at first glance.

Undo should follow mental models

Undo is traditionally thought of as the rescuer of users in distress, the knight in shining armor, the cavalry galloping over the ridge, the superhero swooping in at the last second. As a computational facility, Undo has no merit. Because they don't make mistakes, computers have no need for Undo. Human beings, on the other hand, make mistakes all the time, and Undo is a facility that exists for their exclusive use. This singular observation should immediately tell us that of all the facilities in an app, Undo should be modeled the least like its construction methods—its implementation model—and the most like the user's mental model.

Not only do humans make mistakes, they make mistakes as part of their everyday behavior. From a computer's standpoint, a false start, a misdirected glance, a pause, a sneeze, some experimentation, an "uh," and a "you know" are all errors. But from a human standpoint, they are perfectly normal. Human "mistakes" are so commonplace that if you think of them as "errors" or even as abnormal behavior, you will adversely affect the design of your software.

User mental models of mistakes

Users generally don't believe, or at least don't want to believe, that they make mistakes. This is another way of saying that the persona's mental model typically doesn't include error on his part. Following a persona's mental model means absolving him of blame. The implementation model, however, is based on an error-free CPU. Following the implementation model means proposing that all culpability must rest with the user. Thus, most software assumes that it is blameless, and any problems are purely the user's fault.

The solution is for the user-interface designer to abandon the idea that the user can make a mistake. This means that everything the user does is something he or she considers to be valid and reasonable. Most people don't like to admit to mistakes in their own minds, so the app shouldn't contradict this mindset in its interactions with users.

Undo enables exploration

If we design software from the point of view that nothing users do should constitute a mistake, we immediately begin to see things differently. We cease to imagine the user as a module of code or a peripheral that drives the computer, and we begin to imagine him as an explorer, probing the unknown. We understand that exploration involves inevitable forays into blind alleys and down dead ends. It is natural for humans to experiment,

to vary their actions, to probe gently against the veil of the unknown to see where their boundaries lie. How can they know what they can do with a tool unless they experiment with it? Of course, the degree of willingness to experiment varies widely from person to person, but most people experiment at least a little bit.

Developers, who are highly paid to think like computers, view such behavior only as errors that must be handled by the code. From the implementation model—necessarily the developer's point of view—such gentle, innocent probing represents a continuous series of "mistakes." From a humanistic perspective based on our users' mental models, these actions are natural and normal. An application can either rebuff those perceived mistakes or assist users in their explorations. Undo is thus a primary tool for supporting exploration in software user interfaces. It allows users to reverse one or more previous actions if they change their mind.

A significant benefit of Undo is purely psychological: It reassures users. It is much easier to enter a cave if you are confident that you can get back out of it at any time. The Undo function is that comforting rope ladder to the surface, supporting the user's willingness to explore further by assuring him that he can back out of any dead-end caverns.

Curiously, users often don't think about Undo until they need it, in much the same way that homeowners don't think about their insurance policies until disaster strikes. Users frequently charge into the cave half prepared and start looking for the rope ladder—for Undo—only when they encounter trouble.

Designing an Undo facility

Although users need Undo, it doesn't directly support any particular goal that underlies their tasks. Rather, it supports a necessary condition—trustworthiness—on the way to a real goal. It doesn't help users achieve their goals, but it keeps negative occurrences from spoiling the effort.

Users visualize the Undo facility in different ways, depending on the situation and their expectations. If a user is very computer-naive, he might see it as an unconditional panic button for extricating himself from a hopelessly tangled misadventure. A more experienced computer user might visualize Undo as a storage facility for deleted data. A really computer-sympathetic user with a logical mind might see it as a stack of procedures that can be undone one at a time in reverse order. To create an effective Undo facility, we must satisfy as many of these mental models as we expect our personas will bring to bear.

The secret to designing a successful Undo system is to make sure that it supports typically used tools and avoids any hint that Undo signals (whether visually, audibly, or textually) a failure by the user. It should be less a tool for reversing errors and more a tool for supporting exploration. Errors are generally single, incorrect actions. Exploration,

by contrast, is a long series of probes and steps, some of which are keepers and some of which must be abandoned.

Undo works best as a global, app-wide function that undoes the last action, regardless of whether it was done by direct manipulation or through a dialog. One of the biggest problems in current implementations of Undo functionality is when users lose the ability to reverse their actions after they save the document (in Excel, for example). Just because the user has saved her work to avoid losing it in a crash doesn't necessarily mean that she wants to commit to all the changes she has made. Furthermore, with our large disk drives, there is no reason not to save the Undo buffer with the document.

Undo can also be problematic for documents with embedded objects. If the user makes changes to a spreadsheet embedded in a Word document, clicks the Word document, and then invokes Undo, the most recent Word action is undone instead of the most recent spreadsheet action. Users have a difficult time with this. It forces them to abandon their mental model of a single unified document and forces them to think in terms of the implementation model: One document is embedded within another, and each has a separate editor with a separate Undo buffer.

Common types of Undo

As is so common in the world of software, there is no adequate terminology to describe the different types of Undo—they are uniformly referred to as "Undo" and left at that. This language gap contributes to the lack of innovation in producing new and better variants of Undo. In this section, we define several Undo variants and explain their differences.

Incremental and procedural actions

Undo operates on the user's actions. A typical user action in a typical application has a procedure component—what the user did—and often a data component—what information was affected. When the user requests an Undo function, the procedure component of the action is reversed. If the action had a data component—resulting in the addition, modification, or deletion of data—that data is modified appropriately. Cutting, pasting, drawing, typing, and deleting are all actions that have a data component, so undoing them involves removing or replacing the affected text or image parts. Actions that include a data component are called *incremental actions*.

Many undoable actions are data-free transformations such as a paragraph reformatting operation in a word processor or a rotation in a drawing app. Both of these operations act on data, but neither of them adds, modifies, or deletes data (from the perspective of the database, although a user may not share this view). Actions like these (with only a procedure component) are *procedural actions*. Most existing Undo functions don't

discriminate between procedural and incremental actions but simply reverse the most recent action.

Blind and explanatory Undo

Normally, Undo is invoked by a menu item or toolbar control with an unchanging label or icon. Users know that triggering the idiom undoes the last operation, but there is no indication of what that operation is. This is called a *blind Undo*. On the other hand, if the idiom includes a textual or visual description of the particular operation that will be undone, it is an *explanatory Undo*.

For example, if the user's last operation was to type the word **design**, the Undo function on the menu says "Undo Typing design." Explanatory Undo is, generally, a much more pleasant feature than blind Undo. It is fairly easy to put on a menu item, but it is more difficult to put on a toolbar control, although putting the explanation in a ToolTip is a good compromise. (See Chapter 18 for more about toolbars and ToolTips.)

Single and multiple Undo

The two most familiar types of Undo in common use today are single Undo and multiple Undo. *Single Undo* is the most basic variant, reversing the effects of the most recent user action, whether procedural or incremental. Performing a single Undo twice usually undoes the Undo and brings the system back to the state it was in before the first Undo was activated.

This facility is very effective because it is so simple to operate. The user interface is basic and clear, easy to describe and remember. The user gets precisely one free lunch. This is by far the most frequently implemented Undo, and it is certainly adequate, if not optimal, for many applications. For some users, the absence of this simple Undo is sufficient grounds to abandon a product.

A user generally notices most of his command mistakes right away: Something about what he did doesn't feel or look right, so he pauses to evaluate the situation. If the representation is clear, he sees his mistake and selects the Undo function to reset things to the previously correct state; then he proceeds.

Multiple Undo can be performed repeatedly in succession. It can undo more than one previous operation, in reverse temporal order—a *reversible history*. Any app with simple Undo must remember the user's last operation and, if applicable, cache any changed data. If the application implements multiple Undo, it must maintain a stack of operations, the depth of which the user may set as an advanced preference. Each time Undo is

invoked, it performs an incremental Undo: It reverses the most recent operation, replacing or removing data as necessary and discarding the restored operation from the stack.

Limitations of single Undo

The biggest limitation of single-level, functional Undo occurs when the user accidentally short-circuits the capability of the Undo facility to rescue him. This problem crops up when the user doesn't notice his mistake immediately. For example, assume he deletes six paragraphs of text, and then deletes one word, and then decides that the six paragraphs were erroneously deleted and should be replaced. Unfortunately, performing Undo now merely brings back the one word, and the six paragraphs are lost forever. The Undo function has failed him by behaving literally rather than practically. Anybody can clearly see that the six paragraphs are more important than the single word, yet the app freely discarded those paragraphs in favor of the one word. The application's blindness caused it to keep a quarter and throw away a fifty-dollar bill, simply because the quarter was offered last.

In some applications, any click of the mouse, however innocent of function it might be, causes the single Undo function to forget the last meaningful thing the user did. Although multiple Undo solves these problems, it introduces some significant problems of its own.

Limitations of multiple Undo

The response to the weaknesses of single-level Undo has been to create a multiple-level implementation of the same incremental Undo. The application saves each action the user takes. When the user selects Undo repeatedly, each action is undone in the reverse order of its original invocation. In the example given in the preceding section, the user can restore the deleted word with the first invocation of Undo and restore the precious six paragraphs with a second invocation. Having to redundantly re-delete the single word is a small price to pay for being able to recover those six valuable paragraphs. The excision of the one-word re-deletion tends to go unnoticed, just as we don't notice the cost of ambulance trips: Don't quibble over the little stuff when lives are at stake. But this doesn't change the fact that the Undo mechanism is built on a faulty model, and in other circumstances, undoing functions in a strict LIFO (last in, first out) order can make the cure as painful as the disease.

Imagine again our user deleting six paragraphs of text, calling up another document, and performing a global find-and-replace function. To retrieve the missing six paragraphs, the user must first unnecessarily undo the rather complex global find-and-replace operation. This time, the intervening operation was not the insignificant single-word deletion of the earlier example. The intervening operation was complex and difficult, and

having to undo it is clearly an unpleasant excision effort. It would sure be nice to be able to choose which operation in the queue to undo and to be able to leave intervening—but valid—operations untouched.

The problems with multiple Undo are not so much due to its behavior as much as they are due to its representation. Most Undo facilities are constructed in an unrelentingly function-centric manner. They remember what the user does function by function and separate her actions by individual function. In the time-honored way of creating represented models that follow implementation models, Undo systems tend to model code and data structures instead of user goals. Each click of the Undo button reverses precisely one function-sized bite of behavior. Reversing on a function-by-function basis is an appropriate mental model for solving most simple problems that arise when the user makes an erroneous entry. The mistake is noticed right away, and the user takes action to fix it right away, usually by the time he's taken two or three actions. However, when the problem grows more convoluted, the incremental, multiple-step Undo model doesn't scale very well.

Undo and Redo

The *Redo* function came into being as the result of the implementation model for Undo, wherein operations must be undone in reverse sequence, and in which no operation may be undone without first undoing all the valid intervening operations. Redo essentially undoes Undo and is easy to implement if developers have already gone to the effort of implementing Undo.

Redo prevents a diabolical situation in multiple Undo. If the user wants to back out of a sequence of actions, he clicks the Undo control a few times, waiting to see things return to the desired state. It is very easy in this situation to press Undo one time too many. The user immediately sees that he has undone something desirable. Redo solves this problem by allowing him to undo the Undo, putting back the last good action.

Many applications that implement single Undo treat the last undone action as an undoable action. In effect, this makes a second invocation of the Undo function a minimal Redo function.

Group multiple Undo

Microsoft Word contains what has unfortunately become a somewhat typical facility—a variation of multiple Undo that we will call *group multiple Undo*. It has several levels, showing a textual description of each operation in the Undo stack. You can examine the list of past operations and select an operation in the list to undo. However, you are not undoing that one operation, but rather all operations back to that point, inclusive (see Figure 15-4). This style of multiple Undo is also employed by many Adobe products.
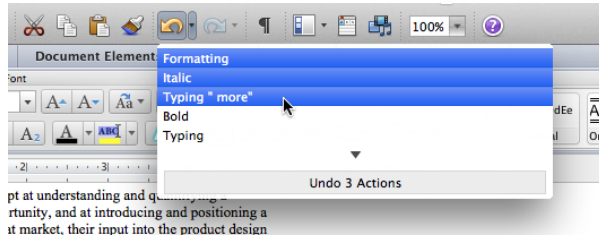
Figure 15-4: With Microsoft Office's Undo/Redo facility, you can undo multiple actions, but only as a group; you can't choose to undo only the thing you did three actions ago. Redo works in the same manner.

As a result, you cannot recover your six missing paragraphs without first reversing all the intervening operations. After you select one or more operations to undo, the list of undone operations becomes available in reverse order in the Redo control. Redo works exactly the same way as Undo. You can select as many operations to redo as you want, and all operations up to that specific one are redone.

The application offers two visual clues to this fact. If the user selects the fifth item in the list, that item and all four items before it in the list are selected. Also, the text legend says "Undo 5 actions." The fact that the designers had to add that text legend tells us that, regardless of how developers constructed it, the users were applying a different mental model. The users imagined that they could go down the list and select a single action from the past to Undo. The app didn't offer that option, so the signs were posted. This is like a door with a pull handle that has a Push sign—which everybody still pulls on anyway. While multiple Undo is certainly a very useful mechanism, there's no reason not to finish the job and use our ample computing resources to allow users to undo just the undesirable actions, instead of everything that has happened since them.

Other types of Undo

Undo in its simplest form—single Undo—conforms to the user's mental model: "I just did something I now wish I hadn't. I want to click a button and undo that last thing I did." Unfortunately, this represented model rapidly diverges from the user's mental model as the complexity of the situation grows. In this section, we discuss models of Undo-like behavior that work a bit differently from the more standard Undo and Redo idioms.

Discontiguous multiple Undo

When the user goes down a logical dead end (rather than merely mistyping data), he can often take several complex steps into the unknown before realizing that he is lost and needs to get a bearing on known territory. At this point, however, he may have performed several interlaced functions, only some of which are undesirable. He may want

to keep some actions and nullify others, not necessarily in strict reverse order. What if he entered some text, edited it, and then decided to undo the entry of that text but not undo the editing of it? Such an operation is problematic to implement and explain. Neil Rubenking offers this pernicious example: Suppose that the user did a global replace, changing *tragedy* to *catastrophe*, and then another changing *cat* to *dog*. To undo the first without undoing the second, can the application reliably fix all the *dogastrophes*?

In this more complex situation, the simplistic representation of Undo as a single LIFO stack doesn't satisfy the way it does in simpler situations. The user may want to study his actions as a menu and choose a discontinuous subset of them for reversion while keeping others. This demands an explanatory Undo with a more robust presentation than might otherwise be necessary for a normal blind multiple Undo. Additionally, the means for selecting from that presentation must be more sophisticated. Representing the operation in the queue to show the user what he is actually undoing is a more difficult problem.

Category-specific Undo

The Backspace key is really an Undo function, albeit a special one. When the user mistypes, the Backspace key "undoes" the erroneous characters. If the user mistypes something, and then performs an unrelated function such as paragraph formatting, and then presses the Backspace key repeatedly, the mistyped characters are erased, and the formatting operation is ignored. Depending on how you look at it, this can be a great, flexible advantage, allowing users to undo discontiguously at any selected location. You could also see it as a trap for users, because they can move the cursor and inadvertently backspace away characters that were not the last ones keyed in.

Logic says that this latter case is a problem. Empirical observation says that it is rarely a problem for users. Such discontiguous, incremental Undo—so hard to explain in words—is so natural and easy to use because everything is visible: Users can clearly see what will be backspaced away. Backspace is a classic example of an incremental Undo, reversing only some data while ignoring other, intervening actions. Yet if you imagined an Undo facility that had a pointer that could be moved and that could undo the last function that occurred where the pointer points, you'd probably think that such a feature would be patently unmanageable and would confuse a typical user. Experience tells us that Backspace does nothing of the sort. It works as well as it does because its behavior is consistent with the user's mental model of the cursor: Because it is the source of added characters, it can also reasonably be the locus of deleted characters.

Granted, Backspace is a special case. But using this concept as a springboard, we could perhaps create different categories of incremental Undo, like a format-Undo function that would undo only previous format commands and other types of *category-specific Undo* actions. If the user entered some text, changed it to italic, entered some more text,

increased the paragraph indentation, entered some more text, and then clicked the Format-Undo button, only the indentation increase would be undone. A second click of the Format-Undo button would reverse the italic operation. Neither invocation of the Format-Undo would affect the content.

What are the implications of category-specific Undo in a nontext application? A drawing app, for example, could have separate Undo commands for pigment application tools, transformations, and cut-and-paste. There is really no reason that we couldn't have independent Undo functions for each particular class of operation.

Pigment application tools include all drawing implements—pencils, pens, fills, sprayers, brushes—and all shape tools—rectangles, lines, ellipses, arrows. Transformations include all image-manipulation tools—shear, sharpness, hue, rotate, contrast, and line weight. Cut-and-paste tools include all lassos, marquees, clones, drags, and other repositioning tools. Unlike the Backspace function in the word processor, undoing a pigment application in a drawing app would be temporal and would work independently of selection. That is, the pigment that is removed first would be the last pigment applied, regardless of the current selection. Western text has an implied reading order from the upper left to the lower right. Deleting from the lower right to the upper left maps to a strong, intrinsic mental model, so it seems natural. In a drawing, no such conventional order exists, so any deletion order other than one based on entry sequence would be disconcerting to users.

A better alternative might be to undo within the current selection only. The user selects a graphic object, for example, and requests a transformation-Undo. The last transformation to have been applied to that *selected object* would be reversed.

Most software users are familiar with incremental Undo and would find a category-specific Undo novel and possibly disturbing. However, the ubiquity of the Backspace key shows that incremental Undo is a learned behavior that users find helpful. If more apps had modal Undo tools, users would soon adapt to them. They would even come to expect them, the way they expect to find the Backspace key on word processors.

Deleted data buffers

As the user works on a document for an extended time, she may want a repository of deleted text. Consider the six missing paragraphs from the earlier example. If they are separated from the user by a couple of complex search and replaces, they can be as difficult to reclaim through Undo as they are to rekey. Our user is thinking, "If the app would just remember the stuff I deleted and keep it in a special place, I could go get what I want directly."

The user is imagining a repository of the data components of her actions, rather than merely a LIFO stack of functions—a *deleted data buffer*. The user wants the missing text without regard to which function got rid of it. The usual manifest model forces her not only to be aware of every intermediate step but also to reverse each one in turn. To create a facility more amenable to our user, we can create, in addition to the normal Undo stack, an independent buffer that collects all deleted text or data. At any time, she can open this buffer as a document and use standard cut-and-paste or click-and-drag idioms to examine and recover the desired text. If the entries in this deletion buffer were headed with simple date stamps and document names, navigation would be simple and visual.

The user can then browse the buffer of deleted data at will, randomly rather than sequentially. Finding those six missing paragraphs would be a simple, visual procedure, regardless of the number or type of complex, intervening steps the user had taken. A deleted data buffer should be offered in addition to the regular, incremental, multiple Undo because it complements it. The data must be saved in a buffer, anyway. This feature would be quite useful in most applications, whether spreadsheet, drawing app, or invoice generator.

Versioning and reversion

Users occasionally want to back up long distances, but when they do, the granular actions are not terrifically important. The need for an incremental Undo remains, but discerning the individual components of more than the last few operations is overkill in most cases. *Versioning* (as we discussed in Chapter 14) simply makes a copy of the entire document the way a camera snapshot captures an image in time. Because versioning involves the entire document, it is typically implemented by direct use of the file system. The biggest difference between versioning and other Undo systems is that the user must explicitly request the version—recording a copy or snapshot of the document. After he has done this, he can safely modify the original. If he later decides that his changes were undesirable, he can return to the saved copy—a previous version of the document.

Many tools exist to support the versioning concept in source code, but this concept is just emerging in the world outside of software development. 37signals' Writeboard, for example (see Figure 15-5), automatically creates versions of a collaborative text document. It allows users to compare versions and, of course, revert to any previous version.
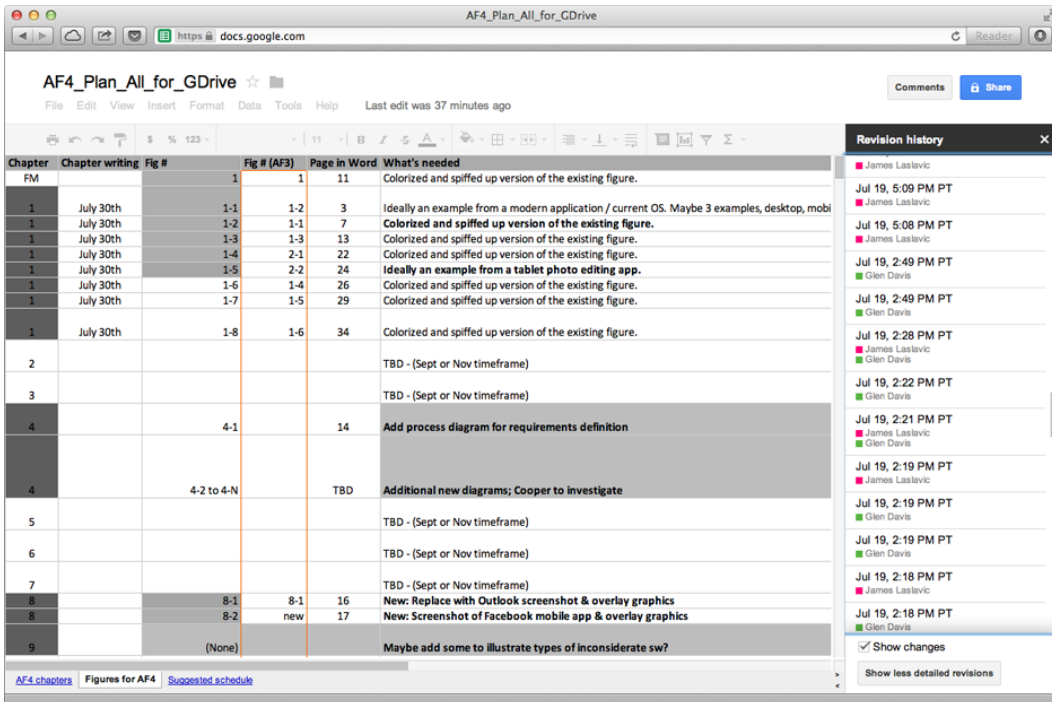
Figure 15-5: Google Docs allows multiple people to collaborate on a single document. It creates a new version every time the user saves changes to the document and allows users to view the different versions. This can be quite useful because it allows collaboration without worry that valuable work will be overwritten.

Critical to the effectiveness of a versioning facility is the behavior of the revert command. It should provide a list of the available saved versions of the document in question. This should include some information about each document, such as the time and day it was recorded, the name of the person who recorded it, the size, and some optional user-entered notes. A user should be able to understand the differences among versions and ultimately choose to revert to any one of these versions. In the case of reversion, the current state of the document should be saved as another version that can be reverted to.

Freezing

Freezing involves locking selected data within a document so that it cannot be changed. Anything that has already been entered becomes uneditable, although new data can be added. Existing paragraphs are untouchable, but new ones can be added between older ones.

This idiom is much more useful for a graphic document than for a text document. It is much like an artist spraying a drawing with fixative. All marks made up to that point are now permanent, yet new marks can be made at will. Images already placed on the screen are locked down and cannot be changed, but new images can be freely superimposed on the older ones. Corel Painter offers a similar feature with its Wet Paint and Dry Paint commands.

Undoing the undoable

Some operations simply cannot be undone because they involve some action that triggers a device not under the application's direct control. For example, after an e-mail message has been sent, there is no undoing it. (Gmail gives you a short amount of time to halt an e-mail by not actually sending it for a few seconds after you click Send, which is really quite clever. See Figure 15-6.)
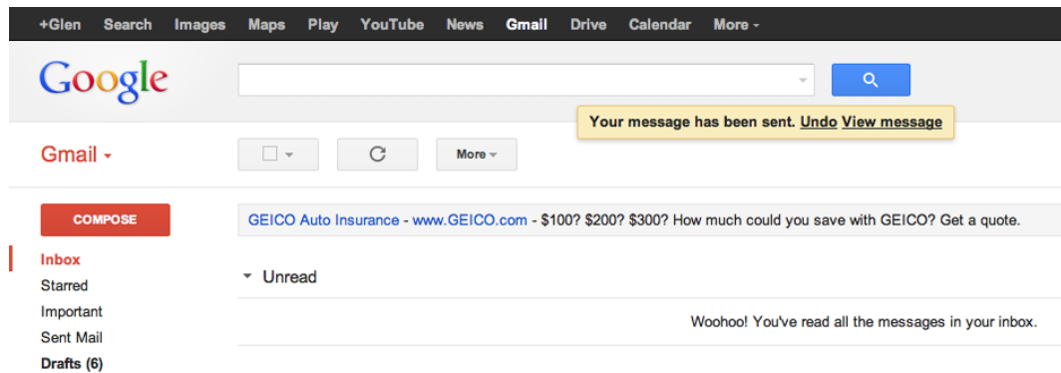
Figure 15-6: Gmail lets you temporarily undo the undoable—sending an e-mail message—by waiting a few seconds after you click Send before really sending it.

Why isn't a filename Undo provided? Because it doesn't fall into the traditional view of what Undo is for; developers generally don't provide a true Undo function for changing a filename.

There are also situations where we're told that it's impossible to undo an action because of business rules or institutional policies. Examples include records of financial transactions and entries in medical charts. In these cases, it may very well be true that Undo isn't an appropriate function, but you can still better support human goals and mental models by providing a way to reverse or adjust the action while leaving an audit trail.

Spend some time looking at your own application and see if you can find functions that seem as if they should be undoable but currently aren't. You may be surprised by how many you find.

What If: Compare and Preview

Besides providing robust support for the terminally indecisive, the paired Undo-Redo function is a convenient comparison tool. Suppose you want to compare the visual effect of ragged-right margins versus justified right margins. You start with ragged right and then invoke Justification. Then you invoke Undo to see ragged right. Then you invoke Redo to see justified margins again. In effect, toggling between Undo and Redo implements a comparison or what-if function; it just happens to be represented in the form of its implementation model. If this same function were added to the interface following a user's mental model, it might be represented as a comparison or *what-if control*. This function would let you compare several states before confirming action.

Some TV remote controls include a Jump button that switches between the current channel and the previous channel—very convenient for viewing two programs concurrently. The Jump function provides the same utility as the Undo-Redo function pair with a single command—a 50 percent reduction in excise (see Chapter 12) for the same functionality.

When used as comparison functions, Undo and Redo are really one function and not two. One says "Apply this change," and the other says "Don't apply this change." A single Compare button might more accurately represent the action to users. Although we have been describing this tool in the context of a text-oriented word processing app, a Compare function might be most useful in an image processing or drawing application, where users apply successive visual transformations. The ability to see the image *with* the transformation (or even multiple variants of it simultaneously) and quickly and easily compare it to the image *without* the transformation would be a great help to the digital artist. Many products address this with thumbnail "preview" images, as shown in Figure 15-7.

Compare may seem like an advanced function, and it is for some applications. Just as the Jump function may not be used by the majority of TV watchers, the Compare button would remain a nicety for frequent users. This shouldn't detract from its usefulness, however. And for some applications, like photo manipulation and other media authoring apps, visual comparison tools that show the future before it happens have become almost a necessity.
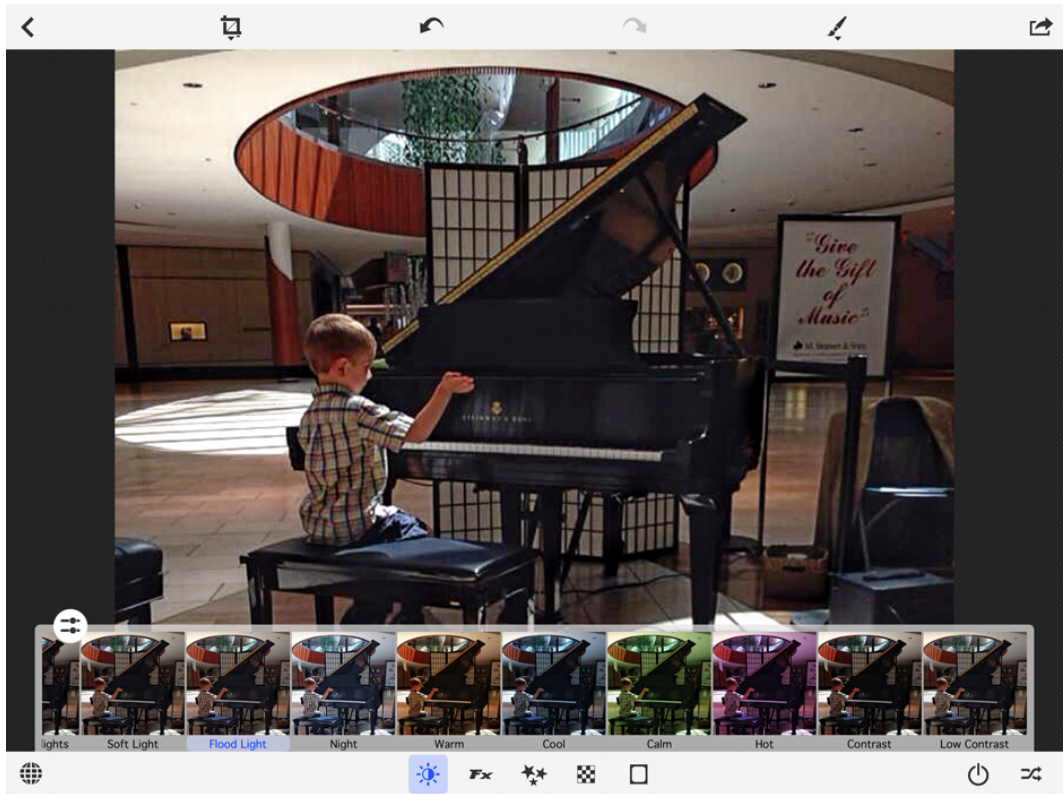
Figure 15-7: Numerous photo processing apps on the iPad, including Photo Toaster, provide preview thumbnails of the image you are working on, each showing the result of a different effect or image parameter change. Tapping the thumbnail applies the change to the image, which is in itself a sort of preview, since it can be undone with a single additional tap.

DESIGNING FOR DIFFERENT NEEDS

As we discussed in Part I, personas and scenarios help us focus our design efforts on the goals, behaviors, needs, and mental models of real users. In addition to the specific focus that personas can give a design effort, some consistent and generalizable patterns of user needs should inform how our products are designed. In this chapter, we'll explore some strategies for serving these well-known needs: *learnability and help*, *customizability*, *localization and globalization,* and *accessibility*.

Learnability and Help

Two concepts are particularly useful in sorting out the needs of users with different levels of experience trying to learn an interface: command modalities and working sets. The fallback option, should these prove insufficient, is online help in its various forms. This section covers each of these methods of helping users understand and learn an interface.

Command modalities

User interfaces are, in a reductionist sense, a means for users to enter data and issue commands to the computer. Data entry is generally fairly straightforward: dictating to a speech recognition algorithm, typing into an empty page or text field, using a finger or stylus to draw, clicking and dragging objects, or picking a value from a menu or similar

widget. Commands that activate functions are a bit more difficult to learn, since users need to figure out both *what* commands are available and *how* they are to be used.

Command modalities are the distinct techniques for allowing users to issue these instructions to the application. Direct-manipulation handles, drop-down and pop-up menu items, toolbar controls, and keyboard accelerators are all examples of command modalities.

Considerate user interfaces often provide *multiple command* modalities for critical functions—menu items, toolbar items, keyboard accelerators, gestures, or direct-manipulation controls—each with the parallel capability to invoke a single, particular command. This redundancy enables users with different skill sets and aptitudes to direct the application according to their abilities and inclinations. Mobile apps have less capacity for multiple modalities, but the tradeoff is that there is usually fewer interface elements to search when looking for a particular function.

Pedagogic, immediate, and invisible commands

Some command modalities offer new users more support. Dialog boxes and command menus (such as those found on a traditional desktop application's menu bar, as shown in Figure 16-1) teach the user with descriptive text. This is why commands presented in this manner express a *pedagogic modality*—commands that *teach* their behavior using inspection. Beginners use the pedagogical behavior of menus as they get oriented in a new application. But perpetual intermediates often want to leave menus behind to find more immediate and efficient tools, in the form of *immediate* and *invisible* commands.

Direct-manipulation controls like drag handles; real-time manipulation controls like sliders and knobs; and even pushbuttons and their toolbar variants, are commands that express an *immediate modality*. Immediate modality controls have an immediate effect on data (or its presentation) without any intermediary. Neither menus nor dialog boxes have this immediate property. Each one requires an intermediate step, sometimes more than one.

Keyboard accelerators and gestures take the idea of immediacy one step further: There is no locus of these commands in the visual interface—only invisible keystrokes, swipes, pinches, or flicks of the finger. These types of command interfaces express an *invisible modality*. Users must memorize invisible commands, because typically the interface offers little or no visual indication that they exist. Invisible commands also need to be initially identified for the user, unless they follow widely used conventions (such as flicking up or down to scroll on a touchscreen interface) or by having a reliable way to inform new users that they exist. Invisible commands are used extensively by intermediates and even more by experts.
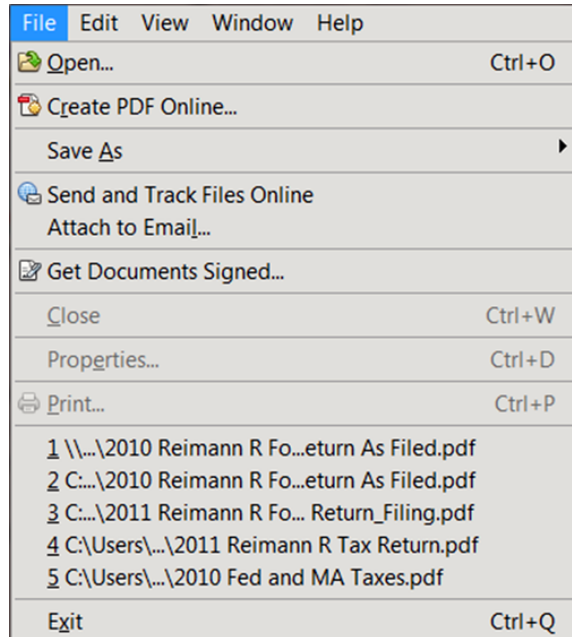
Figure 16-1: Menus on the Windows version of Adobe Reader give users a textual overview of the application's functionality, call out keyboard mnemonics and accelerators, and offer toolbar icons. Unfortunately, this pedagogic idiom is seldom available in mobile apps, due to space constraints.

Information in the world versus information in the head

Donald Norman provides a useful perspective on command modalities. In *The Design of Everyday Things* (Basic Books, 2002), Norman uses the phrases *information in the world* and *information in the head* to refer to different ways that users access information.

When he talks about information in the world, Norman refers to situations in which insufficient information is available in an environment or interface to accomplish something. A kiosk showing a map of downtown, for example, is information in the world. We don't have to bother remembering exactly where the Transamerica Building is, because we can find it by reading a map.

Opposing this is information in your head, which refers to knowledge that you have learned or memorized, like the back-alley shortcut that isn't printed on any map.

Information in your head is much faster and easier to use than information in the world, but you are responsible for ensuring that you learn it, that you don't forget it, and that it stays up to date. Information in the world is slower and more cumbersome, but very dependable.

Pedagogic commands are designed to be learnable via information in the world. Invisible commands must be memorized and thus count as information in the head. Immediate commands fall somewhere in between.

A menu item or dialog is necessarily filled with informational context, which is why it is a *pedagogic command*. Conversely, keyboard accelerators constitute *invisible commands* because using them requires the user to have memorized information about the functions and their keyboard equivalents, which may not be expressed in the visual interface.

Memorization vectors

New users are happy with pedagogic commands, but as they progress to become perpetual intermediates, the slow, repetitive verbosity of pedagogic interfaces starts to seem tedious. Users like to find more immediate commands for frequent tasks. This is a natural and appropriate user desire, and, if our software is to be judged easy to use, we must satisfy it. The solution consists of two components. First, we must provide immediate (or invisible) commands in addition to the pedagogic ones. Second, we must provide a path by which the user can learn the immediate command corresponding to each pedagogic command. This path is called a *memorization vector*.

There are several ways to provide memorization vectors for users. The least effective method is to mention the vector only in the user documentation. The slightly better, but still ineffective, method is to mention it in the app's main online help system. These methods put the onus of finding the memorization vector on users and also leave it up to users to realize that they need to find it in the first place.

Better still is to integrate memorization vectors directly into the main interface. The menus of most desktop applications already have two standard methods. As defined by Microsoft, a typical Windows application has two sets of immediate, keyboard-based commands: mnemonics and accelerators. In Microsoft Word, for example, the *mnemonic* for Save is Alt+F and then S. Alt+F navigates to the File menu and S issues the save command. The memorization vector mnemonics in Windows is shown when the user presses the alt key. The characters are shown with an underline, or—in the case of the Office Suite—with modal tooltips (see Figure 16-2). The user then presses the appropriate key or presses Alt again to hide the hints.

The *accelerator* for Save is Ctrl+S (Cmd+S on the Mac). Accelerators are noted explicitly on the right side of a menu item; this acts as a memorization vector. Adobe Reader, as shown in Figure 16-1, takes this a step further by also including the icons for toolbar commands to the left of their corresponding menu commands. Microsoft, meanwhile, shows accelerators as part of the ToolTips on the controls in the Ribbon UI in their Office Suite applications (see Figure 16-2).
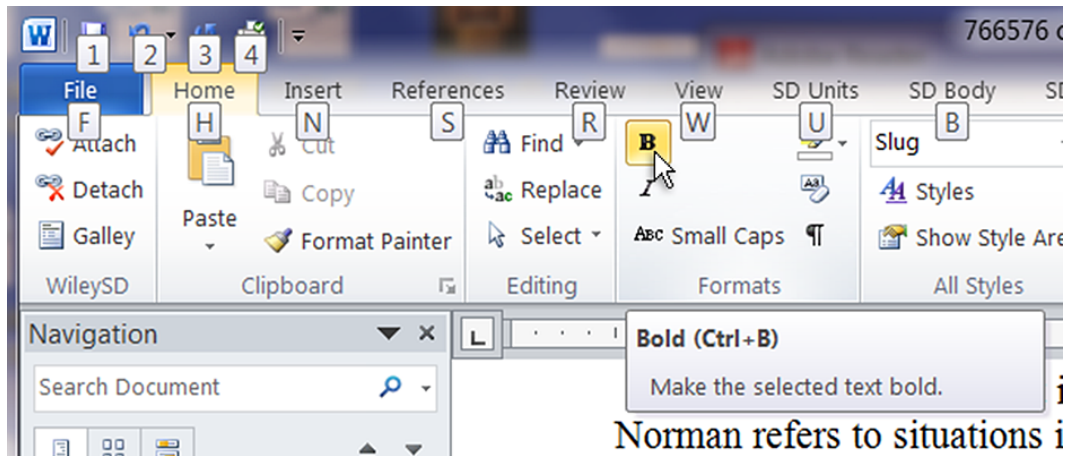
Figure 16-2: Office Suite applications use small pop-up boxes to display mnemonics when Alt is pressed, and ToolTips to display keyboard accelerators, since the standard menus have been replaced by the toolbar-ish Ribbon UI.

Mac applications usually don't support mnemonics, but they often do have accelerator and palette or toolbar icon mappings.

None of these vectors intrudes on a new user. He may not even notice their existence until he has used the app for some time—that is, until he becomes an intermediate user. Eventually, he will notice these visual hints and will wonder about their meaning. Most reasonably intelligent people—most users—will comprehend the accelerator connection without any help. The mnemonic is slightly tougher. But once the user is clued in to the use of the Alt metakey, by either direction or accident, the idiom is easy to remember and use wherever it occurs.

Mobile operating systems, notably, lack common memorization vectors. It may be because there aren't any "spare" real estate or compound interactions (see Chapter 13) in which to place these signals. The closest they come to is the first-run tours (see below) and tutorials that play when the user first uses the device or app. As the mobile platform matures, we are eager to see how designers will help provide this bridge, or whether users are satisfied perpetually using the slow-but-discoverable controls until someone tells them about faster gestures.

As we'll discuss in Chapter 18, *icon buttons* are an excellent technique whereby icons are used to provide memorization vectors for transitioning from menus to toolbars. The icon identifying each function or facility should be shown on every artifact of the user interface that deals with it: each menu, each icon button, each dialog box, every mention in the help text, and every mention in the printed documentation. A memorization vector

formed of visual symbols in the interface is the most effective technique, yet it remains underexploited in the industry at large.

Working sets

Because we all learn (by repetition) the things we do frequently, perpetual intermediates end up memorizing a moderately sized subset of commands and features. We call this set of memorized features a *working set*. The commands that comprise any user's working set are idiosyncratic to that individual, although they will likely overlap significantly with the working sets of other users who exhibit similar use patterns. In Excel, for example, almost every user will enter formulas, specify fonts and labels, and print pages. But Sally's working set might include drawing graphs, whereas Elliot's working set might include linking spreadsheets.

Modeling usage patterns can yield a subset of functions that designers can confidently conclude will be frequently accessed by most users. This *minimal working set* can be determined via usage analytics if you're working with an existing app that provides them and/or Goal-Directed Design methods, using scenarios to discover the functional needs of your personas. These needs translate directly into the contents of the minimal working set.

The commands in any person's working set are those that they use most often. Users want those commands to be especially quick and easy to invoke. This means that the designer should, at the very least, use immediate modality for all commands in the minimal working set of the application's primary users.

Although an application's minimal working set is by definition part of each user's full working set, individual user preferences and job requirements will dictate which additional features are included. Even custom software written for corporate operations can offer a range of features from which each user can pick and choose. This means that the designer, while providing immediate access to the minimal working set, must also provide means for promoting other commands to immediate modality. Similarly, any commands with immediate modality also require duplicate pedagogic versions to enable beginners to learn the interface. This implies that most functions in the interface should have multiple command modalities.

There is an exception to the rule of multiple command modalities: dangerous commands such as Erase All, Clear, and Abandon Changes should not have accidentally-activated or easy, immediate modality commands associated with them. Instead, they need to be protected within menus and dialog boxes (in keeping with our design principle from Chapter 11: *Hide the ejector seat levers*).

Contextual help and assistive interfaces

Needless to say, the best kind of application help is the kind that provides assistance when and where in the interface it is needed, without the user's needing to break his or her flow (see Chapter 11) to seek it out. Whether the situation is the first use of an app or specific to the use of an individual control or feature, a number of patterns support help in context or help users accomplish involved tasks more easily.

Guided tours and overlays

Guided tours and *overlays* are patterns that have become popular on mobile platforms because they provide reasonable solutions to the problem of initial learnability. Since mobile apps must rely more on immediate and invisible command modalities (because there's usually insufficient screen real estate for pedagogic command modalities), tours and overlays fill the need for some sort of pedagogy to bootstrap new users.

These patterns, while more optimized for mobile, have seen increasing use in desktop apps as well. Both try to tackle the problem of introducing a new app to users by providing a brief overview of the most important functions for typical use.

Guided tours provide the needed introduction to features and interface behaviors through a sequential set of screens or cards, each of which contains brief text and images (see Figure 16-3). They either describe a set of basic functions in order of importance or take the user through a typical sequential process, such as creating, editing, and sharing a document using the app. Users advance to the next screen in a tour by swiping or tapping. Tours have a structure somewhat similar to a wizard. The major difference is that, instead of asking for user input to configure something in the app, the sequence of cards, screens, or dialogs exists purely to demonstrate product function and behavior.

OS X has an interesting variant of this in the settings for mouse and trackpad gesture configuration: Rather than showing a sequence of mostly static cards, the UI demonstrates the gestures being configured using short, repeating video clips of hands performing the gestures.

Guided tours usually launch automatically the first time an app is run, and sometimes when a new version of an app is released with significant new features. It's important that tours have a "skip" button available on each screen of the tour, in case the user wants to get straight to work without visiting each screen. Of course, a screen to dismiss the tour at the end also is needed. The final screen of the tour should include a way to manually relaunch the tour.
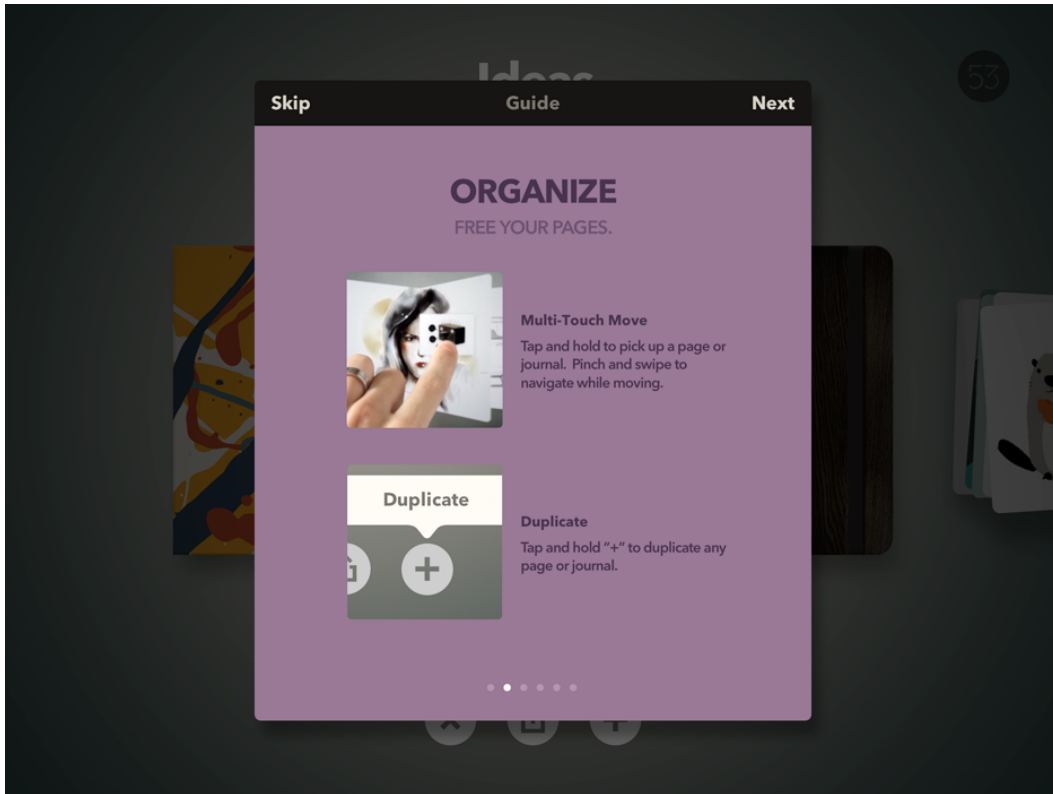
Figure 16-3: FiftyThree Inc.'s iOS app, Paper, uses a guided tour to explain its main features and interactions. The user swipes through a set of illustrated cards, each of which describes a different pair of features or interactions. When the app is opened for the first time, the Welcome tour is available from an About menu accessed by tapping the company logo.

Generally speaking, tours shouldn't go on for more than five to seven screens at most. If you make them too long, your users probably will be unable to remember what they've just seen. They also will begin getting antsy if the tour seems interminable.

Overlays are a different approach to introducing functionality, best suited to relatively simple apps whose functions are not pedagogically evident. As the name implies, an overlay is like a transparent sheet laid over the interface on which arrows and descriptive text are embedded. The end result is a set of annotations that point out the app's key features or behaviors and give brief descriptions of their usage (see Figure 16-4).
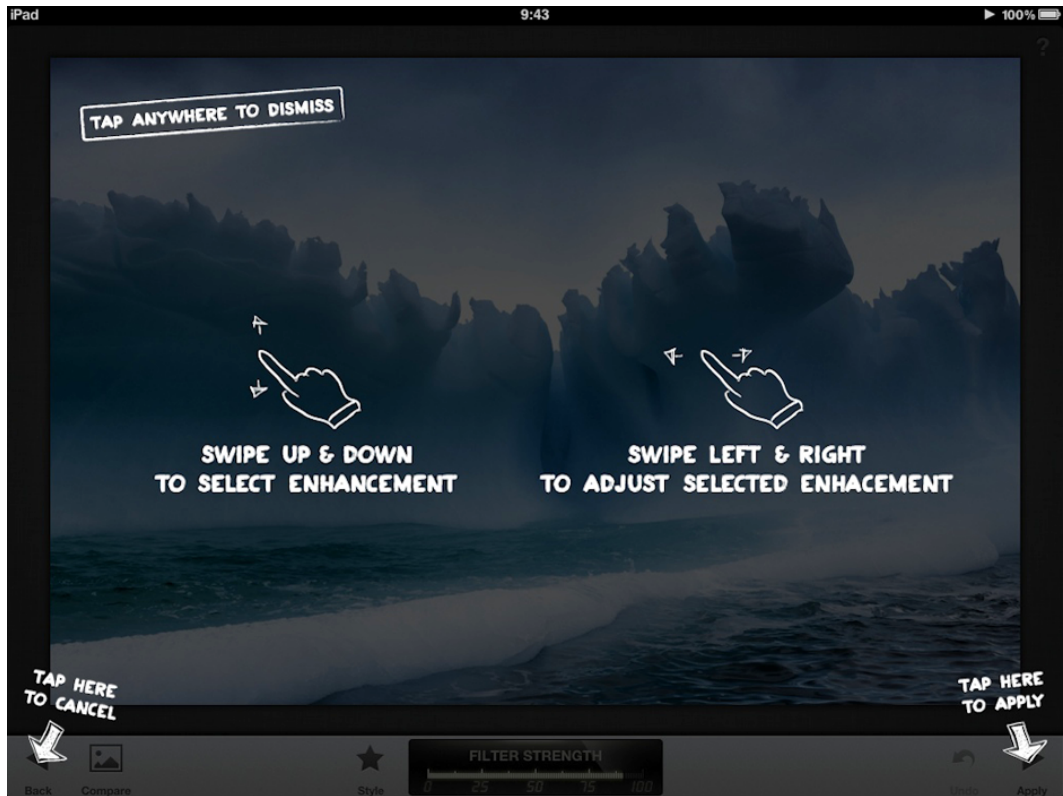
Figure 16-4: The Snapseed app uses an overlay to show key features and behaviors. Unlike some overlays that use a close box, Snapseed's allows you to tap anywhere on the screen to dismiss. After initial use, the overlay remains accessible from the Help menu.

Like guided tours, overlays typically are launched when an app is first run (or when it is updated with a major new release version). An overlay should include a means to relaunch it somewhere in the app—often in a settings menu or via a tiny help icon placed in an unobtrusive corner of the screen.

Zite, shown in Figure 16-5, is a newsreader app that combines the sequential guided tour concept with the idea of an overlay. It walks the user through a series of full-screen overlays accessed by swiping. It ends with a large Done button in the center of the screen.
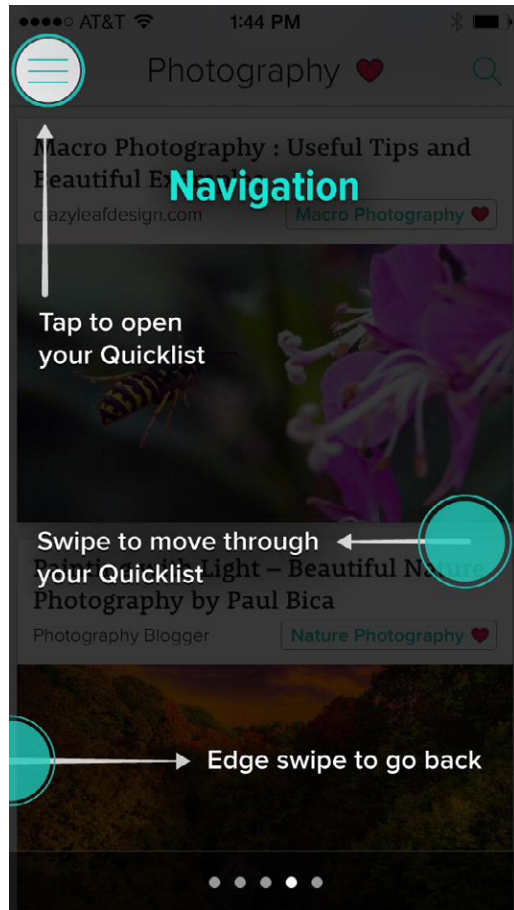
Figure 16-5: Zite is a newsreader app that uses a combination of guided tour and overlay to introduce readers to the app. The tour is available at any time from a tab in the menu system.

This approach is useful in that each feature discussed can be shown in the spatial context of a full screen, potentially making it a bit easier for users to orient themselves.

Galleries and templates

Not all users of document-creation applications are capable of building nicely-formatted documents from scratch. Many apps, however, offer users only atomic tools: the equivalent of hammers, saws, and chisels. That is fine for some users, but others require more: the equivalent of an unfinished table or chair that they can then sand and paint.

For example, consider an app like OmniGraffle on the Mac, shown in Figure 16-6, which lets you create diagrams, flowcharts, and user interface mock-ups.
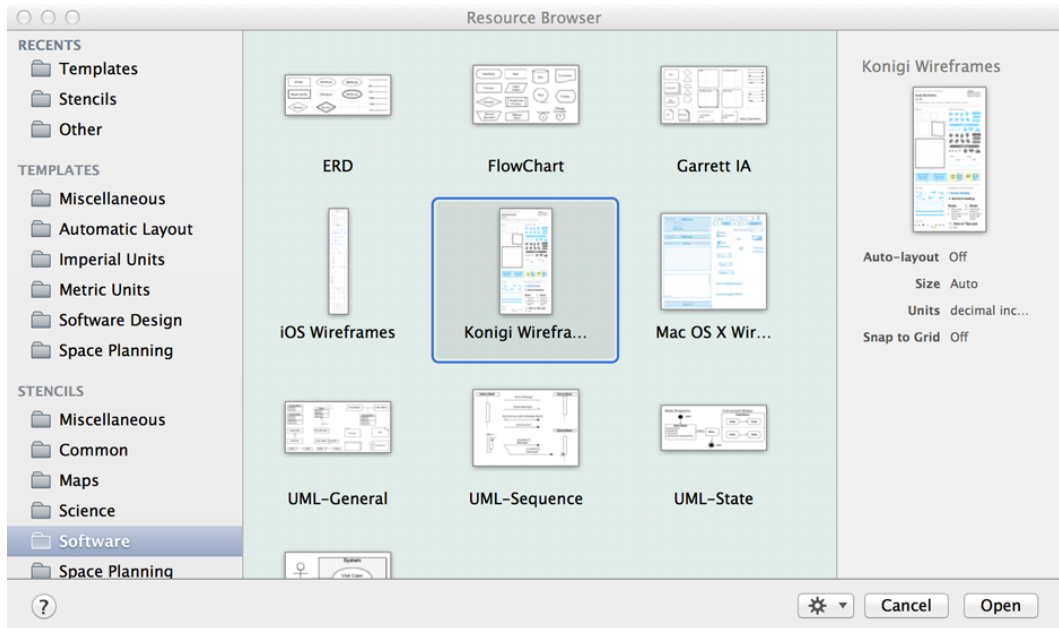
Figure 16-6: OmniGraffle Pro offers galleries of templates at both the document level and the level of line and shape styles.

Undoubtedly some users will want to create their diagrams from scratch, but most would jump at the chance to start with some stylistic choices made for them in the form of layout *templates*. Similarly, some users may want to draw their own shapes for things such as arrows and stars, but most people would be happy to select from a *gallery* of predefined shapes (OmniGraffle calls them stencils). Naturally, users should be able to tweak a template after they choose it.

> **DESIGN PRINCIPLE**
>
> *Offer users a gallery of ready-to-use templates.*

Some applications already offer galleries of predesigned templates (Microsoft's Office and Apple's iWork suites, for example), but more should do the same. Blank slates intimidate most people, and users shouldn't have to deal with one if they don't want to. A gallery of basic document types is a fine solution.

Input and content area hints

A common but subtle form of contextual help is known as *hints*: small and often grayedback text that provides brief directions or examples of use in input fields. This text can live below the input field (usually at a small point size) but is frequently inside the field

before it gets input focus. Once the field gets a cursor in it, the *input hint* text is cleared, and the field is ready for input. An expansion of this idea has become popular in apps that have a larger or central content area that is initially empty. Rather than sit there emptily without lifting a finger to help the user figure out how to get started, clever apps use this empty space to provide a more verbose description of what to do. Or they even provide one-time configuration controls as part of a *content area hint*, as shown in Figure 16-7.

Figure 16-7: Camera+ is an iOS photo app that uses the otherwise empty photo content area at initial launch to provide some verbose hinting and configuration controls.

Pros and cons of wizards

Wizards are an idiom invented by Microsoft that rapidly gained popularity among developers and user-interface designers. A wizard attempts to guarantee success in using a feature by stepping users through a series of steps. These ease the user through a complex

process, typically for configuring a feature of the application, operating system, or connected hardware device.

Each of the wizard's dialogs asks users a question or two in sequence, and in the end the application performs whatever configuration task was requested. Although they are well meaning, the barrage of questions a wizard asks can feel like an interrogation to users, and it violates the design principle of *Provide choices rather than ask questions* (see Chapter 11).

Wizards have other problems, too. Most are written as rigid step-by-step procedures, rather than as intelligent conversations between the user and the application. These sorts of wizards rapidly devolve into exercises in confirmation messaging. The user learns that he merely needs to click the Next button on each screen, without critically analyzing why. Poorly designed wizards also tend to ask obscure questions. A user who doesn't know what an IP address is in a normal dialog will be equally mystified by it in a wizard.

Wizards are appropriate in a few cases. One is during the initial configuration of a hardware device, where registration, activation, or other handshaking between devices and services is required. iPhones and iPads start with a short wizard to select a language and activate various services before releasing the user to the home screen. Similarly, Sonos smart speakers use a wizard to identify a new device added to a whole-home audio network, which requires the controller to detect a button press.

A second appropriate use of the wizard format is for online survey interfaces. Since surveys are a set of questions, a wizard can appropriately break a survey into unintimidating chunks, while providing encouragement using a progress bar.

For most other contexts, a better way to create a wizard is to make a simple, automatic function that asks no questions of users. It just does the job, making reasonable assumptions (based on past behavior, or using well-researched defaults) about proper attributes and organization. The user then can change the output as he or she sees fit using standard tools. In other words, the best wizard is really more like a smart version of a gallery or template.

Wizards were purportedly designed to improve user interfaces. But in many cases they are having the opposite effect. They give developers and designers license to put raw implementation model interfaces on complex features with the bland assurance that "We'll make it easy with a wizard." This is all too reminiscent of the standard abdication of responsibility to users and usability: "We'll be sure to document it in the manual."

ToolTips and ToolTip overlays

ToolTips (see Chapter 18) are an example of modeless interactive help, and they are very effective for desktop or stylus applications. If you were to ask the user to explain how to

perform an interface task, he would likely point to objects on the screen to augment his explanation. This is the essence of how ToolTips work, so the interaction is quite natural.

Unfortunately for mobile interfaces, touchscreens cannot yet support a finger hover state. But most mobile apps don't have enough real estate to permit modeless explanations onscreen while primary interactions are occurring anyway. The mobile solution to this conundrum is a hybrid of desktop-style ToolTip and mobile overlay concepts—ToolTip overlays.

ToolTip overlays are usually triggered by tapping a help button. Brief, ToolTip-like labels or notes for the primary functions on the current screen are displayed, each in proximity and pointing to its associated control (see Figure 16-8). The difference is that they are all turned on at once and presented modally, often with a close box that must be tapped to dismiss them.
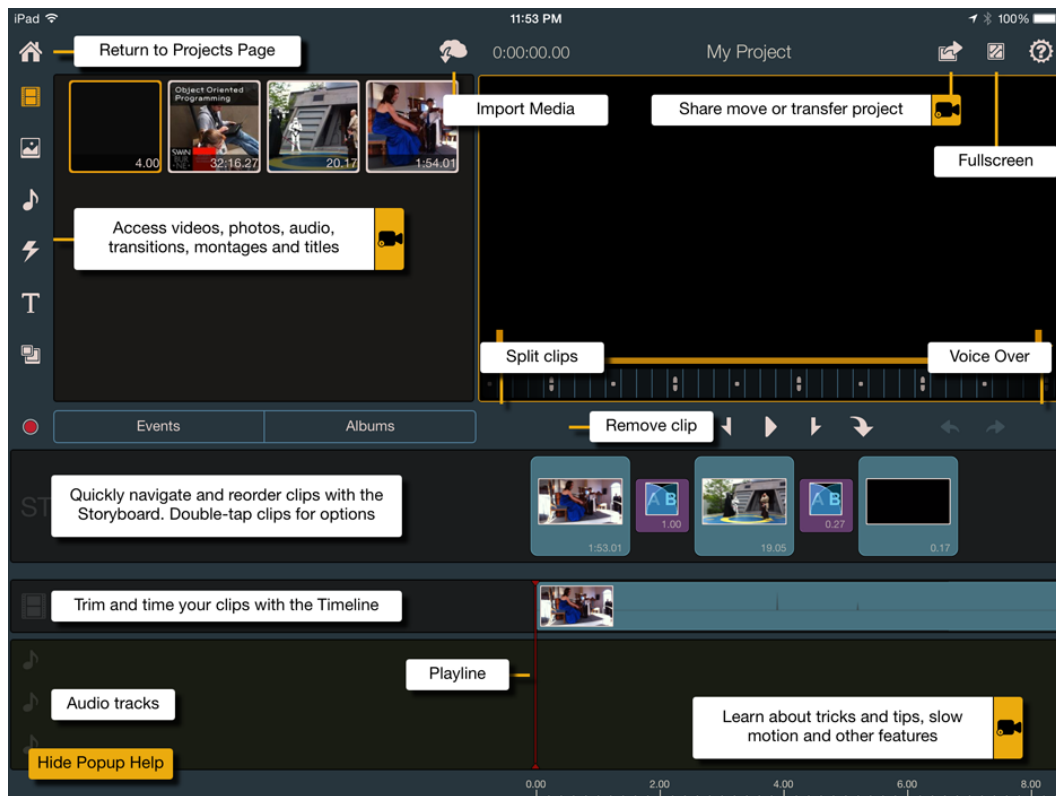
Figure 16-8: Pinnacle Studio has a Tooltip overlay facility, which they call pop-up help. It is launched from the app's help menu. Their implementation is interesting because you can continue to use the app while the pop-up help is activated (not that you'd typically want to); it is dismissed by tapping the yellow button in the lower left corner.

While this approach can be overwhelming, it can be appropriate for complex authoring apps if used as a kind of "cheat sheet" for helping users remember controls and functions. As such, this idiom is best not used as a welcome screen.

Traditional online help

It's important to have guided tours, overlays, or other "quick start" help for beginners. But the more verbose traditional online help should be focused on people who are already successfully using the product and who want to expand their horizons: the perpetual intermediates.

A complex application with many features and functions should come with a reference document: a place where users who want to expand their horizons can find definitive answers. Many users will turn to a general internet search engine to find an answer, and you need to make sure your answer is out there as the definitive one. Printed user manuals can be comfortable to use when users are studying application functionality as a whole, but they are cumbersome for getting quick answers to specific questions. This is the area where online help, with its indexed and full-text search capability, can shine.

Full-text search versus indexing

Although it might be tempting to forgo the extensive work involved in indexing because of the existence of a full-text search capability, there are good reasons to reconsider this decision. Full-text search is only as complete as the wording of the help text itself, and this might not encompass language that reflects users' mental models.

A user who needs to solve a problem might be thinking "How do I turn this cell black?," rather than "How can I set the shading of this cell to 100 percent?" If the help text or its index does not capture a user's way of phrasing or thinking, the help system will fail. It is usually easier to create these kinds of synonym mappings in an index, rather than in the help text itself. This index needs to be generated by examining the app and all its features, not simply by examining the help text itself. This is not always easy, because it demands that a highly skilled indexer also be intimately familiar with each of the application's features and functions.

The set of index entries is arguably as important as the help text itself. The user will forgive a poorly written help entry more easily than he will forgive what he believes to be a missing help entry. The more goal-directed the thinking in creating the text and

the index, the better they will map to what might occur to the user when he searches for an answer.

A great example of a useful index can be found in *The Joy of Cooking* by Irma S. Rombauer and Marion Rombauer Becker (Scribner, 2006). Its index is one of the most complete and robust of any the authors have used.

It may sometimes be easier to rework the interface to improve its learnability than it is to create a really good index. While good online help is very useful and often critical, it should never be a crutch for a poorly designed product. Good design should greatly reduce users' reliance on any help system.

Overview descriptions

The other missing ingredient from most online help systems is the *overview*. For example, if users want to know how the Enter Macro command works, and the help system explains uselessly that it is the facility that lets you enter macros into the system. What we need to know is scope, effect, power, upside, downside, general process, and why we might want to use this facility both in absolute terms and in comparison to similar products from other vendors. Be sure and lead your sections with overviews to explain these fundamental concepts.

In-app user guides

Software applications are increasingly delivered online, without printed manuals, but the need for reference documentation still exists. User guides typically shouldn't be necessary for a well-designed mobile app or a simple desktop application. But for tablet-based authoring tools of particular complexity, such as digital audio workstation and other pro-level media editing apps, or for desktop productivity software of almost any sort, it's useful to provide an integrated user guide that's always accessible from the help menu (see Figure 16-9).

In-app guides should not be the first line of help; that task should be handled by guided tours or overlays. Instead, in-app guides should be a reference for detailed information on using complex functions. If your app is a complex pro tool, your users will appreciate the inclusion of the guide in-app so that they don't have to go looking for it on your website, and even more so if the guide's table of contents is hyperlinked, and the guide itself is full-text searchable, well-indexed, and printable.
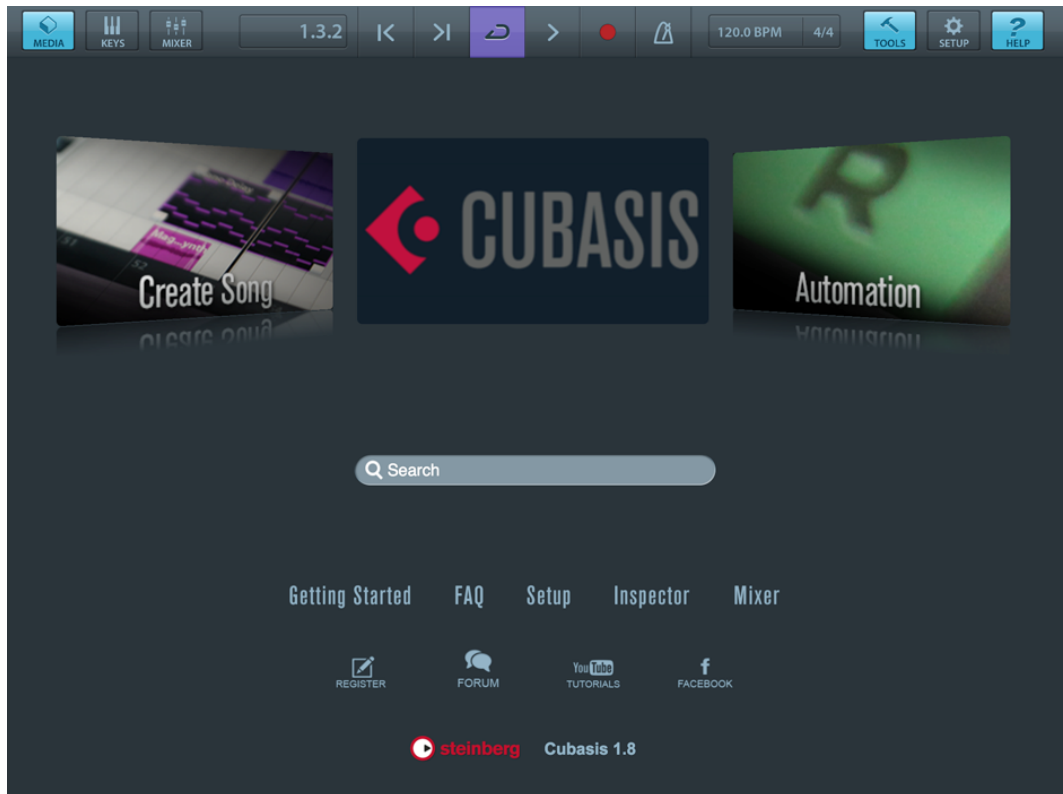
Figure 16-9: Steinberg's Cubasis app has a sophisticated help system that includes a searchable in-app user guide, as well as links to user forums and video tutorials.

Customizability

Interaction designers often face the conundrum of whether to make their products user-customizable. It is easy to be torn between some users' need to have things done their way and the clear problem this creates when the app's navigation suffers due to familiar elements being moved or hidden. The solution is to cast the problem in a different light.

Personalization

People like to change things around to suit themselves. Even beginners, not to mention perpetual intermediates, like to put their own personal stamp on a frequently used application, changing it so that it looks or acts the way they prefer, uniquely suiting their tastes. People will do this for the same reason they fill their identical cubicles with pictures of their spouses and kids, plants, favorite paintings, quotes, and cartoons.

Decorating the persistent objects—the walls—gives them individuality without major structural changes. It also allows you to recognize a hallway as being different from dozens of identical hallways because it is the one with the M. C. Escher poster hanging in it. The term *personalization* describes the decoration or embellishment of persistent objects.

Changing the color of objects on the screen is a personalization task. Windows has always been very accommodating in this respect, allowing users to independently change the color of each component of the Windows interface, including the color and pattern of the desktop itself. Windows gives users a *practical* ability to change the system font, too. Personalization is *idiosyncratically modal* (discussed a bit later in this chapter): People either love it, or they don't. You must accommodate both categories of users.

Tools for personalizing must be simple and easy to use, giving users a visual preview of their selections. Above all, they must be easy to undo. A dialog box that lets users change colors should offer a function that returns everything to the factory settings.

Personalization makes the places in which we work more likable and familiar. It makes them more human and pleasant to be in. The same is true of software. Giving users the ability to decorate their personal applications is both fun and potentially useful as a navigational aid.

On the other hand, moving the persistent objects *themselves* can hamper navigation. If the facilities people come into your office over the weekend and rearrange all the cubicles, finding your office again on Monday morning will be tough. (Persistent objects and their importance to navigation are discussed in Chapter 12.)

Is this an apparent contradiction? Not really. Adding decoration to persistent objects helps navigation, whereas moving persistent objects hinders navigation. The term *configuration* describes moving, adding, or deleting persistent objects.

Configuration

Configuration is desirable for more experienced users. Perpetual intermediates, after they have established a working set of functions, will want to configure the interface to make those functions easier to find and use. They will also want to tune the application itself for speed and ease, but in all cases, the level of custom configuration will be light to moderate.

Configuration is a necessity for expert users. They are already beyond the need for more traditional navigation aids because they are so familiar with the product. Experts may use the application for several hours every day; in fact, it may be the main application for accomplishing the bulk of their job.

Moving controls around on toolbars is a form of personalization. However, the three left-most toolbar controls on many desktop apps, which correspond to File New, File Open, and File Save, are now so common that they can be considered persistent objects. A user who moves these around is *configuring* his application as much as he is personalizing it. Thus, there is a gray area between configuration and personalization.

Most intermediate users won't squawk if they can't configure their app, as long as it does its job well. Some expert users may feel slighted, but they will still use and appreciate the application if it works how they expect. In some cases, however, flexibility is critical. If you're designing for a rapidly evolving workflow, it's of utmost importance that the software used to support the workflow can evolve as quickly as the state of the art.

Also, corporate IT managers value configuration. It allows them to subtly coerce corporate users into practicing common methods or adhere to standards. They appreciate the ability to add macros and commands to menus and toolbars that make the off-the-shelf software work more intimately with established company processes, tools, and software. Many IT managers base their buying decisions on how easily an application can be configured for their enterprise environment. If they are buying 10 or 20 thousand copies of an app, they rightly feel that they should be able to adapt it to their particular style of work. Thus, it is not by accident that Microsoft Office applications are among the most configurable shrink-wrapped software titles available.

Idiosyncratically modal behavior

Often usability testing may show that the user population is split almost equally on the effectiveness of a user interface idiom. Half of the users clearly prefer one idiom, and the other half prefer another. This sort of clear division of a population's preferences into two or more large groups indicates that their preferences are *idiosyncratically modal.*

Development organizations can become similarly emotionally split on issues like this. One group becomes the menu-item camp, and the rest of the developers are in the tool-bar camp. As they wrangle and argue over the relative merits of the two methods, the answer is staring them in the face: Use both!

When the user population splits on preferred idioms, the software designers *must* offer both idioms. Both groups should be satisfied. It is no good to satisfy one-half of the population while angering the other half, regardless of which group you or your developers align yourselves with.

Windows offers an excellent example of how to cater to idiosyncratically modal desires in its menu implementation. Some people like menus that work how they did on the original Apple Macintosh. You click a menu bar item to make the menu appear; then—while

still holding down the button—you drag down the menu and release the mouse button on your choice. Other people find this process difficult and prefer to accomplish it without having to hold down the mouse button while they drag. Windows satisfies this preference by letting users click and release on the menu bar item to make the menu appear. Then users can move the mouse—button released—to the menu item of choice. Another click and release selects the item and closes the menu. The user can also still click and drag to select a menu item. The brilliance of these idioms is that they coexist peacefully with each other. Any user can mix the two idioms or stick with one or the other. No preferences or options need to be set; it just works.

Starting in Windows 95, Microsoft added a third idiosyncratically modal idiom to standard menu behavior: The user clicks and releases as before, but now he can drag the mouse along the menu bar, and the other menus are triggered automatically in turn (they did not carry this behavior over to their Ribbon UI, however). The Mac now supports each of these idioms on its menus; amazingly, all three are accommodated seamlessly.

Localization and Globalization

Localization refers to translating an application for a particular language and culture. *Globalization* refers to making an application as universal as possible across many languages and cultures. Designing applications for use in different languages and cultures presents some special challenges to designers. Here again, consideration of command modalities can provide guidance.

Immediate interfaces such as direct manipulation and toolbar icon buttons are idiomatic (see Chapter 13) and visual rather than textual. Therefore, they can be globalized with considerable ease. Of course, it is important for designers to do their homework to ensure that colors or symbols chosen for these idioms do not have particular meanings in different cultures that the designer does not intend. (In Japan, for example, an X in a check box would likely be interpreted as *deselection* rather than selection.) However, in general, nonmetaphorical idioms should be fairly safe for globalized interfaces.

Pedagogic interfaces such as menu items, field labels, ToolTips, and instructional hints are language-dependent and thus must be the subject of localization via translation into appropriate languages. Here are some issues to bear in mind when creating interfaces that must be localized:

- The words and phrases in some languages tend to be longer than in others. German words, for example, on average can be significantly longer than those in English, and Spanish sentences tend to be some of the longest. Plan button and other text label layouts accordingly, especially on space-constrained mobile devices.

- Words in some languages, Asian languages in particular, can be difficult to sort alphabetically.

- Ordering of day-month-year and the use of 12- or 24-hour notation for time vary from country to country.

- Decimal points in numbers and currency are represented differently. Some countries use periods and commas the opposite of how they are used in the U.S.

- Some countries use week numbers (for example, week 50 is in mid-December), and some countries use calendars other than the Gregorian calendar.

Menu items and dialogs, when they are translated, need to be considered holistically. It is important to make sure that translated interfaces remain coherent as a whole. Items and labels that translate straightforwardly in a vacuum may become confusing when grouped with other independently translated items. Semantics of the interface need to be preserved at the higher level as well as at the detail level.

Accessibility

Designing for accessibility means designing your app so that it can be effectively used by people with cognitive, sensory, or motor impairments due to age, accident, or illness—as well as those without such impairments.

The World Health Organization estimates that 750 million people worldwide have some kind of disability.[1] Nevertheless, accessibility is an area within interaction design—and user experience in general— that is frequently overlooked. Disabled users are often underserved by technology products.

While it is true that not every application may require an accessibility strategy or design, it's safe to say that most enterprise and consumer applications have users for whom accessible interfaces are necessary. This is particularly true for any application targeted at seniors, or at patients suffering from serious or debilitating illnesses.

Goals of accessibility

For a product or service to be considered accessible, it should meet the following conditions for both unimpaired and impaired users:

- Users can perceive and understand all instructions, information, and feedback.

- Users can perceive, understand, and easily manipulate any controls and inputs.

- Users can navigate easily, and always be aware of where there are in an interface and navigational structure.

It's not necessary that these conditions (especially the first two) be accomplished within a single presentation of an interface for all users. A typical accessibility strategy is to design a separate accessibility mode or set of accessibility options. These options alter screen contrast and colors, change the size and weight of text, or turn on a screen reader and audible navigation system.

Accessibility personas

During the research and modeling phase of your design, as part of your accessibility strategy, you might want to consider creating an *accessibility persona* to add to your persona set. Naturally, the ideal method of creating this persona would be to interview users or potential users of your product who have disabilities that would affect their use of the product. If this isn't possible, you can still create a provisional persona to help focus on accessibility issues in a somewhat less targeted way. Typically, an accessibility persona would be considered a secondary persona—someone with needs similar to your primary personas, but with some special needs that must be addressed without compromising their experience. Sometimes, though, selecting an accessibility persona as a primary can result in breakthrough products, as OXO and Smart Design did with their Good Grips product—it turns out that kitchen tools that are optimized for people with arthritis are more satisfying for everyone.

Accessibility guidelines

The following 10 guidelines are not a replacement for exploring the specific needs of disabled users regarding your product and the design trade-offs involved. But they do provide a reasonable starting point for approaching accessible application design. More details about each appear below.

- Leverage OS accessibility tools and guidelines.
- Don't override user-selected system settings.
- Enable standard keyboard access methods.
- Incorporate display options for those with limited vision.
- Provide visual-only and audible-only output.
- Don't flash, flicker, or blink visual elements.
- Use simple, clear, brief language.
- Use response times that support all users.
- Keep layouts and task flows consistent.
- Provide text equivalents for visual elements.

Leverage OS accessibility tools and guidelines

Some operating systems offer accessibility support such as screen readers and audible navigation aids for the vision-impaired, such as VoiceOver for iOS and TalkBack for Android. Your application should be structured to support the use of these OS-level tools and should conform to user interface guidelines for designing and implementing accessibility functionality. Keep in mind the following points:

- Your application shouldn't use keystrokes or gestures that are already committed to enabling OS-level accessibility features for application functions.

- Your application should work properly when accessibility features are turned on.

- Your application should use standard application programming interfaces (APIs) for input and output when possible to ensure compatibility with OS and third-party assistive technologies, such as screen readers.

Don't override user-selected system settings

The application should not override system-level settings that support accessibility options for interface attributes such as color schemes, font sizes, and typefaces. This need not be true in the default application settings, but some accessibility options should revert to OS-level hints for visual style. Similarly, your application should accommodate any system-level accessibility settings for input methods and devices.

Enable standard keyboard access methods

For desktop applications, keyboard accelerators and mnemonics should be employed (see Chapter 18), as well as a rational tab navigation scheme. The user should be able to traverse the entire set of user interface controls and content areas using the Tab key. Arrow keys should let the user traverse list, table, and menu contents. The Enter key should activate buttons and toggles.

Incorporate display options for those with limited vision

Application settings should support a range of options for users with vision problems:

- A high-contrast (minimum 80 percent) display option, using predominantly black text on white backgrounds

- Options for enlarging the typeface and increasing its weight (ideally, these are independent settings)

- An option for color-blind-friendly information display, if applicable

- An option for minimizing motion and animation in UI elements, if these are used in the default interface

Furthermore, your application should not rely on color alone as the sole method of conveying the meaning of data or functionality. It should use other attributes, such as size, position, brightness, shape, and/or textual labeling, to make meaning clear.

Provide visual-only and audible-only output

Support for vision-impaired users should be available in the form of audible interfaces, such as those provided by screen readers and OS-level accessibility services.. Applications should also support redundant visual and audio feedback for hearing-impaired users. Typically, the UI for vision-impaired users is realized as a separate application mode. Support for the hearing-impaired usually can be managed within the standard UI through careful design of the standard user feedback mechanisms to include both audible and visual elements.

Don't flash, flicker, scroll, or blink visual elements

This suggestion is pretty self-explanatory. Flashing and other blinking at speeds greater than twice a second (2 Hz) can be confusing to the vision-impaired. They also can cause seizures in individuals with epilepsy and other brain disorders. Plus, it's usually annoying for everyone else as well. Automatically scrolling text and other animations can be confusing and difficult for those with visual impairments to perceive.

Use simple, clear, brief language

This is another straightforward suggestion that needs little explanation. This is something you should be doing anyway. The shorter and simpler the text labels and instructional text in your interface (as long as they are still appropriately descriptive), the easier they will be to learn and use.

Use response times that support all users

Allow users to elect for longer response times. A good rule of thumb for the longer duration is 10 times the average response times currently in your application. This includes the length of time that transitory notifiers are left visible after they launch. It also should apply to any timers on actions. In general, unless there is a really good reason to time out on an action (such as security), you should ideally avoid doing so, or if you must, make the time-out period user-adjustable.

Keep layouts and task flows consistent

Again, this advice is good for all users. People with cognitive, motor, or vision impairments are best served if they need to remember and execute only a single navigation and action paradigm, rather than several different or incompatible ones. Consider how navigating your interface using a keyboard is likely to work on your screens, and try to make it as consistent as possible across all views and panes.

Provide text equivalents for visual elements

Finally, make sure that any purely visual elements or controls in your desktop application or website are marked with text, so that screen readers can enunciate them. Microsoft Windows, for example, allows invisible ToolTips that can be enunciated by screen readers. They don't appear for users of the default application.

Similarly, it's important for web interfaces to assign tags to visual elements. This allows them to be understood by people using text-based browsers, browser-based voice readers, and other web-based accessibility tools.

Notes
1. World Health Organization, 2003

INTEGRATING VISUAL DESIGN

As an interaction designer, you put a lot of effort into understanding your product's users. You also spend time crafting the interface's behaviors and the presentation of the content that helps users achieve their goals. However, these efforts will fall short unless you also dedicate significant work to clearly communicating to your users both what content is available and how they can interact with it. With interactive products this communication almost always happens visually, via a display. (With custom hardware, you can also communicate some product behavior through physical properties.)

In this chapter, we'll talk about effective, goal-directed visual interface design strategies. In Part III, we will provide more details about specific interaction and interface idioms.

Visual Art and Visual Design

Practitioners of fine art and practitioners of visual design share a visual medium. However, while both must be skilled in and knowledgeable about that medium, their work serves different ends. Art is a means of self-expression on topics of emotional or intellectual concern to the artist and, sometimes, to society at large. Few constraints are imposed on the artist, and the more singular and unique the product of the artist's exertions, the more highly it is valued.

Designers, on the other hand, typically aim to create artifacts with specific utility for the people who use them. Whereas the concern of contemporary artists is primarily self-expression, visual designers are concerned with clear communication. As Kevin Mullet and Darrell Sano note in their book *Designing Visual Interfaces* (Prentice Hall, 1994), "design is concerned with finding the representation best suited to the communication of some specific information." In keeping with a Goal-Directed approach, visual designers should endeavor to present behavior and information in such a way that it is understandable and useful, supporting the organization's branding objectives as well as the personas' goals.

To be clear, this approach does not exclude aesthetic concerns, but rather places such concerns within a goal-directed framework. Although visual communication always involves some subjective judgment, we endeavor to minimize questions of taste. We've found that the clear articulation of user experience goals and business objectives is an invaluable foundation to designing the aspects of an interface in support of brand identity, user experience, and emotional response. (See Chapter 3 for more about visceral processing.)

The Elements of Visual Interface Design

At its root, visual interface design is concerned with the treatment and arrangement of visual elements to communicate behavior and information. Every element in a visual composition has a number of properties, such as shape and color, that work together to create meaning. The ways in which these properties are applied to each element (and how they change over time and with interaction) allow users to make sense of content and the graphical interface. For example, when two interface objects share the same color, users assume they are related or similar. When two objects have contrasting colors, users assume the objects have some categorical difference. Visual interface design capitalizes on the human ability to differentiate between objects by distinct visual appearance, and in so doing creates meaning that is richer than the use of words alone.

When crafting a visual interface, keep in mind the following considerations.

Context, context, context

Every single visual design guideline is subject to the context in which it is used. Are your users doing information work on large-screened desktop computers with overhead lighting? Are they standing in a dark room scanning the screen for the tiniest of biological details? Are they walking across a city holding your design in the glare of the sun? Are they cuddled up on a couch just playing around? Similar to conveying the brand (see below), the context of use must be taken as part of the givens that constrain the visual design.

Shape

Is it round, square, or amoeba-like? Shape is the primary way we recognize *what* an object is. We tend to recognize objects by their outlines; a silhouette of a pineapple that's been textured with blue fur still reads as a pineapple. However, distinguishing among different shapes takes a higher level of attention than distinguishing some other properties, such as color or size. This means it's not the best property to contrast when your purpose is to capture the user's attention. The weakness of shape as a factor in object recognition is apparent to anyone who's glanced at Apple's OS X dock and mistakenly selected the round iTunes icon instead of the round iDVD icon, or latched on to the photo in iWeb and mistook it for iPhoto. These icons have different shapes, but they are of similar size, color, and texture.

Size

How big or small is it in relation to other items on the screen? Larger items draw our attention more, particularly when they're much larger than similar things around them. Size is also an *ordered* and *quantitative* variable, which means that people automatically sequence objects in terms of their size and tend to assign relative quantities to those differences. If we have four sizes of text, for example, we assume that relative importance increases with size, and that bolded content is more important than regular. This makes size a useful property in conveying information hierarchies (more on them in a minute). Sufficient distinction in size is also enough to draw our attention quickly. Be aware that using size can have a cost. In his classic *The Semiology of Graphics* (University of Wisconsin Press, 1983), Jacques Bertin describes size as a *dissociative* property, which means that when something is very small or very large, it can be difficult to decipher other variables, such as shape.

Color

Though most people speak of color loosely, designers must be very precise and deliberate when considering colors in an interface. Any choices should first take into account the users' goals, environment, the content, and the brand. After that, it's most useful to think of interface color in terms of value, hue, and saturation.

Value

How light or dark is it? Of course, the idea of lightness or darkness is meaningful primarily in the context of an object compared to the background. On a dark background, dark type is faint, whereas on a light background, dark type is pronounced. Like size, value can be dissociative. If a photo is too dark or too light, for example, you can no longer perceive other details about it. People perceive contrasts in value quickly and easily, so

value can be a good tool for drawing attention to elements that need to stand out. Value is also an *ordered* variable. For example, lower-value (darker) colors on a map are easy to interpret as deeper water or denser population.

Hue

Is it yellow, red, or orange? Great differences in hue draw our attention quickly, but users often have multilayered associations with hue. In some professions, hue has specific meaning we can take advantage of. For example, an accountant sees red as negative and black as positive, and (at least in the Western systems we're familiar with) a securities trader sees blue as "buy" and red as "sell." Colors also take on meaning from the social contexts in which we've grown up. To Westerners who've grown up with traffic signals, red means "stop" and sometimes even "danger," whereas in China, red is the color of good luck. Similarly, white is associated with purity and peace in the West, but with funerals and death in Asia. Unlike size or value, though, hue is not intrinsically ordered or quantitative, so it's less ideal for conveying that sort of data.

Color is best used judiciously to convey important meaning in an interface. To create an effective visual system that allows users to keep track of implied meanings, you should use a limited number of hues. The "carnival effect" of having a crowded color palette overwhelms users and limits your ability to communicate. Hue is also where an interface's branding and communication needs can collide; it can take a talented visual designer (and skilled diplomat) to navigate these waters. Hue is also tricky since color blindness is common among the general population, and there are many types of color blindness.

Saturation

Is the hue brilliant, like springtime flowers, or dull, like a gray stone? Saturation draws attention similar to the way that hue and value do, that is, when there is a strong contrast at play. The sapphire object will stand out amidst an array of moss green objects. Saturation is *quantitative*, in that greater saturation ties tightly to higher values. Though saturated colors can imply excitement and dynamism, it can also read as loud and cacophonous. The "carnival effect" mentioned above can be exacerbated with too much saturation across the palette, and can compete with actual content.

HSV in combination

Hue, saturation, and value are three variables that together can describe any color in an interface, in a model sensibly named HSV. (Another common system, RGB, lets designers specify the red, green, and blue values for a given color.) Designers should be judicious in how they use contrasts within these variables as well as how they relate across the entire palette.

Orientation

Is it pointing up, down, or sideways? This is a useful variable to employ when you have directional information to convey (up or down, backward or forward). Orientation can be difficult to perceive with some shapes or at small sizes, though, so it's best used as a secondary communication vector. For example, if you want to show that the stock market is going down in a single graphic, you might want to use a downward-pointing arrow that's also red.

Texture

Is it rough or smooth, regular or uneven? Of course, elements on a screen don't have real texture, but they can have the appearance of it. Texture is seldom useful for conveying differences or calling attention, since it requires a lot of attention to distinguish. Texture also takes a fair number of pixels and higher color resolutions to convey. However, it can be an important affordance cue; when we see a textured rubber area on a physical device, we assume that's where we're meant to grab it. Ridges or bumps on a user interface (UI) element generally indicate that it's draggable, and a bevel or drop shadow on a button makes it seem more clickable.

The current fashion for "flat" or non-skeuomorphic design has brought about a diminished use of texture or simulated materiality. But we have found that even in a highly minimalist design, a small amount of texture applied appropriately can greatly improve the learnability of a user interface.

Position

Where is it relative to other elements? Like size, position is both an *ordered* and *quantitative* variable, which means it's useful for conveying information about hierarchy.

We can leverage a screen's *reading order* to locate elements sequentially. For Western readers, this puts the most important or first-used element in the top left. Position can also be used to create spatial relationships between objects on the screen and objects in the physical world, as often happens with medical and piloting interfaces.

Spatial relationships can in turn be used to allude to conceptual relationships: Items that are grouped together on the screen are interpreted to be similar. The use of spatial positioning to express logical relationships can be further enhanced with motion. In the Mail app on iOS, the horizontal animation that transitions from the inbox to an individual e-mail helps reinforce the logical hierarchy that is used to organize the application.

Text and Typography

Text is a critical component of almost all user interfaces. Written language can convey dense and nuanced information, but you must be careful to use text appropriately, because it also has great potential to confuse and complicate. Good and effective typography is its own field of study, but the following are good rules of thumb.

People recognize words primarily by their shapes. The more distinct the shape, the easier the word is to recognize, which is why WORDS TYPED IN ALL CAPITAL LETTERS ARE HARDER TO READ than a mixture of upper and lowercase. They also seem to be shouting. The familiar pattern-matching hints of word shape are absent in capitalized words, so we must pay much closer attention to decipher what is written. Avoid using all caps in your interfaces.

Recognizing words is also different from *reading*, in which we scan the lines of text and interpret the meaning of words in context. This is fine for content of course, but less so for interfaces. Interfaces should try to minimize the amount of text that must be read in order to use it successfully.

> **DESIGN PRINCIPLE**
>
> *Visually show what; textually tell which.*

When text must be read in interfaces, the following guidelines apply:

- **Use high-contrast text**—Make sure that the text contrasts with its background, and do not use complementary colors that may affect readability. We aim for 80 percent contrast as a general rule.

- **Choose an appropriate typeface and size**—In general, a crisp sans-serif font such as Verdana or Tahoma is your best bet for readability. Serif typefaces such as Times and Georgia can appear "hairy" onscreen, but this can be mitigated with very high-resolution displays, using a large-enough size, and sub-pixel font smoothing technologies. Type sizes of less than 10 pixels are difficult to read in most situations. If you must use small type, it's usually best to go with an aliased sans-serif typeface.

- **Phrase your text succinctly**—Make your text understandable by using the fewest words necessary to clearly convey meaning. Also, try to avoid abbreviations. If you must abbreviate, try to use standard abbreviations.

Information hierarchy

When users are presented with a visual interface, they go through an unconscious process of evaluating the most important object or information there, and what the relationships

are between it and the other visible content and controls. To make this decoding process as quick and easy as possible for users, visual designers create an *information hierarchy*, using differences in visual attributes (large vs. small, light vs. dark, top vs. bottom, etc.) to stratify the interface. For transient applications, the information hierarchy should be very apparent, with strong contrasts between the "levels" of importance in a given layout. With sovereign applications, the information hierarchy can be more subtle.

Motion and change over time

Any of the elements mentioned in this section can change over time to convey information, relationship between parts, command attention, ease transitions between modes, and confirm the effects of commands. In iOS for desktop, for example, elements rarely simply appear and disappear. After clicking an application icon in the dock, it will bounce to confirm that the command was received and the application is loading. Minimizing a window doesn't just make it vanish: The "genie" animation shrinks and deforms it, sliding it into a thumbnail position on the dock. The animation confirms that the minimize command was received and tells the user exactly where the window waits, minimized, until the user summons it again. Mastery of how building blocks change over time, and especially motion, is a vital skill for the visual interface designer.

Visual Interface Design Principles

The human brain is a powerful pattern-recognition computer, making sense of the dense quantities of visual information that bombard us everywhere we look. Our brains manage the overwhelming amount of data flowing into our eyes by discerning visual patterns and establishing priorities to the things we see. This in turn allows us to make sense of the visual world. Pattern recognition is what allows us to process visual information so quickly. For example, imagine manually calculating the trajectory of a thrown baseball to predict where it lands. With a pen and paper you'd need a formula and measurements of its path, speed, weight, wind. But our eyes and brains together do it in a split second, without conscious effort on our part. To most effectively convey the behavior of an application to users, visual interface designers should be taking advantage of users' innate visual processing ability.

One chapter is not nearly enough to do justice to the topic of visual interface design. However, some important principles can help make your visual interface more compelling and easier to use. As mentioned earlier in the chapter, Mullet and Sano provide an accessible and detailed analysis of these fundamental principles; we will summarize some of the most important visual interface design concepts here to get you up and running.

Visual interfaces should do the following:

- Convey a tone / communicate the brand
- Lead users through the visual hierarchy
- Provide visual structure and flow at each level of organization
- Signal what users can do on a given screen
- Respond to commands
- Draw attention to important events
- Build a cohesive visual system to ensure consistency across the experience
- Minimize the amount of visual work
- Keep it simple

We discuss each of these principles in more detail in the following sections.

Convey a tone/communicate the brand

More and more, interactive systems are the main way through which customers experience brands. So while the brand considerations should never override users' goals, an effective interface should embody the brand promise of its product line and organization. Photoshop feels similar to the Creative Suite and fits in with the Adobe brand. Outlook feels similar to the rest of the Office Suite, and helps distinguish Microsoft from competitor's products.

It's therefore necessary for you to understand what that brand promise is before undertaking the design of an interface. This can be tricky if the company doesn't have it well articulated. It's rarely if ever communicated explicitly in marketing and advertising materials, and younger or smaller companies may not have had the chance to identify what it is. Larger and public companies almost always have a marketing or design department able to provide it or are willing to work with interaction designers to build one out.

Cooper works with its clients to help identify *experience attributes*, a collection of a handful of adjectives that together describe how any interaction through the product or service should feel (see Chapter 5 for a discussion of how these are created). These attributes are often presented as a "word cloud" that includes smaller words that help inflect or disambiguate the attributes themselves. Once determined, the attributes act as a set of guidelines for the interface design. Most often this directly affects the visual design, but can also be used to guide interaction designers when deciding between functionally similar designs.

The attributes sometimes express tension between its words. "Secure" and "nimble" might be in the same cloud, for example. These are useful tensions, as early style studies

can optimize for one or two of the experience attributes. This makes them in turn easier to distinguish and discuss with stakeholders, and shows how each relates to the brand.

Lead users through the visual hierarchy

In looking at any set of visual elements, users unconsciously ask themselves "What's important here?" followed almost immediately by "How are these things related?" We need to make sure our user interfaces provide answers to both of these questions by creating hierarchy and establishing relationships.

Based on scenarios, determine which controls and bits of data users need to understand instantly, which are secondary, and which are needed only by exception. This ranking informs the visual hierarchy.

Next use the basic visual elements (position, color, size, etc.) to distinguish levels of hierarchy. The most important elements could be larger; have greater contrast in hue, saturation, and/or value in relation to the background; and be positioned above and indented or outdented in relation to other items. Less important elements could be less saturated, have less value and hue contrast against the background, and should be smaller than and placed in consistent alignment with other items.

Of course, you should adjust these properties with restraint, since the most important element doesn't need to be huge, red, and outdented. Often, varying just one of these properties does the trick. If you find that two items of different importance are competing for attention, it's often a good approach to "turn down" the less important one, rather than "turn up" the more important. This will leave you with more "headroom" to emphasize critical elements. Think about it this way: If every word on a page is red and bold, do any of them stand out?

Establishing a clear visual hierarchy is one of the harder tasks in visual interface design. It takes skill and talent to keep an overall style, optimize information density, and respect the needs of the particular screen. Though users almost never notice good visual hierarchy, a bad one will jump out for its confusion and difficulty.

Establish relationships

To convey which elements are related, return to your scenarios to determine not only which elements have similar functions but also which elements are used together most often. Elements that tend to be used together generally should be grouped *spatially* and perhaps *sequentially* to reinforce conceptual relationships and to minimize mouse movement. Elements that aren't necessarily used together but have similar functions may be grouped *visually* even if they are not grouped spatially.

Items in proximity to one another generally are related. In many interfaces, this grouping is done in a heavy-handed fashion, with bounding boxes everywhere you look, sometimes even around just one or two elements. In many cases, you can accomplish the same thing more effectively with differences in *proximity*. For example, on a toolbar, perhaps there are 4 pixels between buttons. To group the File commands, such as Open, New, and Save, you could simply leave 8 pixels between the File command buttons and other groups of buttons.

So group items that are not adjacent by giving them common visual properties, forming a pattern that eventually takes on meaning for users. The standard blue links in HTML, for example, make it easy for a user to parse a screen for content-related navigation options.

After you have decided what the groups are and how best to communicate them visually, consider how distinguishable they need to be, and how prominent the group needs to appear in the display.

Occasionally, squint at it

A good way to help ensure that a visual interface design employs hierarchy and relationships effectively is to use what graphic designers call the *squint test*. Close one eye and squint at the screen with the other eye to see which elements pop out, which are fuzzy, and which seem to be grouped. Changing your perspective can often uncover previously undetected issues in layout and composition.

Provide visual structure and flow at each level of organization

It's useful to think of user interfaces as being composed of visual and behavioral elements, which are used in groups, which are then grouped into panes, which then may, in turn, be grouped into screens, views, or pages. These groupings, as discussed earlier, can be accomplished through spacing or shared visual properties. A sovereign application may have many such levels of structure. Therefore, it is critical that you maintain a clear visual structure so that the user can easily navigate from one part of your interface to another, as his workflow requires. The rest of this section describes several important attributes that help define a crisp visual structure.

Align to a grid

Aligning visual elements is one of the key ways that designers can help users experience a product in an organized, systematic way. Grouped elements should be aligned both horizontally and vertically, as shown in Figure 17-1. In general, every element on the screen should be aligned with as many other elements as possible. The decision not to align

elements or groups of elements should be made judiciously, and always to achieve a specific differentiating effect. In particular, designers should take care to do the following:

- **Align labels**—Labels for controls stacked vertically should be aligned with each other; unless labels differ widely in length, left justification is easier for users to scan than right justification.

- **Align within a set of controls**—A related group of check boxes, radio buttons, or text fields should be aligned according to a regular grid.

- **Align across control groups and panes**—Groups of controls and other screen elements should all follow the same grid wherever possible.
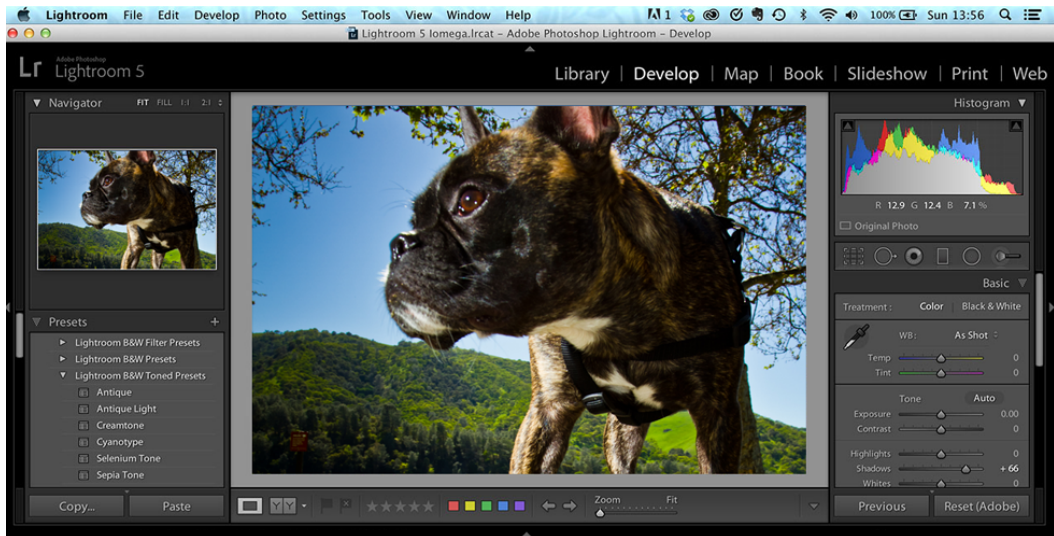
Figure 17-1: Adobe Lightroom makes very effective use of alignment to a layout grid. Text, controls, and control groups are all tightly aligned, with a consistent atomic spacing grid. It should be noted that the right alignment of controls and control group labels may compromise scanability.

A *grid system* is one of the most powerful tools available to the visual designer. Popularized by Swiss typographers in the years after World War II, a grid provides a uniform and consistent structure to layout, which is particularly important when you're designing an interface with several levels of visual or functional complexity. After interaction designers have defined the overall framework for the application and its user interface elements (as discussed in Chapter 5), visual interface designers should help regularize the layout into a grid structure. It should emphasize top-level elements and structures and provide room for lower-level or less-important controls.

Typically, the grid divides the screen into several large horizontal and vertical regions, as shown in Figure 17-2. A well-designed grid employs an *atomic grid unit* that represents the smallest spacing between elements. For example, if your atomic unit is 4 pixels, spacing between screen elements and groups will all be in multiples of 4 pixels.
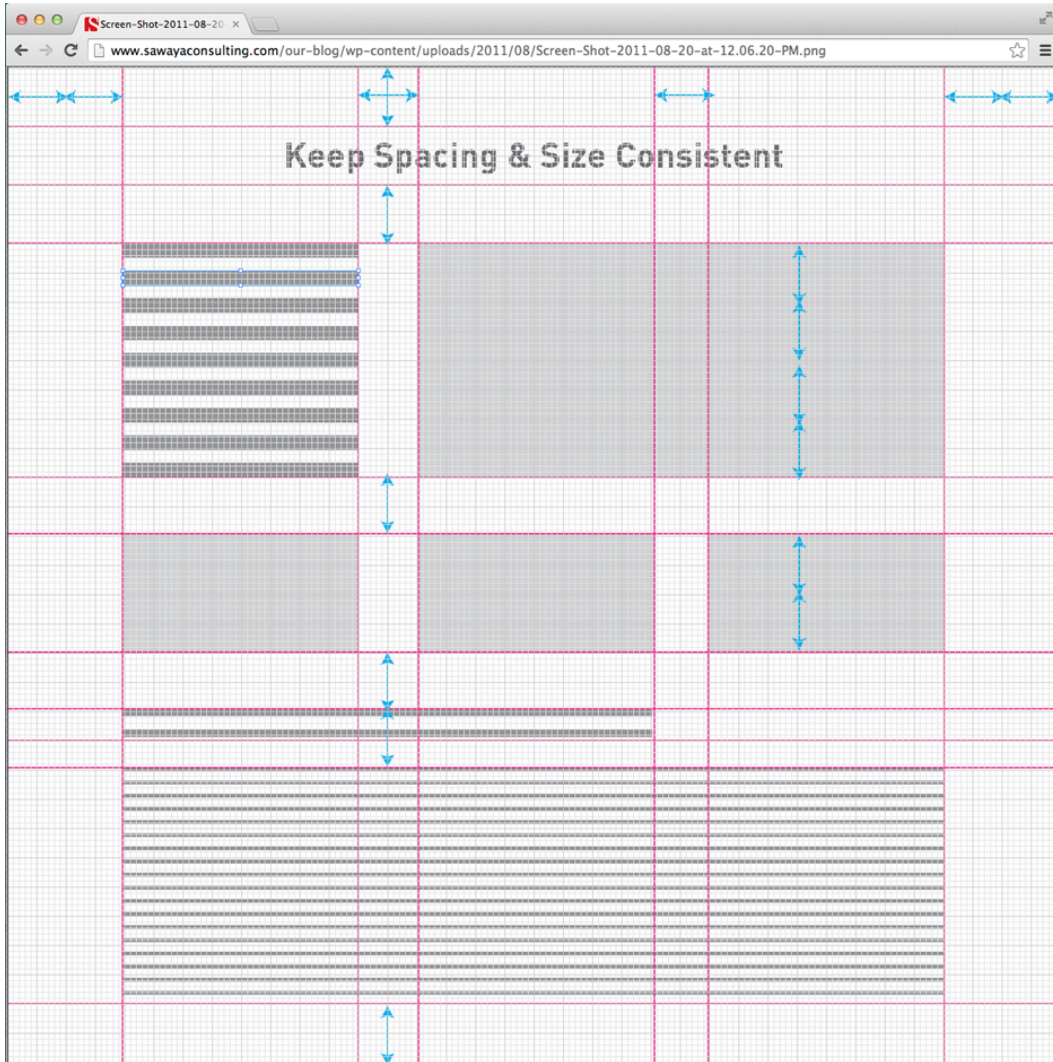
Figure 17-2: This sample layout grid prescribes the size and position of the various screen areas employed by a website. This grid ensures regularity across different screens. It also reduces the amount of work that a designer must do to lay out the screens and the work that the user must do to read and understand the screens.

Ideally, a grid should also have consistent relationships between different-sized screen areas. These relationships typically are expressed as ratios. Here are three commonly used ratios:

- The celebrated "golden section," or phi (approximately 1.61), is found frequently in nature and is thought to be particularly pleasing to the human eye.

- The square root of 2 (approximately 1:1.14) is the basis of the international paper size standard (the A4 sheet).
- 4:3 is the aspect ratio of most computer displays.

Of course, you must strike a balance between idealized geometric relationships and the specific spatial needs of the functions and information that must be presented onscreen. A perfect implementation of the golden section will do nothing to improve the readability of a screen where things are jammed together with insufficient spacing.

A good layout grid is *modular*, which means that it should be flexible enough to handle necessary variation while maintaining consistency wherever possible. And, as with most things in design, simplicity and consistency are desirable. If two areas of the screen require approximately the same amount of space, make them exactly the same size. If two areas have different requirements, make them substantially different. If the atomic grid unit is too small, the grid will become unrecognizable in its complexity. Slight differences can feel unstable to users (although they are unlikely to know the source of these feelings) and ultimately fail to capitalize on the potential strength of a strong grid system.

The key is to be decisive in your layout. *Almost* a square is no good. *Almost* a double square is also no good. *Almost* a golden rectangle is no good. If your layout is close to a simple fraction of the screen, such as a half, third, or fifth, adjust the layout so that it is exactly a half, third, or fifth. Make your proportions bold, crisp, and exact.

Using a grid system in visual interface design provides several benefits:

- **Usability**—Because grids attempt to regularize positioning of elements, users can quickly learn where to find key interface elements. If a screen header is always in precisely the same location, the user doesn't have to think or scan to find it. Consistent spacing and positioning support people's innate visual-processing mechanisms. A well-designed grid greatly improves the screen's readability.

- **Aesthetic appeal**—If you carefully apply an atomic grid and choose the appropriate relationships between the various areas of the screen, your design can create a sense of order that feels comfortable to users.

- **Efficiency**—Standardizing your layouts will reduce the amount of labor required to produce high-quality visual interfaces. We find that defining and implementing a grid early in design refinement results in less iteration and "tweaking" of interface designs. A well-defined and communicated grid system results in designs that can be modified and extended, allowing developers to make appropriate layout decisions should alterations prove necessary.

Create a logical path

In addition to precisely following a grid, the layout must properly structure an efficient *logical path* for users to follow through the interface, as shown in Figure 17-3. It must take into account the fact that (for Western users who read this way) the eye moves from top to bottom and left to right.

Logical path
Eye movements match the path through the interface

No logical path
Everything is all over the place
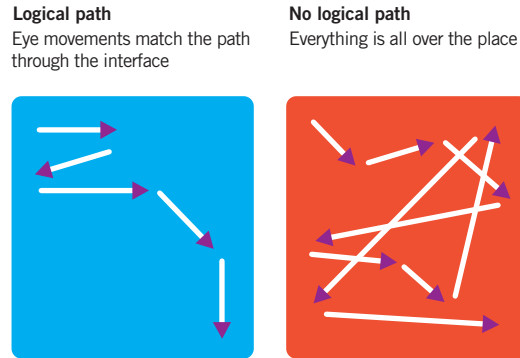
Figure 17-3: Eye movement across an interface should form a logical path that enables users to efficiently and effectively accomplish goals and tasks.

Balance the interface elements

Perfectly symmetrical interfaces lack a sense of hierarchy that encourages the user's eye to flow through the screen. Balanced asymmetry provides visual entry points to the screen and major screen areas. Experienced visual designers are adept at achieving asymmetrical balance by controlling the visual weight of individual elements much as you might balance people of different weights on a seesaw. Asymmetrical design is difficult to achieve in the context of user interfaces because of the high premium placed on white space by screen real-estate constraints. The squint test is again useful for seeing whether a display looks lopsided.

Signal what users can do on a given screen

A user encountering a screen or a function for the first time looks to the visual design to help him determine what he can do on the screen. This is the principle of affordance, discussed in Chapter 13. Affordance breaks down to design of controls and content categories with layout (of course), icons, visual symbols, and by pre-visualizing results when possible.

Use icons

In addition to their functional value, icons can play a significant role in conveying the desired brand attributes. Bold, cartoonish icons may be great if you're designing a

website for kids, whereas precise, conservatively rendered icons may be more appropriate for a productivity application. Whatever the style, it should be consistent. If some of your icons use bold black lines and rounded corners, and others use thin, angular lines, the visual style won't hold together.

Icon design and rendering is a craft in and of itself; rendering understandable images at low resolution takes considerable time and practice and is better left to trained visual designers. Icons are a complicated topic from a cognitive standpoint, so we will highlight only a few key points here. If you want to understand more about what makes usable icons, we highly recommend William Horton's *The Icon Book* (Wiley, 1994). You may find the examples dated, but the principles still hold true.

Convey a sense of the function

Designing icons to represent functions or operations performed on objects leads to interesting challenges. The most significant challenge is to represent an abstract concept in iconic, visual language. In these cases, it is best to rely on idioms rather than to force a concrete representation where none makes sense. You also should consider adding Tool-Tips (see Chapter 18) or text labels.

For more obviously concrete functions, some guidelines apply:

- Represent both the **action** and an **object** acted on to improve comprehension. Nouns and verbs are easier to comprehend together than verbs alone. (For example, a Cut command represented by a document with an X through it may be more readily understood than a more metaphorical image of a pair of scissors.)

- Beware of metaphors and representations that may not have the intended meanings for your target audience. For example, although the thumbs-up gesture means "OK" in Western cultures and might strike you as an appropriate way to communicate approval, it is offensive in Middle Eastern (and other) cultures and should be avoided in any internationalized application.

- Group related functions to provide context, either spatially or, if this is not appropriate, using color or other common visual themes.

- Keep icons simple; avoid excessive visual detail.

- Reuse elements when possible so that users need to learn them only once.

Associate visual symbols with objects

Most applications will need visual symbols to represent objects in the user's workflow. For example, in a photo management app, each image file is represented by a thumbnail. When these symbols can't be representational or metaphoric, they can idiomatic. (See

Chapter 13 for more information on the strengths of idioms.) You could represent these objects with text alone, such as with a filename, but a unique visual helps an intermediate user locate it onscreen quickly. To establish the connection between symbol and object, try to use the symbol whenever the object is represented onscreen.

Designers must also take care to visually differentiate symbols representing different object types. Discerning a particular icon within a screen full of similar icons is as difficult as discerning a particular word within a screen full of words. It's particularly important to visually differentiate objects that exhibit different behaviors, such as buttons, sliders, and check boxes.

> **DESIGN PRINCIPLE**
>
> *Visually distinguish elements that behave differently.*

Render icons and visual symbols simply

The graphics capabilities of color screens is now commonly at a very high resolution, so it is tempting to render icons and visuals with ever-increasing detail, producing an almost photographic quality. However, this trend ultimately does not serve user goals, especially in productivity applications. Icons should remain simple and schematic, minimizing the number of colors and shades and retaining a modest size.

Although fully-rendered icons may look great, they tend to fail for a number of reasons. They draw undue attention to themselves. They render poorly at small sizes, meaning that they must take up extra real estate to be legible. They encourage a lack of visual cohesion in the interface, because only a small number of functions (mostly those related to hardware) can be adequately represented with such concrete photorealistic images.

Pre-visualize results when possible

Instead of using words alone to describe the results of interface functions (or, worse, not giving any description), use visual elements to *convey* users what the results will be. Don't confuse this with using icons on control affordances. Rather, in addition to using text to communicate a setting or state, render an illustrative picture or diagram that communicates the *behavior*. Although visualization often consumes more space, its capability to clearly communicate is well worth the pixels. In recent years, Microsoft has discovered this fact, and the dialogs in Microsoft Word, for example, have begun to bristle with visualizations of their meaning in addition to the textual controls. Photoshop and other image-manipulation applications have long shown thumbnail previews of the results of visual-processing operations.

Microsoft Word's Print Preview view, shown in Figure 17-4, shows what a printed document will look like with the current paper size and margin settings. Many users have trouble visualizing what a 1.2-inch left margin looks like; the Preview control shows them. Microsoft could go one better by allowing direct input on the Preview control in addition to output, allowing users to drag the picture's left margin and watch the numeric value in the corresponding spinner ratchet up and down. The associated text field is still important—you can't just replace it with the visual one. The text shows the precise values of the settings, whereas the visual control accurately portrays the look of the resulting page.
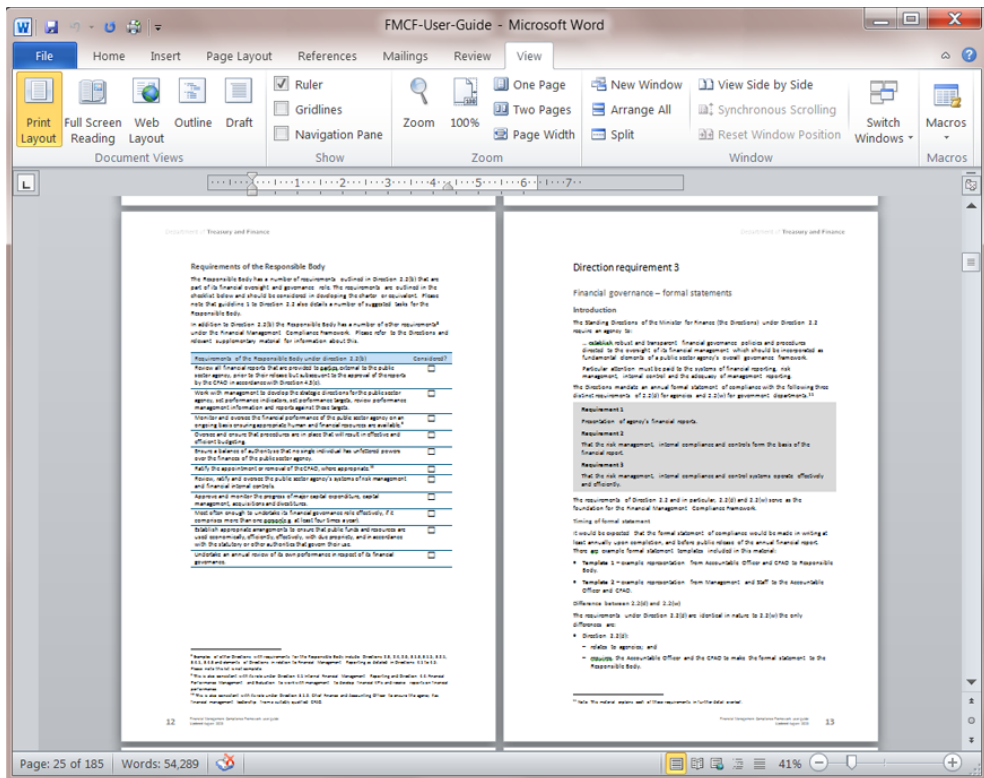
Figure 17-4: Microsoft Word Print Preview is a good example of a visual expression of application functionality. Rather than requiring users to visualize what a 1.2-inch margin might look like, this function allows the user to easily understand the ramifications of different settings.

Respond to commands

After executing a command from a swipe, tap, or click, the user needs to see some response, to know that the system has "heard" them. In some cases, the output is instant and immediate. The user selected a new typeface for some selected text, and that text changes to display in the new one. This response does not need extra visual design beyond that of the tools itself.

If the response takes longer than a tenth of a second but less than a second, you will need to provide one subtle visual cue that the command was received, and another when the activity is complete.

If the response takes longer than that up to ten seconds, you'll need to let the user know about the small delay and provide some visual cue that the process is running, most commonly with a looping animation of some sort along with an estimate of the time it will take. A common example is the single-pixel progress bars at the top of web pages expected to load quickly.

If a process will take longer than ten seconds, its usually best to design an alert explaining the delay, another for a running status update that lets them know the process is continuing in the background, followed by a respectful cue when the process is complete so they can return to the task.

Draw attention to important events

Older software was conceived as a tool, with users needing to look around to find important system events. But better, more goal-oriented software offers that information to users proactively. Badges are an example on many smartphones that embody this principle. At a glance, the user is aware that he has two games where his opponents have completed their moves, a few text messages, and some social media mentions to check on.

The tools to draw attention involve the fundamentals of human perception and are all based on contrast: Contrast of size, color, motion, etc. Make the thing you want to get attention different, and it will command attention. This sounds simple, but there are two challenges.

The first is that the attention-getting mechanisms are not under our conscious control. That makes sense when you consider that they evolved to alert us to sudden changes in the environment. So present them with a large contrast on screen, and you draw them away from their current task. This can be perceived as rude if it's misapplied. (See Chapter 8 for more about this principle.) The deservedly vilified blink tag from early days of the web is a prime example. Blinking objects command our attention so strongly that it's difficult to pay attention to anything else.

The second challenge is that it can be difficult to keep the attention signal effective but in line with the experience keywords. If your app is meant to be serene, yes, a klaxon will get the users attention, but will also break the promise that the app has made.

Minimize the amount of visual work

Visual noise within an interface is caused by superfluous visual elements that detract from the primary objectives of communicating affordances and information. The same is true for user interfaces. Visual noise can take many forms:

- Ornate embellishment
- 3D rendering that doesn't add information
- Rule boxes and other visually "heavy" elements to separate controls
- Crowding of elements
- Intense colors, textures, and contrast
- Using too many colors
- Weak visual hierarchy

Cluttered interfaces attempt to provide an excess of functionality in a constrained space, resulting in controls that visually interfere with each other. Visually baroque, disorderly, or crowded screens increase the user's cognitive load.

Keep it simple

In general, visual interfaces should strive to be minimal, such as simple geometric forms or a restricted color palette composed primarily of less-saturated or neutral colors balanced with a few high-contrast accent colors that emphasize important information. Typography should not vary widely in an interface: Typically one or two typefaces, specified to just a few sizes, is sufficient for most applications.

Unnecessary variation is the enemy of a coherent, usable design. If the spacing between two sets of elements is nearly the same, make that spacing exactly the same. If two typefaces are nearly the same size, adjust them to be the same size. Every visual element and every difference in color, size, or other visual property should be there for a reason. If you can't articulate a good reason why it's there, get rid of it.

Good visual interfaces, like any good visual design, are visually *efficient*. They make the best use of the minimal set of visual and functional elements. A popular technique used

by both graphic designers and industrial designers is to experiment with removing individual elements to test their contribution to the clarity of the intended message.

DESIGN PRINCIPLE

> *Take things away until the design breaks, and then put that last thing back in.*

As pilot and poet Antoine de Saint-Exupéry famously said, "Perfection is attained not when there is no longer anything to add, but when there is no longer anything to take away." As you create your interfaces, you should constantly be looking to simplify, visually. The more useful work a visual element can accomplish while still retaining clarity, the better.

Related to the drive for simplicity is the concept of *leverage*, which is where a single interface element is used for several related purposes. For example, in Microsoft Windows 8, an icon appears next to the window's title, as shown in Figure 17-5. This icon visually communicates the window's contents (for example, whether it is an Explorer window or a Word document) and provides access to window configuration commands such as Minimize, Maximize, and Close.
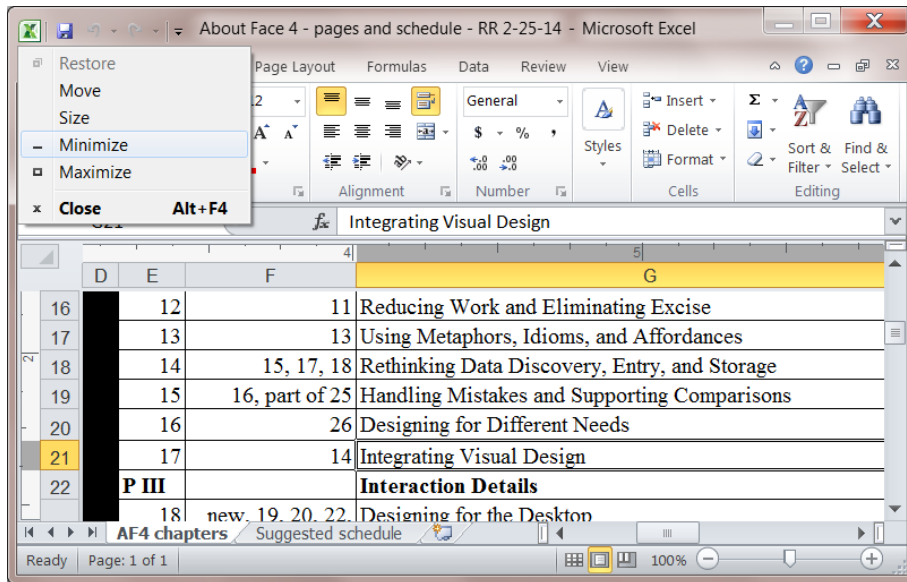
Figure 17-5: The icon in the title bar of windows in Windows 8 is a good example of leverage. It communicates the window's contents and provides access to window configuration commands.

424   Part II: Making Well-Behaved Products

Visual Information Design Principles

Like visual interface design, visual information design has many principles that the designer can use to her advantage. Information design guru Edward Tufte asserts that good visual design is "clear thinking made visible" and that good visual design is achieved through an understanding of the viewer's "cognitive task" and a set of design principles.

Tufte claims that information design has two important problems:

- It is difficult to display multidimensional information (information with more than two variables) on a two-dimensional surface.

- The resolution of the display surface often is not high enough to display dense information. Computers present a particular challenge. Although they can add motion and interactivity, computer displays have a lower information density compared to paper. (The retina displays sold by Apple with some of their products have a much higher pixel density, but are not standard, and risky to presume for all users.)

Although both points are certainly true, the visual interface designer can leverage one capability not available to the print information designer: interactivity. Unlike a print display, which must convey all the data at once, electronic displays can progressively reveal information as users need more detail. This helps make up, at least in part, for the resolution limitations.

Even with the differences between print and digital media, some universal information design principles—indifference to language, culture, and time—help maximize the effectiveness of any information display.

In his beautifully executed volume, *The Visual Display of Quantitative Information* (Graphics Press, 2001), Tufte introduces seven "Grand Principles," which we briefly discuss in the following sections as they relate specifically to digital interfaces and content.

According to Tufte, visually displayed information should do the following:

- Enforce visual comparisons
- Show causality
- Show multiple variables
- Integrate text, graphics, and data in one display
- Ensure the content's quality, relevance, and integrity
- Show things adjacent in space, not stacked in time
- Don't dequantify quantifiable data

We will briefly discuss each of these principles as they apply to the information design of software-enabled media.

Enforce visual comparisons

You should provide a means for users to compare related variables and trends or to compare before-and-after scenarios. Comparison provides a context that makes the information more valuable and comprehensible to users (see Figure 17-6). Adobe Photoshop, along with many other graphics tools, makes frequent use of previews, which allow users to easily do before-and-after comparisons interactively.

Figure 17-6: This graph from Google finance compares the performance of two stocks with the S&P 500 over a period of time. The visual patterns allow a viewer to see that Barclays Bank (BCS) and UBS are closely correlated to each other and only loosely correlate to the S&P 500.

Show causality

Within information graphics, clarify cause and effect. In his books, Tufte relates the classic example of the space shuttle Challenger disaster. Tufte believes the tragedy could have been avoided if charts prepared by NASA scientists had been organized to more clearly present the relationship between air temperature at launch and severity of O-ring failure. In interactive interfaces, you should employ rich visual modeless feedback (see Chapter 15) to inform users of the potential consequences of their actions or to provide hints on how to perform actions.

Show multiple variables

Data displays that provide information on multiple related variables should be able to display them all simultaneously without sacrificing clarity. In an interactive display, the user should be able to selectively turn off and on the variables to make comparisons

easier and correlations (causality) clearer. Investors are often interested in the correlations between different securities, indexes, and indicators. Graphing multiple variables over time helps uncover these correlations, as shown in Figure 17-6.

Integrate text, graphics, and data in one display

Diagrams that require separate keys or legends to decode require additional cognitive processing by users and are less effective than diagrams with integrated legends and labels. Reading and deciphering diagram legends is yet another form of navigation-related excise. Users must move their focus between diagram and legend and then reconcile the two in their minds. Figure 17-7 shows an interactive example that integrates text, graphics, and data, as well as input and output—a highly efficient combination for users.
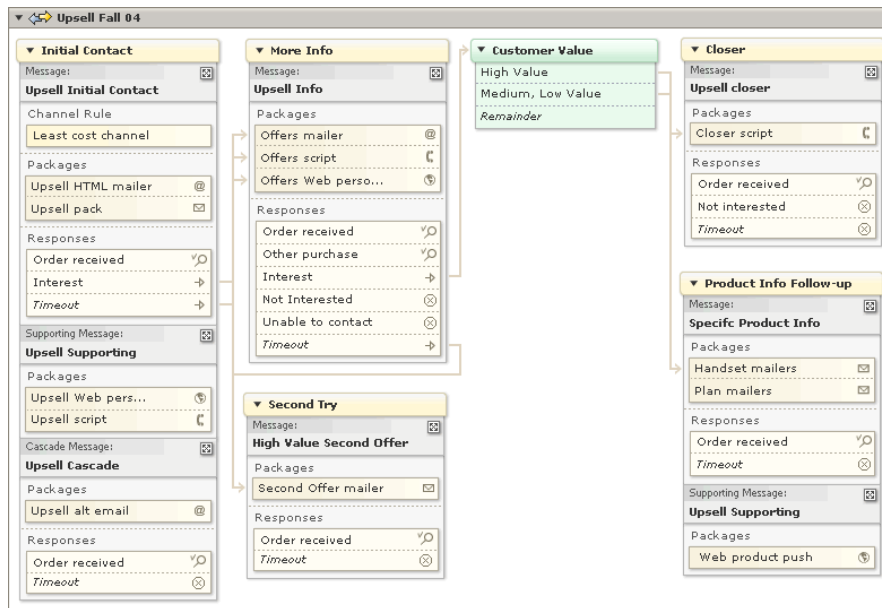
Figure 17-7: This "Communication Plan" is an interface element from a tool for managing outbound marketing campaigns that was designed by Cooper. It gives textual information a visual structure, which in turn is augmented by iconic representations of different object types. Not only does this tool provide output of the current structure of the Communication Plan, but it also allows the user to modify that structure directly through drag-and-drop interactions.

Ensure the content's quality, relevance, and integrity

Don't show information simply because it's technically possible to do so. Make sure that any information you display will help your users achieve particular goals that are relevant to their context. Unreliable or otherwise poor-quality information will damage the trust you must build with users through your product's content, behavior, and visual brand.

Show things adjacent in space, not stacked in time

If you are showing changes over time, it's much easier for users to understand the changes if they are shown adjacent in space, rather than superimposed. When information is stacked in time, you are relying on their short term memory to make the comparison, which is not as reliable or fast as a side-by-side comparison. Cartoon strips are a good example of showing flow and change over time arranged adjacent in space.

Of course, this advice applies to static information displays. In software, *animation* can be used even more effectively to show change over time, as long as technical issues (such as memory constraints and connection speed over the Internet) don't come into play.

Don't dequantify quantifiable data

Although you may want to use graphs and charts to make trends and other quantitative information easy to grasp, you should not abandon the display of the numbers themselves. For example, in the Windows Disk Properties dialog, a pie chart is displayed to give users a rough idea of their free disk space, but the numbers of gigabytes free and used are also displayed in numeric form.

Consistency and Standards

Many in-house usability organizations view themselves as, among other things, the gate-keepers of consistency in digital product design. *Consistency* implies a similar look, feel, and behavior across the various modules of a software product, and this is sometimes extended to apply across all the products a vendor sells. For large software vendors, such as Adobe and Google, which regularly acquire new software titles from smaller vendors, the branding concerns of consistency take on particular urgency. It is obviously in their best interests to make acquired software look as though it belongs, as a first-class offering, alongside products developed in-house. Beyond this, both Apple and Microsoft have an interest in encouraging their own and third-party developers to create applications that have the look and feel of the OS platform on which the application is being run. This way, the user perceives their respective platforms as providing a seamless and comfortable user experience.

Benefits of interface standards

User interface standards provide benefits that address these issues when executed appropriately, although they come at a price. According to Jakob Nielsen, relying on a single interface standard improves users' ability to quickly learn interfaces and enhances their productivity by raising throughput and reducing errors. These benefits accrue because

users can more easily predict application behavior based on past experience with other parts of the interface or with other applications following similar standards.

At the same time, interface standards also benefit software vendors. Customer training and technical support costs are reduced because the consistency that standards bring improves ease of use and learning. Development time and effort are also reduced because formal interface standards provide ready-made decisions on the rendering of the interface that development teams would otherwise be forced to debate during project meetings. Finally, good standards can lead to reduced maintenance costs and improved reuse of design and code.

Risks of interface standards

The primary risk of any standard is that the product that follows it is only as good as the standard itself. Great care must be taken in developing the standard to ensure, as Nielsen says, that it specifies a truly usable interface and that it can be used by the *developers* who must build the interface according to the standard's specifications.

It is also risky to see interface standards as a panacea for good interfaces. Assuming that a standard is the solution to interface design problems is like saying the *Chicago Manual of Style* is all it takes to write a good novel. Most interface standards emphasize the interface's *syntax*—its look and feel—but say little about the interface's deeper behaviors or its higher-level logical and organizational structure. There is a good reason for this: A general interface standard has no knowledge of context incorporated into its formalizations. It takes into account no specific user behaviors and usage patterns within a context. Instead, it focuses on general issues of human perception and cognition and, sometimes, visual branding as well. These concerns are important, but they are presentation details, not the interaction framework on which such rules hang.

Standards, guidelines, and rules of thumb

Although standards are unarguably useful, they need to evolve as technology and our understanding of users and their goals evolve. Some practitioners and developers invoke Apple's or Microsoft's user interface standards as if they were delivered from Mt. Sinai on a tablet. Both companies publish user interface standards, but both companies also freely and frequently violate them and update the guidelines after the fact. When Microsoft proposes an interface standard, it has no qualms about changing it for something better in the next version. This is only natural. Interface design is still coming into maturity, and it is wrong to think that there is benefit in standards that stifle innovation.

The original Macintosh was a spectacular achievement precisely because it transcended all of Apple's previous platforms and standards. Conversely, much of the strength of the

Mac came from the fact that vendors followed Apple's lead and made their interfaces look, work, and act alike. Similarly, many successful Windows applications are unabashedly modeled after Word, Excel, and Outlook.

Interface standards thus are most appropriately treated as detailed guidelines or rules of thumb. Following interface guidelines too rigidly or without carefully considering user needs in context can result in force-fitting an application's interface into an inappropriate interaction model.

When to violate guidelines

So, what should we make of interface guidelines? Instead of asking if we should *follow* standards, it is more useful to ask when we should *violate* standards? The answer is when we have a very good reason.

DESIGN PRINCIPLE	*Obey standards unless there is a truly superior alternative.*

But what constitutes a very good reason? Is it when a new idiom is *measurably* better? Usually this sort of measurement can be elusive, because it can rarely be reduced to a quantifiable factor alone. The best answer is that when an idiom is clearly seen to be significantly better by most people in the target user audience (your personas) who try it, there's a good reason to keep it in the interface. This is how the toolbar came into existence, along with outline views, tabs, and many other idioms. Researchers may have examined these artifacts in the lab, but it was their useful presence in real-world software that confirmed their success.

Your reasons for diverging from guidelines ultimately may not prove to be good enough, and your product may suffer, but you and other designers will learn from the mistake. This is what Berkeley professor Christopher Alexander calls the "unselfconscious process"—an indigenous and unexamined process of slow and tiny forward increments as individuals attempt to improve solutions. New idioms (as well as new uses for old idioms) pose a risk. This is why careful, goal-directed design and appropriate testing with real users in real working conditions are so important.

Consistency and standards across applications

Using standards or guidelines can involve special challenges when a company that sells multiple software titles decides that all its various products must be completely consistent from a user-interface perspective.

From the perspective of visual branding, as discussed earlier, this makes a great deal of sense, although there are some intricacies. Suppose an analysis of personas and markets indicates that there is little overlap between the users of two distinct products and that their goals and needs also are distinct. You might question whether it makes more sense to develop two visual brands that speak specifically to these different customers, rather than using a single, less-targeted look. When it comes to the software's behavior, these issues become even more urgent. A single standard *might* be important if customers will be using the products together as a suite. But even in this case, should a graphics-oriented presentation application like PowerPoint share an interface structure with a text processor like Word? Microsoft's intentions were good, but it went a little too far in enforcing global style guides. PowerPoint doesn't gain much from having a menu structure similar to that of Excel and Word. It also loses quite a bit in ease of use by conforming to an alien structure that diverges from the user's mental models. On the other hand, the designers did draw the line somewhere. PowerPoint has a slide-sorter display—an interface unique to that application.

Designers, then, should bear in mind that consistency doesn't imply rigidity, especially where it isn't appropriate. Interface and interaction style guidelines need to grow and evolve like the software they help describe. Sometimes you must bend the rules to best serve your users and their goals (and sometimes even your company's goals). When this has to happen, try to make changes and additions that are compatible with standards. The spirit of the law, not the letter of the law, should be your guide.

| DESIGN PRINCIPLE | *Consistency doesn't imply rigidity.* |

The design language

One of the visual interface designer's most important tools is the idea of a "design language." Think of this design language as a "vocabulary" of design elements such as shape, color, typography, and how these elements are composed and combined. They create the appropriate emotional tone and establish patterns that a person can recognize, understand, and, ideally, use to create positive associations with the brand of the product or service being created.

A good example is Microsoft's Metro design language, the foundation of Windows 8, Windows Phone, and Xbox user interfaces. By using a common set of visual elements such as content tiles, Microsoft has created a variety of interfaces and experiences that are clearly recognizable (see Figure 17-8).
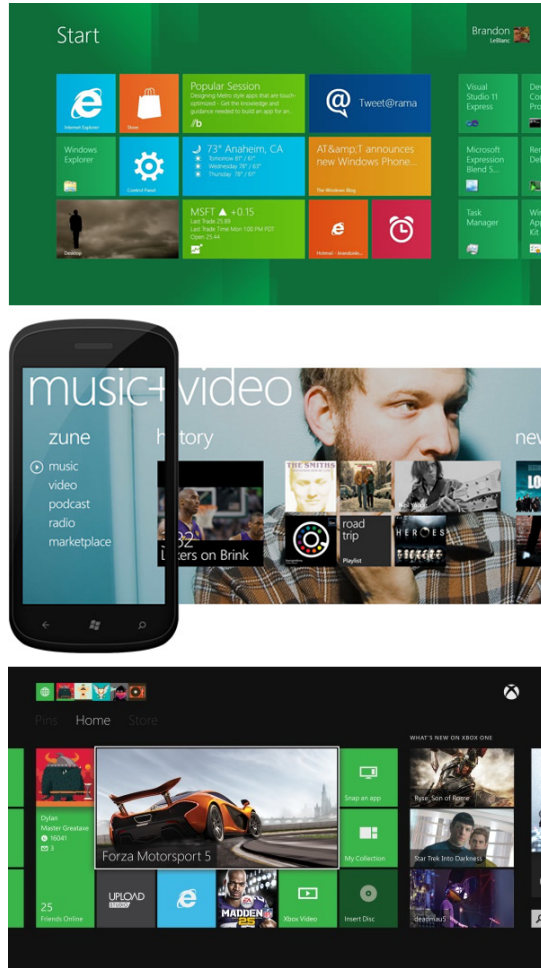
Figure 17-8: Cross-platform examples of Microsoft's Metro design language.

In some cases, this language emerges as a vernacular. But in our experience, it is best arrived at through an explicit process that evaluates a variety of potential visual and interaction languages in terms of brand appropriateness and fitness for purpose. The best design languages evolve through the product design process in a user-centric way. Every design decision is rationalized against other decisions, and variation is reduced to just what is required to create meaning, utility, and the right emotional tone for users.

Design languages are often communicated through standards and style guides, but it's important to note that the two are not synonymous. Just because you have a style guide doesn't mean you have a well-developed design language, and vice versa. It is possible to have a useful design language without having a style guide or standards manual. (However, it should be said that compiling a style guide can help designers rationalize and simplify a design language.)

PART

III
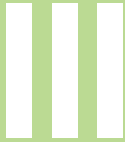
INTERACTION DETAILS

DESIGNING FOR THE DESKTOP

Most modern desktop interfaces derive their appearance from the Xerox Alto, an experimental computer system developed in 1973 at Xerox's Palo Alto Research Center (PARC), now PARC, Inc. PARC's Alto was the first computer with a graphical interface and was designed to explore the potential of computers as desktop business systems. In creating the Alto, PARC researchers developed what became the four pillars of the desktop UI paradigm: windows, icons, menus (and other widgets), and pointer, or WIMP for short.

The Alto was designed as a networked office system in which documents could be composed, edited, and viewed in WYSIWYG (what you see is what you get) form; stored; retrieved; transferred electronically between workstations; and printed. The Alto system, and its commercially unsuccessful progeny, the Xerox Star, contributed many significant innovations to the vernacular of desktop computing that we now regard as commonplace: the mouse, the rectangular window, the scrollbar, the (virtual) pushbutton, the "desktop metaphor," object-oriented programming, multitasking, Ethernet, and laser printing.

PARC's effect on the industry and contemporary computing was profound. Both Steve Jobs and Bill Gates saw the Alto at PARC in the late 1970s and were indelibly impressed.

After hiring away many of PARC's most brilliant minds—who jumped ship after it was clear that Xerox was about to fumble the entire future of computing—Steve Jobs set about reincarnating the Alto/Star in the form of the Lisa. The Lisa was remarkable, accessible, exciting, far too expensive ($9,995 in 1983), and frustratingly slow. It also introduced new graphical idioms, such as drop-down menus, to the new visual language of computing.

About this time, less visually polished desktop interfaces also began to appear on expensive and more powerful UNIX workstations from companies like Sun Microsystems, which fared somewhat better with hardcore scientific and engineering audiences. Not deterred by the commercial failure of the Lisa, Jobs began a secret project to develop an affordable incarnation of the Alto.

The result was the legendary Macintosh, a machine that has had enormous influence on our technology, design, and culture. The Mac single-handedly brought an awareness of design and aesthetics to the industry. It not only raised the standards for user-friendliness, but it also enfranchised a whole population of skilled individuals from disparate fields who were previously locked out of computing because of the industry's self-absorption in techno-trivia. Microsoft, after creating some of the first software for the Mac, went on to develop its own WIMP interface for PCs—Windows—between them defining our personal computing paradigms for over two decades.

This chapter covers detailed design considerations for modern desktop GUIs of all flavors. It focuses on the behaviors of windows and their structural and navigational components, as well as pointer-driven selection and manipulation of onscreen objects.

Anatomy of a Desktop App

As you may remember from our earlier discussion of software posture (see Chapter 9), the two primary types of desktop interfaces are sovereign and transient. By far, the majority of actual work that gets done on desktop applications is done in sovereign applications. Transients exist in supporting roles for brief, intermittent, or largely background tasks (such as music playback or instant messaging). Consequently, this section focuses on the basic structural patterns of sovereign desktop apps, the building blocks of which we'll discuss later in this chapter, as well as in Chapter 21.

Primary and secondary windows

The top-level structure of desktop applications (as opposed to the operating system itself) is the window—the movable, resizable container within which both content and functional controls for the app primarily reside. In terms of structuring your application, you can think of it as having a primary window and, in many cases, one or more secondary windows.

Primary window

The *primary window* contains your application's content, typically expressed in the form of documents that can be created, edited, and shared. Less frequently, it contains other

sorts of objects with properties that can be manipulated and configured, or media that can be viewed or played. Primary windows often are divided into *panes* that contain content, a means of navigating between different content objects, and sets of frequently used functions for manipulating or controlling the content. Primary windows typically are designed to assume sovereign posture, filling most of the screen and supporting full-screen modes.

Secondary windows

Secondary windows support the primary window, providing access to less frequently used properties and functions, typically in the form of *dialogs*. We'll discuss dialogs and their structure at length in Chapter 21. If your application allows panes located in the primary window to be detached and manipulated separately, these floating panels or palettes also take on a role as secondary windows.

Primary window structure

Primary windows, as mentioned, frequently are subdivided into multiple functional areas:

- A content or work area
- A menu bar
- Multiple toolbars, panels, or palettes that help you navigate to or select content objects or operate on selected content objects within the work area

Menus and toolbars

Menus and toolbars are collections of related actions the user can instruct the application to perform, such as "close this document" or "invert the colors of the current selection." Menus are accessed by clicking on words arranged near the top of the screen, and are often subject to standardization rules from the operating system. Toolbars are more specific to the application, often summoned or dismissed through menus, and—once active—present their functions as a collection of visual icons, often with small labels.

When function menus are included within the primary window, they appear across the top of the window within a menu bar. Traditional toolbars appear directly below the menu bar (or, in OS X, across the top of the window). Newer UI idioms such as Microsoft's ribbon seek to take the place of both menus and toolbars, replacing them with a tabbed toolbar-like construct. It is more verbose than a toolbar and therefore shares some of the pedagogic features of menus. We discuss menus, toolbars, and related UI idioms in detail later in this chapter.

Content panes

Content panes form the primary work area within most desktop applications, whether it is the editable view of a form or document or (as in the case of a software music synthesizer, for example) a complex control panel. An application typically has one primary content area. But applications that support editing multiple documents or views of a document (such as in CAD software) side-by-side may have multiple content panes. These panes may affect each other or allow dragging and dropping of objects between them.

Index panes

Index panes provide navigation and access to documents or objects that ultimately appear in the content view(s) for editing or configuration. Sometimes selecting an object in an index view displays it in the content area (such as in an e-mail app, where selecting an e-mail in the index pane displays its contents in the content pane). In other cases, objects dragged from an index pane to a content pane are added to the pane, leaving its existing contents intact. This behavior is typical in authoring tools, where index panes are often used to house asset libraries or metadata.

Tool palettes

Although they have many visual similarities to toolbars, tool palettes serve a unique purpose. They allow the user to rapidly switch between the application's modes of operation by selecting one tool from a set of tools. Each tool assigns a different set of operations to actions, such as clicking or dragging. This mode change typically is hinted at by a change in the cursor's visual design to match the semantics of the currently selected mode or tool.

Tool palettes typically are vertically oriented and are typically positioned (at least by default) on the left edge of the primary window. We discuss tool palettes in detail later in this chapter.

Sidebars

Sidebars are a relatively recent but popular interaction idiom. They most often allow object or document properties to be manipulated without the need to resort to modal or modeless dialogs. This streamlines the workflow in complex authoring applications. Typically sidebars are positioned on either the right or left side of the primary window. But they can be in both places, and even positioned along the bottom of the window, or in place of a toolbar. We discuss sidebars in detail later in this chapter.

Windows on the Desktop

A WIMP fundamental that emerged from PARC is the idea of rectangular windows containing application controls and documents. The rectangular theme of modern GUIs is so dominating and omnipresent that it is often seen as vital to the success of visual interaction.

There are good reasons to display data in rectangular panes. Probably the least important is that rectangular panes are a good match for our display technology: CRTs and LCDs have an easier time with rectangles than with other shapes. More important is the fact that most data output used by humans is in a rectangular format: We have viewed text on rectangular sheets since Gutenberg, and most other forms, such as photographs, film, and video also conform to a rectangular grid. Rectangular graphs and diagrams are also the easiest for us to make sense of. Rectangles are also quite space-efficient. So they just seem to work, cognitively and efficiently.

Overlapping windows

Application windows on the PARC systems, as well as on the Lisa and the Mac, were rendered as overlapping shapes on the metaphorical desktop. They could be dragged over one another (obscuring windows underneath), stacked, and independently resized.

Overlapping windows demonstrated clearly that there are better ways to transfer control between concurrently running applications other than typing obscure commands. The visual metaphor of overlapping shapes seems to work well initially. Your physical desk, if it is like ours, is covered with papers. When you want to read or edit one, you pull it out of the pile, lay it on top, and get to work. The virtual desktop mimics this real-world interaction reasonably well.

The problem is that, just like in the real world, this metaphor doesn't scale, especially if your desk is covered with papers and, like your laptop's screen, is only 15 inches across diagonally. The overlapping window *concept* is good, but its execution is somewhat impractical without the addition of other idioms to aid navigation between multiple applications and documents.

Overlapping windows create other problems, too. A user who mis-clicks the mouse a few pixels in the wrong direction can find that his application has apparently disappeared, replaced by another one that was lurking beneath it. User testing at Microsoft showed that a typical user might launch the same word processor several times in the mistaken belief that he had somehow "lost" the application and needed to start over. Problems like these prompted Microsoft to introduce the taskbar. In OS X, Apple decided to address this problem with Expose. Even though it provides an innovative idiom for keeping track

of open windows, Expose suffers from a curious lack of integration with applications minimized to Apple's taskbar-like Dock.

A final point of confusion regarding overlapping windows is the multitude of other desktop idioms that are also represented by overlapping shapes. The familiar dialog box is one, as are menus and floating tool palettes. Such overlapping within a single application is natural and a well-formed idiom. It even has a faint trace of metaphor: that of someone handing you an important note. The problem arises when you scale up to using many open applications. The sheer number of overlapping layers can lead to visual noise and clutter, as well as obscuring which layers belong to which application.

Tiled windows

After the commercial success of the Macintosh, Bill Gates and his engineering team at Microsoft created a response to the Apple/Xerox GUI paradigm.

The first version of Windows diverged somewhat from the pattern established by Xerox and Apple. Instead of using overlapping rectangular windows to represent the overlapping sheets of paper on one's desktop, Windows 1.0 relied on what was called *tiling* to allow users to have more than one application onscreen at a time (Xerox PARC's CEDAR, however, predated Windows as the first tiling window manager).

Tiling meant that applications divided the screen into uniform rectangular *tiles*, evenly parceling out the available space to each running application. Tiling was invented as an idealistic means to solve the orientation and navigation problems caused by overlapping windows. Navigation between tiled windows is much easier than between overlapped windows, but the cost in pixel real estate for each tiled app is horrendous.

Windows 1.0, however, didn't rigidly enforce tiling like CEDAR did, so as soon as the user moved any of the neatly tiled windows, he was thrust right back into the excise of window manipulation. Tiling window managers failed as a mainstream idiom, although remnants can still be found. Try right-clicking the current Windows taskbar, and choose "Show windows side by side." The new Start screen on Windows 8, with its mosaic of dynamically updating app content tiles, harks back to the tiled windows concept as well, but in a more appropriate and useful incarnation.

Virtual desktop spaces

Overlapping windows don't make it easy to navigate between multiple running applications, so vendors continue to search for new ways to achieve this. The *virtual desktop* or *session* managers on some platforms extend the desktop to many times the size of the physical display by, in essence, adding a set of virtual screens. (Apple calls this feature *Spaces* in OS X.) The virtual desktop UI typically shows thumbnail images of all the

desktop spaces, each of which can display different sets of apps and open windows, the state of which can be preserved across login sessions. You switch between these virtual desktops by clicking the one you want to make active (or pressing a key command to move between them). In some versions, you can even drag tiny window thumbnails from one desktop to another. This kind of metamanagement of apps and windows can be useful to power users who keep many apps they work with open simultaneously.

Full-screen applications

While virtual desktops are a reasonably elegant solution to a complex problem for power users, the basic problem of working with windows seemed to have been lost in the shuffle: How can a more typical user easily navigate between applications?

Multiple windows sharing a small screen—whether overlapping or tiled—is not a good general solution (although it has important occasional uses). However, with recent OS releases from both Apple and Microsoft, we are moving toward a world of full-screen applications. Each application occupies the entire screen when it is "up at bat." Tools like the taskbar borrow the minimum quantity of pixels from the running application to provide a visual method of changing the lineup. (This concept is actually quite similar to the early days of the Mac with its Switcher facility, which transitioned the Mac display between one full-screen application and another.) This solution is much more pixel-friendly, less confusing to users, and highly appropriate when an application is being used for an extended period of time. In OS X and Windows 8, users can make their applications full-screen or overlapping. With growing influence from tablet (and even phone) experiences, the bias is increasingly toward full-screen experiences.

Multipaned applications

It turns out that a powerful idiom takes the best elements of tiled windows and provides them within a sovereign, full-screen application—the idiom of *multipaned windows*. Multipaned windows consist of independent views or *panes* that share a single window. Adjacent panes are separated by fixed or movable dividers or *splitters*. The classic example of a multipaned application is Microsoft Outlook. Separate panes are used to display the list of mailboxes, contents of the selected mailbox, a selected message, and upcoming appointments and tasks, all on one screen (see Figure 18-1).

The advantage of multipaned applications is that independent but related information can be easily displayed in a single, sovereign posture screen in a manner that reduces both navigation and window management excise to almost nil. For a sovereign application of any complexity, multipane design is practically a requirement. Specifically, designs that provide navigation and/or building blocks in one pane and allow viewing or construction of data in an adjacent pane are an effective pattern that bears consideration.
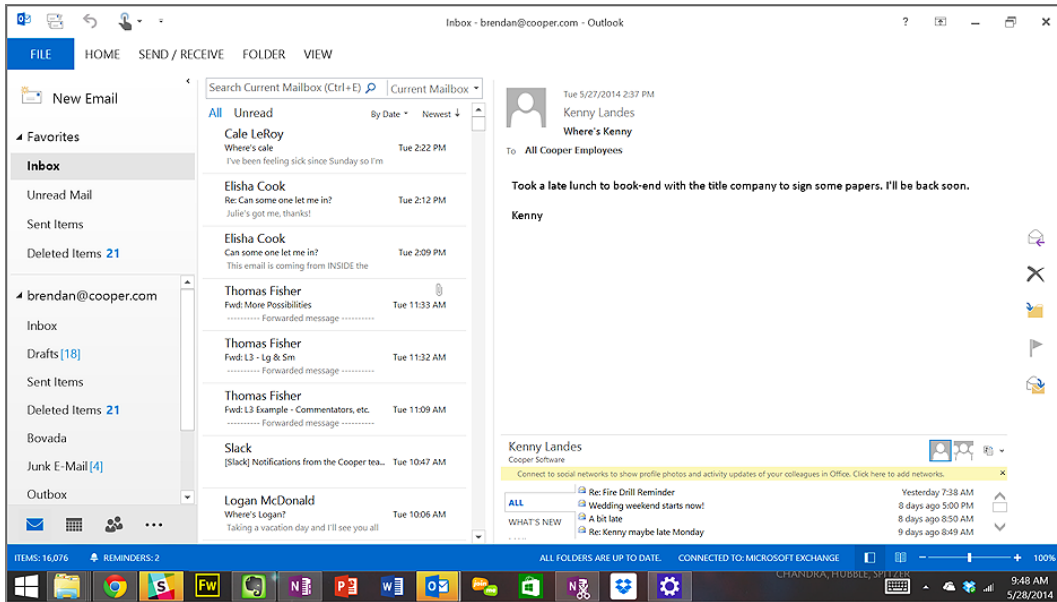
Figure 18-1: Microsoft Outlook is a classic example of a multipaned application. The far-left pane contains a list of mailboxes. It also lets you switch between views such as Mail and Calendar. The top-center pane shows all the messages in the selected mailbox, and the pane below it shows the contents of the selected message. The pane on the right shows the next three appointments and upcoming tasks.

Another form of multiple panes is *stacked panes* or *tabs*. These are common to preferences, settings, and other complex dialogs; they are also sometimes useful in sovereign windows. Most modern web browsers let users have many sites open at a time, accessible through tabs at the top. Another good example is Microsoft Excel, which allows related spreadsheets to be accessible via inverted tabs at the bottom of the screen. Excel makes use of stacked panes with its Sheets.

Window states

With the ability to expand to full screen, an application's primary window can be in one of three states: minimized, maximized, or *pluralized*.

Minimized windows get collapsed into icons on the desktop (on older OSs) or into the taskbar (Windows) or the Dock (OS X). *Maximized* windows fill the entire screen, covering whatever is beneath them.

Microsoft and Apple both somehow manage to avoid referring directly to the third state. The only hint of a name is on the Microsoft system menu (click the application icon in the upper-left corner of the window to see it) where the Restore command describes how to

get to it. This function switches a maximized primary window to that *other* state—the *pluralized* or *restored* state.

The pluralized state is that in-between condition where the window is neither an icon nor maximized to cover the entire screen. When a window is pluralized, it shares the screen with icons and other pluralized windows. Pluralized windows can be either tiled or overlapping (but are usually the latter).

The normal state for a sovereign application is maximized. There is little reason for such an application to be pluralized, other than to support switching between applications or dragging and dropping data between applications or documents Transient-posture applications, such as Windows Explorer, the Finder, the calculator, or the many IM and other social media applications in popular use today, are appropriately displayed in a pluralized window.

Windows and documents: MDI vs SDI

If you want to copy a cell from one spreadsheet and paste it to another, opening and closing both spreadsheets in turn is tedious. It's much easier to have two spreadsheets open simultaneously. There are two ways to accomplish this. You can have one spreadsheet application that can contain two or more spreadsheet instances. Or you can have multiple instances of the entire spreadsheet application, each one containing a single instance of a spreadsheet.

In the early days of Windows, Microsoft chose the first option for the simple, practical reason of resource frugality, and called it the *multiple document interface*, or MDI. One application with multiple spreadsheets (documents) conserved more bytes and CPU cycles than multiple instances of the same application, and performance was a serious issue then. Eventually, as technology improved, Microsoft abandoned MDI and embraced the other approach, the *single document interface*, or SDI.

MDI is actually reasonable enough in certain contexts. In particular, it is useful when users need to work on multiple related views of information in a single environment (for example, a set of related screen mockups in Photoshop).

> **DESIGN PRINCIPLE**   *The utility of any interaction idiom is context-dependent.*

SDI generally works well, and seems simpler for users, but it's not a global panacea. While it's reasonably convenient to switch between instances of Word to go back and forth between different documents, you wouldn't want a purchasing agent to have to

switch between multiple instances of his massive enterprise planning system to look at an invoice and the vendor's billing history.

MDI, due to its window-in-a-window nature, can be abused. Navigation becomes oppressive if document views require full window management within the MDI container (as some early versions of MDI did). Everything described in our earlier discussion of excise introduced by minimizing, maximizing, and pluralizing windows goes double for document windows inside an MDI application—a truly heinous example of window management excise. In most cases it is far superior to transition cleanly from one fully open document to the next—or allow tiling or tabbing of open documents, as Photoshop does.

Making use of windows

As described above, desktop applications are constructed of two kinds of windows: primary and secondary (dialog boxes). Determining how to use windows in an application is an important aspect of defining the application's Design Framework (see Chapter 5).

Unnecessary rooms

If we imagine our application as a house, each application window is a separate room. The house itself is represented by the application's primary window, and each room is a pane, document window, or dialog box. We don't add a room to our house unless it has a purpose that cannot be served by other rooms. Similarly, we shouldn't add windows to our application unless they have a purpose that can't or shouldn't be served by existing windows.

It's important to think through this question of purpose by considering prospective users' goals and mental models. The way we think about it, saying that a room has a purpose implies that using it is associated with a goal, but not necessarily with a particular task or function.

> **DESIGN PRINCIPLE**
>
> *A dialog box is another room; have a good reason to go there.*

Even today, a preponderance of secondary windows is a problem that haunts desktop software. Developers often work by breaking the application into discrete functions, and the user interface is then constructed in close parallel. Combine this with the incredible ease with which developers can implement a dialog box, and the obvious (and unfortunate) result is one dialog box per function or feature. The developer who wants to create a better user interface often must build his own without much help from GUI tool vendors.

The result, then, when expediency trumps concern for user experience, is too many *unnecessary rooms*—those secondary windows containing functions that should really be integrated into panes or other surfaces within the primary window.

For example, in Adobe Photoshop, if you want to make a simple change to a photo's brightness and contrast (without worrying about adjustment layers), you must go to the Image menu, select the Adjustments submenu, and then select the Brightness/Contrast command. This triggers a dialog box where you can make your adjustments, as shown in Figure 18-2. This sequence is so common that it is completely unremarkable, and yet it is undeniably poor design. Adjusting the image is the primary task in a photo editing application. The image is in the main window, so that's also where the tools that affect it should be. Changing the brightness and contrast isn't a tangential task; it is integral to the application's purpose.
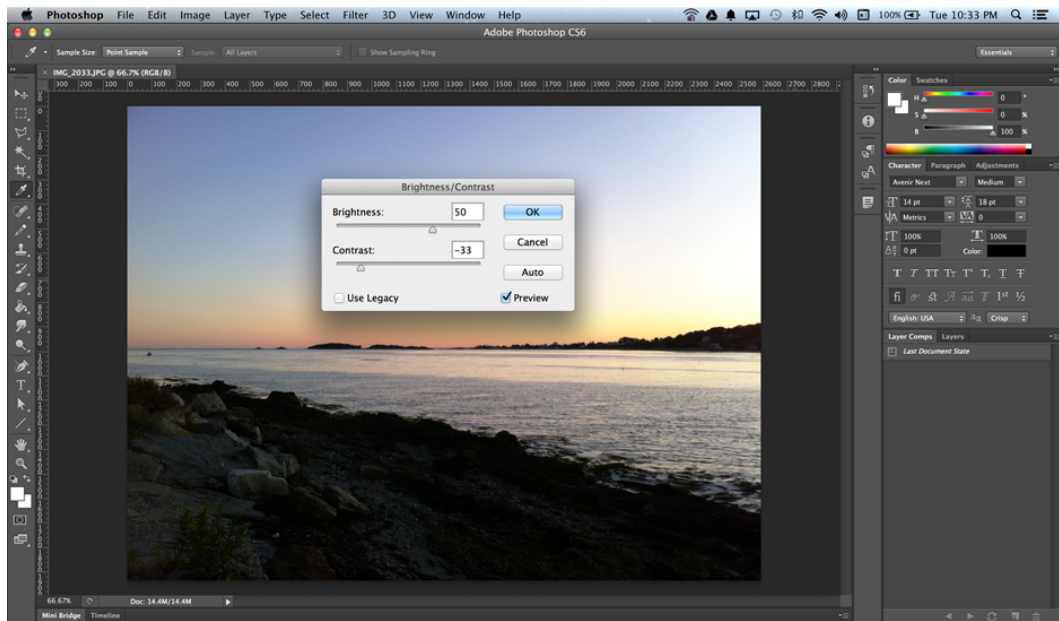
Figure 18-2: One of Adobe Photoshop's many rooms: Brightness/Contrast. We're all used to the fact that we have to invoke a dialog to perform a basic function, so we hardly notice it. But this creates unnecessary work for users, and of course the dialog obscures the most important thing on the screen—the image. Recent versions of Photoshop have begun to move controls like these into modeless sidebars.

Putting functions in a dialog box emphasizes their separateness from the main task. Putting the brightness and contrast adjustment in a dialog box works just fine, but it creates an awkward interaction. From a developer's point of view, adjusting brightness and contrast is a single function, independent of many other functions, so it seems natural to segregate it into its own container. From the user's point of view, however, it is integral to the job at hand and should be obvious in the main window.

The image editing UI is considerably improved in Adobe Lightroom. The application is divided into views or "rooms," each concerned with a specific purpose: Library, Develop, Slideshow, Print, and Web. In the Develop view, brightness and contrast adjustment are presented in a pane on the right side of the main window, along with every other imaginable way of adjusting an image, as shown in Figure 18-3.
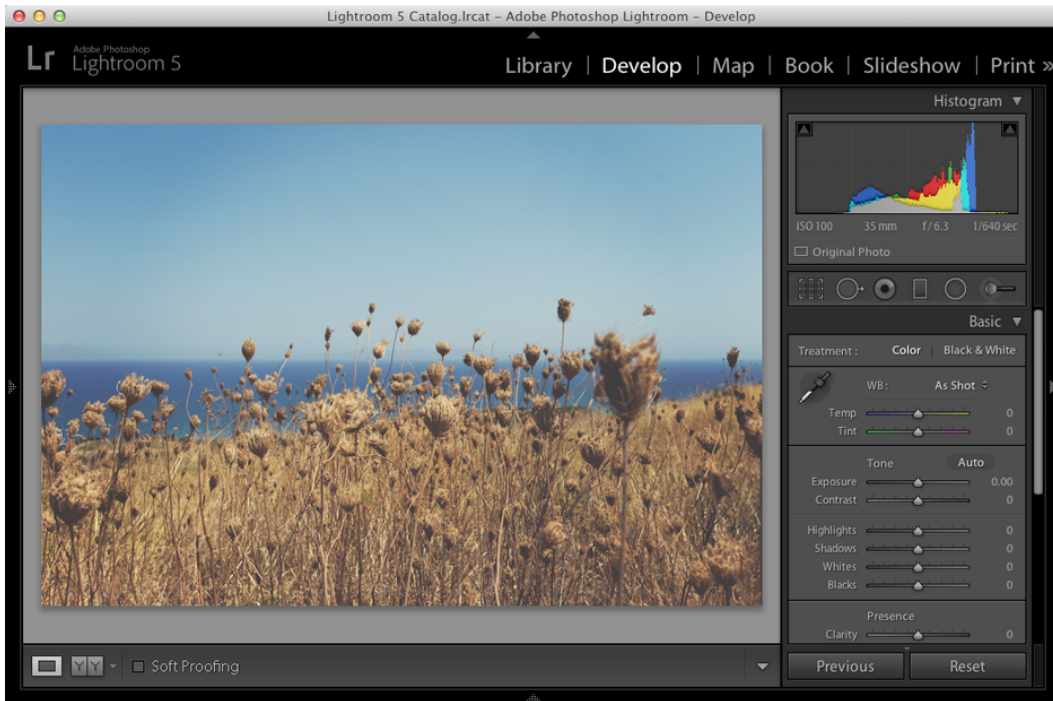
Figure 18-3: Adobe Lightroom shows vast improvements over Photoshop. Critical tools are grouped by purpose and presented directly in the main window, adjacent to the image being adjusted.

> **DESIGN PRINCIPLE**
>
> *Provide functions in the window where they are used.*

Needless to say, a preponderance of secondary windows leads to navigational and window management excise for the user. Try to avoid this kind of *window pollution* in your apps.

Necessary rooms

Sometimes, however, separate rooms for certain functions are appropriate or even necessary. When you want to go swimming, it would be odd if you were offered a living room

full of people as a place to change your clothes. Decorum and modesty are excellent reasons for you to want a separate room in which to change. It is entirely appropriate to provide a separate room when one is needed.

When users perform a function outside their normal sequence of events, it's usually desirable to provide a special place in which to perform it. For example, purging a database is not a normal activity. It involves setting up and using features and facilities that are not part of the normal operation of the database application. The more prosaic parts of the application support daily tasks like entering and examining records, but erasing records en masse is not an everyday occurrence. The purge facility correctly belongs in a separate dialog box. It is entirely appropriate for the application to lead the user into a separate room—a window or dialog—to handle that function.

Using goal-directed thinking, we can examine each function to good effect. If someone is using a graphics application to develop a drawing, his goal is to create an appealing and effective image. All the drawing tools are directly related to this goal, but the pencils, paintbrushes, and erasers are the most tightly connected functions. These tools should be intimately integrated into the workspace itself in the same way that the conventional artist arranges his tools on his drawing board, close at hand. They are ready for immediate use without his having to reach far, let alone having to get up and walk into the next room. In the application, drawing tools should be arrayed on the edges of the drawing space, available with a single click of the mouse. Users shouldn't have to go to the menu or to dialog boxes to access these tools.

For example, Corel Painter arranges artists' tools in trays and lets you move the things that you use frequently to the front of the tray. Although you can hide the various trays and palettes if you want, they appear as the default and are part of the main drawing window. They can be positioned anywhere on the window as well. And if you create a brush that is, for example, thin charcoal in a particular shade of red that you'll need again, you simply "tear it off" the palette and place it wherever you want on your workspace. This is just like laying that charcoal in the tray on your easel. This tool selection design closely mimics how we manipulate tools while drawing.

On the other hand, if you decide to import a piece of clip art, although the function is related to the goal of producing a good drawing, the tools used are not immediately related to drawing. The clip art directory is incongruent with the user's goal of drawing—it is only a means to an end. The conventional artist probably does not keep a book of clip art right on his drawing board. But you can expect that it is close by, probably on a bookshelf immediately adjacent to the drawing board and available without his even getting up. In the drawing application, the clip art facility should be easy to access. But because it involves a whole suite of tools that normally are unneeded, it should be placed in a separate facility: a dialog box.

When you're done creating the artwork, you've achieved your initial goal of creating an effective image. At this point, your goals change. Your new goal is to preserve the picture, protect it, and communicate through it. The need for pens and pencils is over. The need for clip art is over. Leaving these tools behind now is no hardship. The conventional artist would now unpin the drawing from his board, take it into the hall and spray it with fixative, and then roll it up and put it in a mailing tube. He purposely leaves behind his drawing tools. He doesn't want them affected by fixative overspray and doesn't want accidents with paint or charcoal to mar the finished work. He uses mailing tubes infrequently, and they are sufficiently unrelated to the drawing process, so he stores them in a closet. In the software equivalent of this process, you end the drawing application, put away your drawing tools, find an appropriate place on the hard drive to store the image, and send it to someone via e-mail. These functions are clearly separated from the drawing process by goals and motivations.

By examining users' goals, we are naturally guided to an appropriate form for the application. Instead of merely putting every function in a dialog box, we can see that some functions shouldn't be enclosed in a dialog. Others should be put in a dialog that is integral to the interface's main body, and still other functions should be removed from the application.

Menus

Menus are perhaps the oldest idiom in the GUI pantheon. Many concepts and technologies had to come together to make them possible: the mouse, memory-mapped video, powerful (for the time) processors, and pop-up windows. A pop-up window is a rectangle on the screen that appears, overlapping and obscuring the main part of the screen, until it has completed its work, whereupon it disappears, leaving the original screen behind, untouched. The pop-up window is the mechanism used to implement *drop-down menus* (also called *pull-down menus*), as well as dialog boxes.

In modern GUIs, menus are visible across the top row of a screen or window in a *menu bar*. The user points at and clicks one of a set of menu titles on the menu bar, and a list of options (the menu itself) appears in a small window that opens just below it. Menu titles in Windows have a visual *rollover effect* to indicate pliancy (interactivity). A variant of the drop-down menu is a menu that "pops up" when you click—or, more frequently, right-click—an object, even though it has no menu title. This is a *pop-up menu*.

After the menu is open, the user makes a single choice by clicking once or by dragging and releasing. The selection the user makes on the menu either takes immediate effect or launches a dialog box of further options or settings, after which the menu collapses back into its menu title.

Menus as a pedagogic vector

As discussed briefly in Chapter 16, menus represent a *pedagogic vector*. Contrary to user-interface paradigms of 25 years ago, menus and dialog boxes aren't the main methods by which normal users perform everyday functions. So when a user looks at an application for the first time, it is often difficult to size up what that application is capable of. An excellent way to get an impression of an application's power and purpose is to glance at the set of available functions by way of its menus and the dialogs they open. We do this in the same way we look at a restaurant's menu posted at its entrance to get an idea of the type of food, the presentation, the setting, and the price.

> **DESIGN PRINCIPLE** *Use menus to provide a pedagogic vector.*

Understanding the scope of what an application can and can't do is one of the fundamental aspects of creating an atmosphere conducive to learning to use a piece of software. Many otherwise easy-to-use applications put off users because there is no simple, nonthreatening way for them to find out just what the application can do.

Toolbars and direct-manipulation idioms can be too inscrutable for a first-time user to understand, but the textual nature of the menus explains the functions. Reading the words "Format Gallery" (see Figure 18-4) is more enlightening to the new user than trying to interpret an icon button that looks like the one shown in the figure (although ToolTips obviously help).

Figure 18-4: A menu item reading "Format Gallery" is likely to be more enlightening to new users than an icon button like this one. But after new users become intermediates, it's a different story.

For an infrequent user who is somewhat familiar with an application, the menu's main task is as an index to known tools: a place to look when he knows there is a function but he can't remember where it is or what it's called. He doesn't have to keep such trivia in his head. He can depend on the menu to keep it for him, available when he needs it.

For a frequent user, menus provide a stable physical location at which to access one of hundreds of possible commands, or a quick reminder about keyboard shortcuts.

If the main purpose of menus were to execute commands, terseness would be a virtue. But because the main justification of their existence is to teach us about what is available, how to get it, and what shortcuts are available, terseness is really the exact opposite of what we need. Our menus have to explain what a given function does, not just where to invoke it. Therefore, it behooves us to be a little more verbose in our menu item text. Menus shouldn't say "Open," but rather "Open Report"; not "Auto-arrange," but rather "Auto-arrange icons." We should stay away from jargon, because our menu's users won't yet be acquainted with it. A scan of the menus should make clear the scope of the application and the depth and breadth of its various facilities. For this reason, most every function available to the user should be available in menus, so that they are available to be learned.

Another teaching purpose is served by providing hints in the menu to related controls for the same command. Having button icons next to menu commands and including hints that describe keyboard equivalents teaches users about quicker command methods that are available (we discuss this later in the chapter). When you put this information right in the menu, the user may register it subconsciously. It won't intrude upon her conscious thoughts until she is ready to learn it, and then she will find it readily available and already familiar.

Finally, for people to best learn how to use an application, they should be able to examine and experiment without fear of commitment or causing irreparable harm. A global Undo function and the Cancel buttons on dialogs that launch from menu items support this ability well.

Disabled menu items

An important menu convention is to disable (make nonfunctional) menu items when they are unavailable in a given state or are irrelevant to the selected data object or item. The disabled state typically is indicated by lightening or "graying out" the text for the item in the menu. This is a useful and expected idiom. It helps the menu become an even better teaching tool, because users can better understand the context in which certain commands are applicable.

Disable menu items when they are not applicable.

Check mark menu items

Check marks next to menu items are usually used to enable and disable aspects of the application's interface (such as turning toolbars on and off) or adjusting the display of data objects (such as wireframe versus fully rendered images). Users can easily grasp this idiom. It is effective because not only does it provide a functional control, but it also indicates the control's state.

This idiom is probably best used in applications with fairly simple menu structures. If there are lots of possible toggles in the application it can clog up the menu, making it difficult to find more important commands. Opening and scrolling through a menu to find the right item may become laborious. If the attributes in question are frequently toggled, they should also be accessible from a toolbar. If they are infrequently accessed and menu space is at a premium, all similar attributes could be gathered in a dialog box that would provide more instruction and context (as is commonly required for infrequently used functionality).

A check mark menu item is vastly preferable to a *flip-flop menu item* that alternates between two states, always showing the one currently *not* chosen. The problem with the flip-flop menu is the same issue we identified with flip-flop buttons in Chapter 21—namely, that users can't tell if the menu is offering a choice or describing a state. If it says Display Toolbar, does that mean tools are now being displayed, or does it mean that by selecting the option you can begin displaying tools? By using a single check mark menu item instead (the Status bar is either checked or unchecked), you can make the meaning clear.

Icons on menus

Visual symbols next to text items help users recognize them without having to read, so they can be identified faster. They also provide a helpful visual connection to other controls that do the same task. To create a strong visual language, a menu item should show the same icon as its corresponding toolbar icon button.

Use consistent visual symbols on related commands.

With its adoption of the ribbon control, Microsoft has combined menus and toolbars into a single entity in its Office Suite, but for applications continuing to provide standard menus, making a visual link between menu and toolbar remains a powerful means of improving learnability.

Accelerators

Accelerators or keyboard shortcuts provide an easy way to invoke functions from the keyboard. These are commonly function keys (such as F9) or combinations involving modifier keys (Ctrl, Alt, Option, and Command). By convention, they are shown to the right of drop-down menu items—or in ToolTips for applications using the ribbon control—to allow users to learn them as they access menus. Inclusion of these annotations, while called for in style guides, is up to the individual designer, and too often forgotten.

Three tips help you successfully create good accelerators:

- Follow standards.
- Provide for the daily use of accelerators.
- Show how to access accelerators.

Where standard accelerators exist, use them. In particular, this refers to the standard editing set as shown on the Edit menu of most applications. Users quickly learn how much easier it is to press Ctrl+C and Ctrl+V (or the Mac clover-key equivalents) than it is to pull down the Edit menu, select Copy, pull down the Edit menu again, and select Paste. Don't disappoint users when they use your application. Don't forget standards like Ctrl+P for print and Ctrl+S for save.

Identifying the set of commands that will be needed for daily use is the tricky part. You must select the functions that are likely to be used frequently and ensure that those menu items are given accelerators. The good news is that this set won't be large. The bad news is that it can vary significantly from user to user.

The best approach is to perform a triage operation on the available functions. Divide them into three groups: those that are definitely part of everyone's daily use, those that are definitely *not* part of anyone's daily use, and everything else. The first group must have accelerators, and the second group must not. The final group will be the toughest to configure, and it will inevitably be the largest. You can perform a subsequent triage on this group and assign the best accelerators, like F2, F3, F4, and so on, to the winners. More obscure accelerators, like Alt+7, should go to those least likely to be part of someone's everyday commands.

Don't forget to show the accelerator in the menu. An accelerator won't do anyone any good if he has to go to the manual or online help to find it. Put it to the right of the corresponding menu item, where it belongs. Users won't notice it at first, but eventually they will find it, and they will be happy to make the discovery as perpetual intermediates (see Chapter 11). It will give them a sense of accomplishment and a feeling of being an insider. These feelings are well worth encouraging in your customers.

When assigning the key to be paired with the modifier key, try to use the first letter of the command name, such as Ctrl+C for "copy" and Ctrl+P for "paste." This makes the accelerator memorable and easier to learn.

Some applications offer user-configurable accelerators. Often this is a good idea, even a necessity, especially for expert users. Allowing users to customize accelerators on the sovereign applications they use most of the time really lets them adapt the software to their own style of working. Be sure to include a Return to Defaults control along with any customization tools.

Access keys

Access keys or *mnemonics* are another Windows standard (they are also seen in some UNIX GUIs) for adding keystroke commands in parallel to the direct manipulation of menus and dialogs.

The Microsoft style guide covers access keys and accelerators in detail, so we will simply stress that they should not be overlooked. Mnemonics are accessed using the Alt key, arrow keys, and the underlined letter in a menu item or title. Pressing the Alt key places the application in mnemonic mode, and the arrow keys can be used to navigate to the appropriate menu. After the menu opens, pressing the appropriate letter key executes the function. The main purpose of mnemonics is to provide a keyboard equivalent for each menu command. For this reason, mnemonics should be complete, particularly for text-oriented applications. Don't think of them as a convenience so much as a pipeline to the keyboard. Keep in mind that your most experienced users will rely heavily on their keyboards, so to keep them loyal, ensure that the mnemonics are consistent and thoroughly thought out. Mnemonics are not optional.

Cascading menus versus monocline groupings

A variant of the standard drop-down menu provides a secondary menu when the user selects certain items in the primary menu that are marked with a right arrow to the right of the menu item. This mechanism, called a *cascading menu* (see Figure 18-5), is notoriously difficult to use. Cascading menus not only make it much more difficult for users to locate items, but also require precise mouse movements in two dimensions to

navigate them smoothly. (If you trace the path required to select an item in a multilevel cascading menu—such as the Windows Start menu—you will notice that it looks like a path through a maze.)
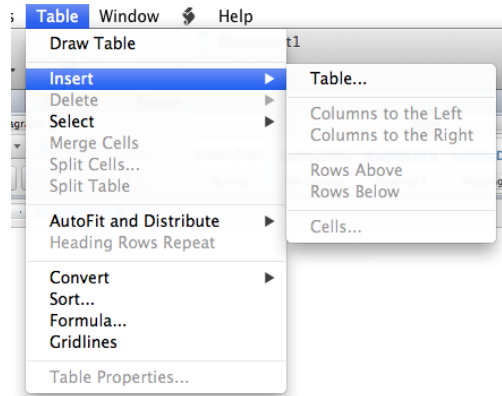
Figure 18-5: An example of a cascading menu from Microsoft Word. Cascading menus make it difficult for users to find and browse the command set, but they do allow menus to usefully contain much larger command sets.

Cascading or hierarchical menus were prevalent in the early days of graphical user interfaces. Menus in modern GUIs have flattened considerably, until most are now only one level deep—a *monocline grouping*, or flat hierarchy. In many cases, especially when optimizing interactions for novice users, flattening the organization of user choices (whether they be commands or objects) can greatly improve the discoverability and learnability of application user interfaces.

The dialog box (which we'll discuss at length later in this chapter) was the mechanism that allowed this simplification of the menu. Dialog boxes enabled software designers to encapsulate all the subchoices of any menu item within a single interactive container.

With the rise of modern high-resolution displays, enough choices can be displayed on a menu bar to organize all an application's functions into about a half-dozen meaningful groups, each group represented by a one-word menu title. The menu for each group also was roomy enough to include all its related functions. The need to go to additional levels of menus is today almost superfluous.

If they must be used at all, cascading menus should be employed only in sophisticated sovereign applications, for rarely used functions. If you implement cascading menus, be sure to allow for a wide threshold in mouse movement so that the submenu doesn't disappear if the mouse cursor deviates slightly from it.

Toolbars, Palettes, and Sidebars

The ubiquitous toolbar is a relatively recent GUI development. Microsoft was the first to introduce the toolbar to mainstream user interfaces. The invention of the toolbar addressed the shortcomings of the modal pull-down menu: slow discoverability and extra physical work to execute functions. Toolbar functions are modeless: always plainly visible, which users can trigger with a single mouse movement and click.

The typical toolbar is a collection of icon buttons in a slab attached to the top (when horizontal) or side (when vertical) of the main window, as shown in Figure 18-6. Essentially, a toolbar consists of one or sometimes two rows (or columns) of visible, immediate, graphically labeled functions.
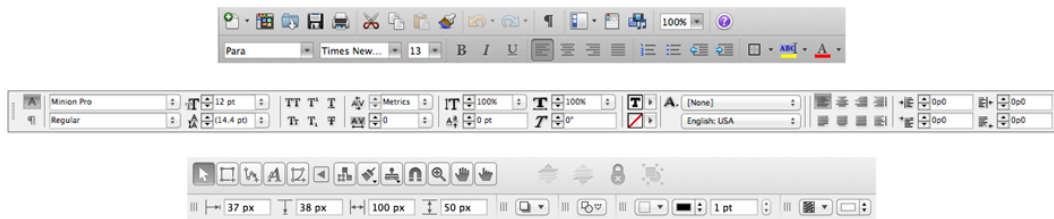
Figure 18-6: Toolbars for Word (top), InDesign (center), and Omnigraffle (bottom) on the Mac. Notice how the Word and InDesign toolbars use icon buttons that show a button outline only on mouseover or selection. This saves space and improves readability.

Toolbars and menus

Toolbars work together with menus to satisfy user needs as they mature: Whereas menus are complete toolsets with the main purpose of teaching inexperienced users and organizing seldom-used advanced functions, toolbars are for frequently used commands and cater to perpetual intermediates. They complement each other perfectly, addressing different user needs at different times.

> **DESIGN PRINCIPLE**
>
> Toolbars give experienced users fast access to frequently used functions.

It's thus a mistake to think of toolbars as simply a speedy version of menus. Think of them rather as receptacles for the essential functions that most users will use the most.

Toolbars versus modeless dialogs

Above, we discussed how the modality of menus make them problematic. Traditionally, two *modeless* tool idioms have been commonly used to get around some of the problems

that reliance on menus introduce. The modeless dialog box (which we'll discuss at length in Chapter 21) is the older of the two idioms. The more recent idiom is the toolbar. Is one better than the other?

Toolbars are modeless, but they don't introduce the conundrums that modeless dialogs do. They also posess two useful characteristics that modeless dialog boxes don't: First, they are visually different from modal dialog boxes, and second, there is no need to worry about dismissing them, because they are always available. They solve other problems, too. Toolbars are incredibly efficient in screen space, especially compared to dialog boxes, and they don't cover what they are operating on.

Users also really seem to understand that the toolbar state reflects what is selected and that interactions with the widgets in a toolbar have a direct and immediate impact on the selection or application, which helps overall learnability.

Modeless dialogs, by contrast, are conventional free-floating windows; users can position them on the screen wherever they like. However, this results in window management excise. While this is certainly a chore, it can also sometimes be handy to have your toolset right next to what you are working on.

 Docking toolbars are a good solution to this conundrum. If you click and drag a docking toolbar and pull it away from the edge of the application, it instantly forms its own small window, frequently with a more compact rectangular layout. You can move it to where you need it, and drag it back to any edge of the application's main window when you are done with it—where it reverts to a toolbar and becomes *docked* against the edge, either vertically or horizontally.

Toolbar buttons

The toolbar gave birth to the toolbar button, or *icon button*, a happy marriage between a button and an icon. Icon buttons are an excellent visual mnemonic for a function. They can be hard for newcomers to interpret, but then, they're not *for* newcomers.

Because toolbars are primarily for providing quick access to frequently used tools, their identifiers must elicit quick recognition from experienced users. The pictorial imagery of symbols suits that role better than text does. Icon buttons have the pliancy of buttons, along with the fast-recognition capability of images. They pack a lot of power into a very small space, but their great strength is also their great weakness: the icon.

Some designers think that they must invent visual metaphors for icon buttons that adequately convey meaning to first-time users. This is a quixotic quest that reflects not only a misunderstanding of the purpose of toolbars but also a futile hope in the magical power of metaphors. As we discussed in Chapter 13, metaphors don't really exist.

The image on the icon button doesn't need to teach users its purpose; it merely needs to be easy to distinguish from the other icons in the set. Users should have help learning its purpose through other means. One of the most effective (as discussed earlier) is including the toolbar icons on the corresponding menu items. In this way, the pedagogy of the menus is extended to an understanding of the toolbar's controls.

ToolTips

It might seem like a good idea to label toolbar buttons with both text and images. This argument has not only logic but also precedent. The original icons on the Macintosh desktop had text subtitles, as did the icon controls on some older web browsers. Icons are useful for allowing quick classification, but beyond that, we need text to tell us *exactly* what the object is for.

The problem is that using both text and images can be very expensive in terms of pixels. Screen space is often too much at a premium to permit verbose labeling of every toolbar or panel icon. Designers who choose to label their icons are trying to satisfy two groups of users with different needs. One wants to learn in a gentle, forgiving environment, and the other knows where frequently used items are but sometimes needs a brief reminder about less-used functions. *ToolTips* provide an effective way to bridge the gap between these two classes of users.

ToolTips are a clever and effective user interface idiom that adds a pedagogical vector to icon buttons without any of the drawbacks of text labeling (see Figure 18-7). In essence, ToolTips provide a text label on a tiny, transient pop-up window. The real genius of Tool-Tips is that they have a well-timed lag that displays the helpful information only after the user has hovered the mouse cursor on the item for a second or so. This is just enough of a delay for the user to be able point to and select the function without getting the ToolTip if she doesn't need it. This ensures that users aren't barraged by little pop-ups as they move the mouse across the toolbar. It also means that if the user forgets what a rarely used icon button is for, she needs to invest only a half-second to find out.
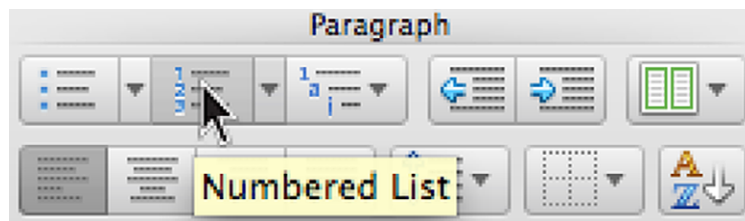
Figure 18-7: This ToolTip from Microsoft Word helps users who have forgotten the icon's meaning without using much real estate on text labels.

ToolTips initially contained a single word or very short phrase that described the hovered-upon icon button. As of Microsoft Office 2007 on Windows, ToolTips now integrate lightweight help content into the ToolTip. By taking advantage of the inherent context sensitivity of ToolTips, better integration with other help mechanisms reduces the excise involved in learning an application.

> **DESIGN PRINCIPLE**   *Use ToolTips with all toolbar and iconic controls.*

ToolTips make the controls on the toolbar much more accessible for intermediate users. As a result, toolbars took the lead as the primary idiom for issuing commands to sovereign applications. This has allowed the menu to recede into the background as a tool for beginners and for invoking advanced or seldom-used functions. The natural order of icon buttons as the primary idiom, with menus as a backup, makes sovereign applications much easier to use. In fact, this trajectory continued into Microsoft Office 2007 with its ribbon control, which replaced menus with visually and textually expressive tabbed toolbars. We discuss the ribbon later in this chapter.

Disabling toolbar controls

Toolbar controls should become disabled if they are not applicable to the current selection. They must not offer a pliant response: The icon button must not depress, for example, and controls should also gray themselves out to make matters absolutely clear.

Some applications make disabled toolbar controls disappear. This can have undesirable effects, especially if the positions of other controls change as well. Users remember toolbar layouts by position. If icon buttons disappear, the trusted toolbar becomes a skittish, tentative idiom that scares new users and disorients the more experienced.

Toolbar control proliferation

After people started to regard the toolbar as something more than just an accelerator for the menu, its growth potential became more apparent. Designers soon realized that there was no reason to restrict the controls on toolbars to icon buttons, and they began inventing new idioms expressly for the toolbar. With the advent of these new constructions, the toolbar truly came into its own as a primary control surface.

After the icon button, the next control to find a home on the toolbar was the combo box, as can be seen in many applications' font style, typeface, and size controls. It is perfectly natural that these selectors be on a toolbar. They offer the same functionality as those

on the drop-down menu, but they also show the current style, font, and font size as a property of the current selection. The idiom delivers more information in return for less effort by users.

After combo boxes were admitted onto the toolbar, the precedent was set, and all kinds of idioms appeared. Some of these toolbar idioms are shown in Figure 18-6.

This variety of controls contributed to a broadening use of the toolbar. When it first appeared, the toolbar was merely a place for fast access to frequently used *functions*. As it developed, controls on the toolbar began to reflect the *state* of the application's data. Instead of an icon button that simply changed a word from plain to italic text, the icon button now began to indicate—by its state—whether the currently selected text was already italicized. The icon button not only controlled the application of the style but also represented the status of the selection with respect to the style.

It was only a matter of time before toolbars began sporting their own menus. The Word toolbar shown in Figure 18-8 shows the Undo drop-down menu. Such sophisticated and powerful idioms continue to push the old-fashioned menu bar further into the background as a purely pedagogic tool.
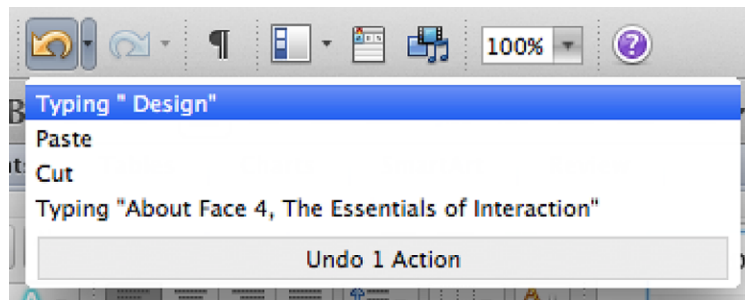
Figure 18-8: Toolbars now contain drop-down menus such as the Undo menu shown here. This provides a compact way to provide powerful functionality.

Movable toolbars

Some applications, such as Adobe's Creative Suite, support movable and detachable toolbars or palettes. Pre-2007, the Microsoft Office suite had a battery of toolbars that users could choose to make visible or invisible. If they were visible, they could be dynamically positioned in one of five locations. They also could be attached—or *docked*—to any of the four sides of the application's main window. If the user dragged the toolbar away from the edge, it configured itself as a floating toolbar, complete with a mini title bar, as shown in Figure 18-9.
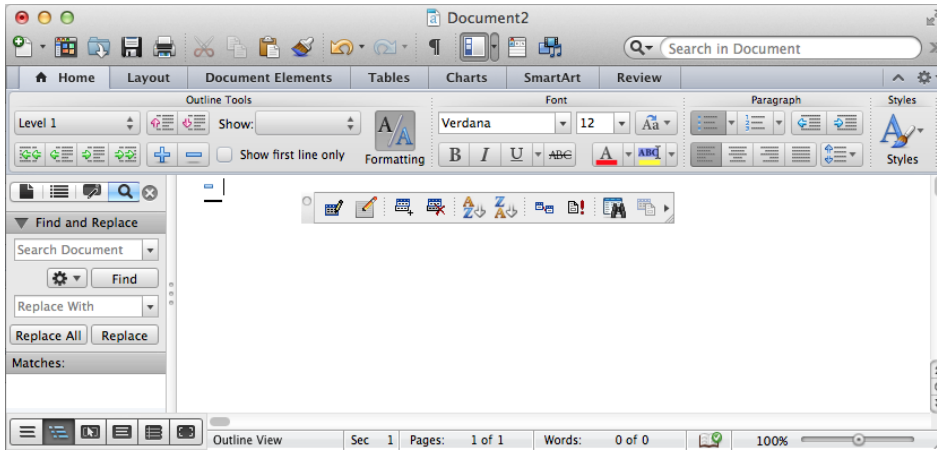
Figure 18-9: Toolbars can be docked horizontally (top), vertically (left), and dragged off the toolbar to form free-floating palettes.

Allowing users to move toolbars around so flexibly also provided the possibility for users to obscure parts of toolbars with other toolbars. Microsoft addressed that problem with an expansion icon button and drop-down menu that appeared only when a toolbar was partly obscured. It provided access to hidden items via a drop-down menu, as shown in Figure 18-10.
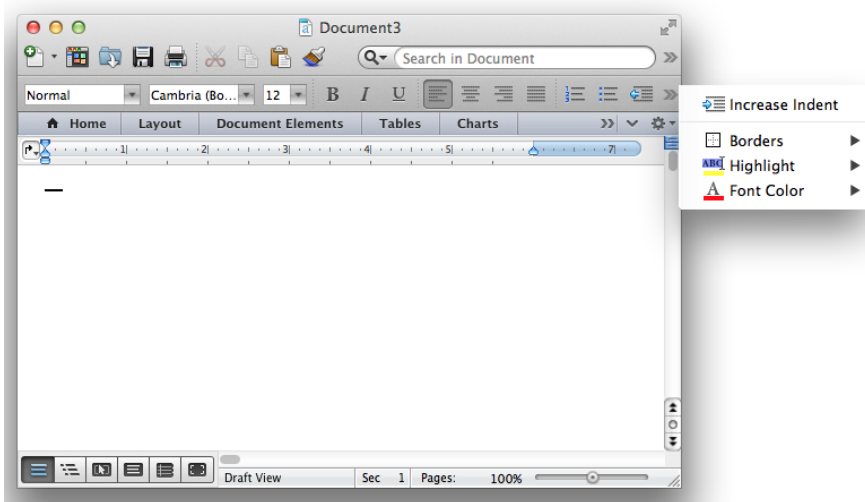
Figure 18-10: Microsoft's clever way of allowing users to overlap toolbars (or fit them in smaller sizes) but still get at all their functions.

Since 2007, Microsoft has moved away from the ultimate flexibility of toolbars to the more predictable and inviting ribbon control (discussed later in the chapter) and single quick-access toolbar. However, they still make use of the same menu idiom for accessing hidden ribbon and toolbar items.

Customizable toolbars

A dilemma arises from the fact that toolbars represent the frequently used functions for *all* users: At least a few of those functions are *different* for *each type* of user. Microsoft arrived at a solution for this conundrum years ago: Ship the application with the best guess at what typical users' daily-use controls are, and let users with more exacting needs customize. (In Office 2007 and later, the ribbon, discussed later, is similarly customizable.)

This solution, while reasonable, gets diluted by the addition of non-daily-use functions to the default toolbars. For example, Word's default toolbar button suite contained functions that were not frequently used, such as the cryptic Insert Autotext. Items like this were perhaps part of a feature checklist or the result of concessions to product management. Although they may have been useful at times, most users did not use them *frequently*. Personas and scenarios are useful tools for helping to sort out which items belong on default toolbar configurations (see Chapters 3 and 4).

Word allows advanced users to customize and configure its ribbon control to their hearts' content. There is a certain danger in providing this level of customizability to the toolbars, because it is possible for a reckless user to make it unrecognizable and unusable. However, it takes some effort to totally wreck things. People generally won't invest much effort in creating something that is ugly and hard to use. More likely, they will make just a few custom changes and enter them one at a time over the course of months or years.

Contextual (pop-up) toolbars

A useful evolution of the toolbar idiom is the contextual toolbar. Similar to a right-click contextual pop-up menu, it provides a small group of icon buttons adjacent to the mouse cursor. In some implementations, the specific icon buttons presented are dependent on the object selected. If text is selected, the buttons provide text-formatting options; if a drawing object is selected, the buttons enable users to change object properties. A variation of this idiom was also popularized with Microsoft Office 2007, where it was called the Mini Toolbar. However, similar idioms have been used in several applications. These include Adobe Photoshop (where the toolbar is docked but changes based on context) and Apple's Logic music production environment (where the toolbar is a modal cursor palette).

The ribbon control

As we discussed earlier in this chapter, Microsoft introduced a new GUI idiom with Office 2007: the ribbon control (see Figure 18-11). In essence, it is an oversized, horizontal, tabbed toolbar with textual labels for groups of functions, as well as a heterogeneous

presentation of icon buttons and textual commands. The tabs provide groupings similar to those used in menus (such as File, Home, Insert, Design, Transitions, Animations, Slide Show, Review, and View in PowerPoint 2010).
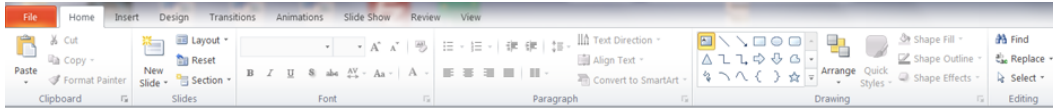
Figure 18-11: The ribbon in PowerPoint replaces the menu system and classic toolbars with what is essentially a tabbed, hybrid menu/toolbar.

Tool palettes

The *tool palette* predates the toolbar as an interaction idiom; the original MacPaint is perhaps the first application to use it. It has been a staple of graphics applications and authoring environments of all kinds ever since.

Tool palettes differ from toolbars in an important way. As already discussed, toolbars are a collection of immediate-access commands that typically act on the current selection, often by changing the values of selected object properties. Tool palettes, on the other hand, contain a set of mutually exclusive controls (meaning that only one may be active at a time), each of which represents an operating mode of the application, including:.

- object creation modes
- object selection modes
- object manipulation modes

Tool palettes also, mostly for historical reasons dating back to MacPaint, tend to be vertically oriented and usually consist of two columns of icon buttons or combo icon buttons. Combo icon buttons can be clicked to reveal other, similar tools. In Adobe Illustrator, for example, clicking and holding on the Eraser gives access to the Scissors and Knife tools as well.

Palettes typically dock and float, mimicking the functionality from the toolbar. Palettes are, as we mentioned, popular in graphics applications, where modeless access to tools is useful—or even critical— for users to maintain a productive flow. Adobe Fireworks (RIP) and other applications originally developed by Macromedia were among the first to provide a more robust docking structure to minimize screen management excise. Recent versions of Photoshop and Illustrator have taken up the idiom, as shown in Figure 18-12.

Figure 18-12: The docked palettes in Adobe Illustrator provide interactivity similar to that of modeless dialog boxes, but they don't require users to spend as much effort and attention invoking, moving, and dismissing dialogs. It doesn't take a lot of imagination to see that these are really quite similar to toolbars in the sense that they use standard controls and widgets to provide application functionality directly, visibly, and persistently in the user interface.

Sidebars, task panes, and drawers

The final step in the evolution of workflow-friendly modeless command idioms was the introduction of the *sidebar* or *task pane*—a pane in the application window dedicated to providing the kind of functions that were formerly delivered through dialog boxes. One of the first applications to do this was Autodesk's 3ds Max, a 3D modeling application that lets you adjust object parameters modelessly through a sidebar. Mainstream applications that feature sidebars include Microsoft Windows Explorer and Internet Explorer with their Explorer Bars, Mozilla Firefox with its Side Bar, Apple's iLife applications with their Inspectors, and Microsoft Office through its Task Pane. Adobe Lightroom has adopted this approach wholeheartedly: Almost all the application's functionality is provided modelessly via sidebars, as shown in Figure 18-13. Recent versions of Adobe Creative Suite applications have begun to adopt similar approaches, with robust tabbed task panes replacing most modal access to functions.

Figure 18-13: Sidebars in Adobe Lightroom replace the need for dozens of dialog boxes. This approach is similar to the palette approach shown in Figure 18-12. But unlike palettes, the sidebar doesn't require users to position it on the screen and doesn't allow users to undock or dismiss it individually (although the entire sidebar may be hidden). This further reduces screen management excise and represents a significant improvement over using dialog boxes to present application functions.

Sidebars hold a lot of promise as an interaction idiom—and they also need not be limited to the sides of the screen. A commonly employed pattern is the dedicated properties area below a document pane or "work space." It lets you modify a selected object while minimizing confusion and screen management excise, as shown in Figure 18-14. Sidebars can contain either persistent controls or contextual controls that change based on the current selection.

Drawers represent a final variant of task panes. The pane can, for purposes of conserving screen real estate for the primary content area, be stowed mostly or completely offscreen in a pop-open drawer. While this can be handy on smaller desktop screens, it also brings back some of the screen management excise that task panes so neatly removed. An alternative to this, supported by many Adobe products, is the ability to hide (and restore) all secondary panes and palettes with a keystroke. This allows power users to temporarily remove the clutter of tools to better focus on the content they are authoring.
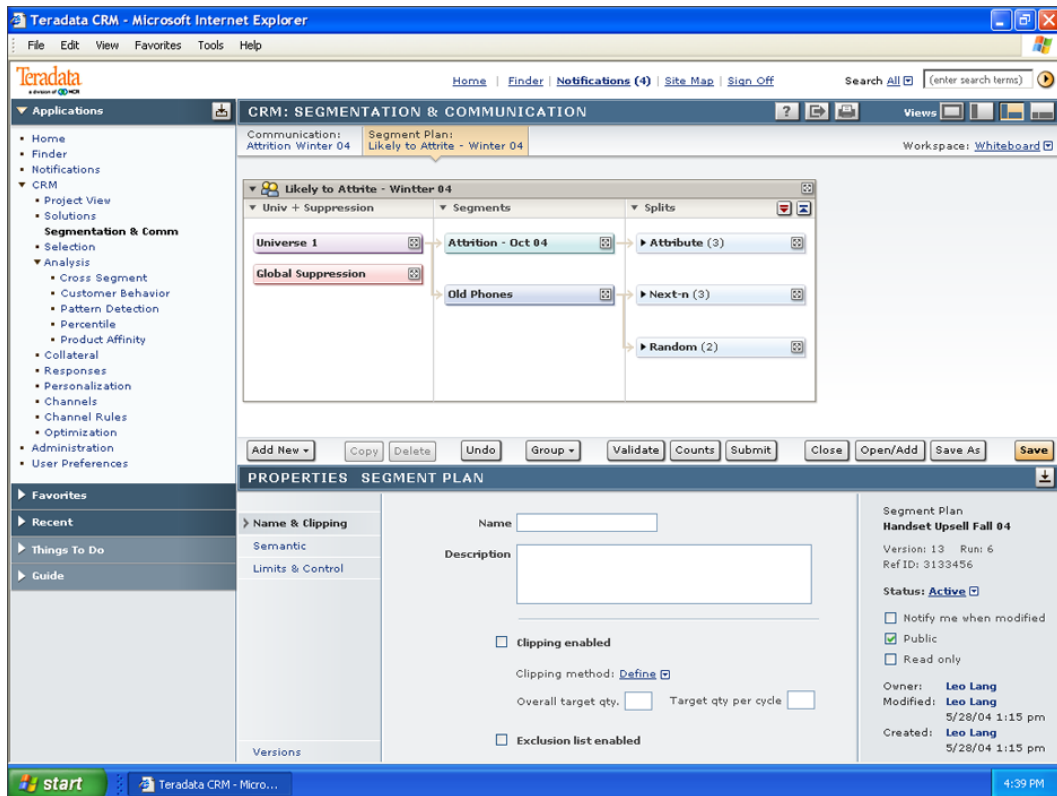
Figure 18-14: This design by Cooper for a customer relationship management (CRM) application features dedicated properties. When the user selects an object in the work space (the top half of the screen, on the left), its properties are displayed below. This retains the user's context and minimizes screen management excise.

Pointing, Selection, and Direct Manipulation

Objects on a screen can be manipulated directly through the use of a *pointing device*. When you think about it, the best way to point to something is with your fingers. They're always handy; you probably have several nearby right now. Their only real drawbacks are that their ends are too blunt for precisely pointing at tiny objects on high-resolution desktop screens, and that most desktop screens still can't recognize being pointed at. (Eyes are also great for pointing, but we usually need them for other things). Because of these limitations, we use a variety of other pointing devices, the most popular— and arguably the most effective— of which is the mouse.

A designer may also take into consideration several other common options for pointers, including trackballs, trackpads, and digitizing tablets. It's worth considering that while the first two behave much like mice (with different ergonomic factors), tablets—as well as their touchscreen cousins—are a bit different.

The mouse is a "relative" pointing device: Moving the mouse moves the cursor *based on the current cursor position*. Tablets and slates are usually "absolute" pointing devices: Each location on the tablet *maps directly to a specific location on the screen*. If you pick up the pen from the top-left corner and put it down in the bottom-right corner, the cursor immediately jumps from the top-left to the bottom-right of the screen.

Touchscreens, when they are implemented on desktop or laptop computers, tend, rather unfortunately as of this writing, to carry over the idea of a pointer, or cursor, even though this is unnecessary and confusing once you can actually point at things using your fingers. Attempting to wed direct touchscreen interactions to relative pointing idioms is simply confusing.

Desktop touchscreen devices—if they must exist—should take a cue from mobile touchscreen UIs, such as iOS, and eliminate the cursor—and everything else that it entails. At the same time, these devices should support horizontal orientation for touch input, since nobody wants to hold his or her arm aloft, interacting with a vertical screen, for hours on end.

The remainder of this section focuses on the more common mouse-based and other relative, pointer-based desktop interactions.

Mouse ergonomics

When you mouse around on the screen, there is a distinct dividing line between near motions and far motions. Either your destination is near enough that you can keep the heel of your hand stationary on your desktop, or you must pick up your hand. When the heel of your hand is down and you move the cursor from place to place, you use the fine motor skills of the muscles in your fingers. When you lift the heel of your hand from the desktop to make a larger move, you use the gross motor skills of the muscles in your arm. Transitioning between gross and fine motor skills is challenging. It involves coordinating two muscle groups that require dexterity to use together, which typically requires time and practice for computer users to master. (It's actually similar to drawing, another skill that requires practice to do well.) Touch typists dislike anything that forces them to move their hands from the home position on the keyboard, because doing so requires a transition between their muscle groups. Similarly, moving the mouse cursor across the screen to manipulate a control forces a change from fine to gross and back to fine motor skills. Don't force users to do this continually.

Clicking a mouse button also requires fine motor control. Without it, the mouse and cursor move inadvertently, botching the intended action. The user must learn to plant the heel of his hand and go into fine motor control mode to position the cursor in the desired location. Then he must maintain that position when he clicks. Furthermore, if the cursor starts far away from the desired control, the user must first use gross motor control to move the cursor near the control before shifting to fine motor control to finish the job. Some controls, such as scrollbars, compound the problem by forcing users to switch between fine and gross motor skills several times to complete an interaction, as shown in Figure 18-15.
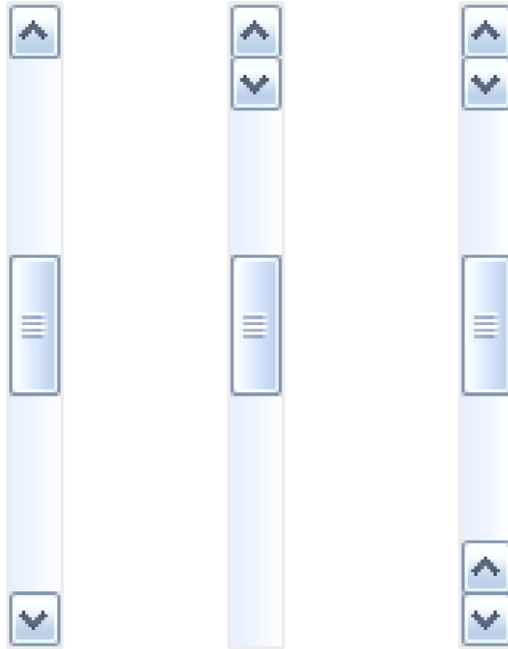
Figure 18-15: The familiar scrollbar, shown on the left, is one of the more difficult-to-use GUI controls. To switch between scrolling up and scrolling down, the user must transition from the fine motor control required for clicking the up button to the gross motor control needed to move her hand to the bottom of the bar. Then she must return to fine motor control to accurately position the mouse and click the down button. If the scrollbar were modified only slightly, as in the center, so that the two buttons were adjacent, the problem would go away. (Macintosh scrollbars can be similarly configured to place both arrow buttons at the bottom.) The scrollbar on the right is a bit visually cluttered, but it has the most flexible interaction. Scroll wheels and capacitive gesture sensors on the input device are also a great solution to the problem.

It's important that designers pay significant attention to users' aptitudes, skills, and usage contexts and make a conscious decision about how much complex motor work using an interface should require. This is a delicate balancing act between reducing complexity and user effort and providing useful and powerful tools. It's almost always a good idea for things that are used together to be placed together.

Not only do less manually-dexterous users find the mouse problematic, but many experienced computer users, particularly touch typists, find the mouse difficult at times. For many data-intensive tasks, the keyboard is superior to the mouse. With minimum movement, a proficient keyboardist has access to around 1600 discrete commands at any one time. The mouse, decidedly less so. It is frustrating to have to pull your hands away from the keyboard to reposition the cursor with the mouse, only to have to return to the keyboard. In the early days of personal computing, it was the keyboard or nothing, and today, it is sometimes the mouse or nothing. Applications should fully support both the mouse and the keyboard for all navigation and selection tasks.

> **DESIGN PRINCIPLE**
>
> *Support both mouse and keyboard use for navigation and selection tasks.*

A significant portion of computer users have some trouble using the mouse, so if we want to be successful, we must design our software in sympathy with them as well as with expert mouse users. This means that each mouse idiom should have at least one nonmouse alternative. Of course, this may not always be possible. It would be ridiculous to try to support drawing interactions without a mouse. However, most enterprise and productivity software lends itself pretty well to keyboard commands.

Mouse buttons and controls

The inventors of the mouse tried to decide how many buttons to put on it, and they couldn't agree. Some believed one button would be sufficient, whereas others swore by two or three buttons. Still others advocated a mouse with several buttons that could be clicked separately or together. Five buttons could yield up to 32 distinct combinations. Ultimately, though, Microsoft chose two for its PC, Apple settled on one button for its Macintosh, and the UNIX workstation community went with three. Apple's extensive user testing determined that the optimum number of buttons for beginners was one, thereby enshrining the single-button mouse in personal computing history. This was unfortunate, because the right mouse button and the context menus typically mapped to it usually come into play soon after someone graduates from beginner status and becomes a perpetual intermediate. A single button sacrifices power for the majority of computer users in exchange for simplicity for beginners. Eventually Apple added a

second (hidden) mouse button, and for a brief time it even added a third beneath a hardware scroll ball. But today's Apple mouse eliminates the affordance for both buttons and gestural swipes with a nearly featureless mouse surface, leaving users to figure out their existence and purpose on their own. Microsoft, on the other hand, seems content to carry on with its familiar two mouse buttons and scroll wheel.

Left mouse button

In general, the left mouse button is used for all the primary direct-manipulation functions, such as triggering controls, making selections, drawing, and so on. The most common meaning of the left mouse button is activation or selection. For standard controls, such as buttons or check boxes, clicking the left mouse button means pushing the button or checking the box. If you are clicking in data, the left mouse button generally means selecting. We'll discuss selection idioms later in the chapter.

Right mouse button

The right mouse button was long treated as nonexistent by Microsoft and many others. Only a few brave developers connected actions to the right mouse button, and those actions were considered to be extra, optional, or advanced functions. When Borland International used the right mouse button as a tool for accessing a dialog box that showed an object's properties, the industry seemed ambivalent toward this action even though it was, as they say, critically acclaimed. This changed with Windows 95, when Microsoft finally followed Borland's lead. Apple reluctantly followed Microsoft's lead as well, and today the right mouse button serves an important and extremely useful role. It enables direct access to properties and other context-specific actions on objects and functions via the ubiquitous context menu.

Scroll wheels and scroll balls

One of the most useful innovations in pointing devices is the scroll wheel. It has several variations, but it is typically a small wheel embedded in the mouse under the user's middle finger. Rolling the wheel forward scrolls the window up, and rolling it backwards scrolls the window down. Pressing it acts like a third mouse button, but few apps actually make good use of this feature. This is fine, because pressing a scroll wheel is rather difficult to do without accidentally scrolling it a bit.

The best thing about the scroll wheel is that it allows users to avoid dealing with the challenges of interacting with scrollbars (see Figure 18-15). Some incarnations of the scroll wheel allow for horizontal as well as vertical scroll control. Some mice, such as the Apple Magic Mouse, have replaced a physical wheel or ball with a capacitive gesture sensor.

Modifier keys

Using *modifier keys* in conjunction with the mouse can extend direct-manipulation idioms. Metakeys include Ctrl, Alt, Command (on Apple computers), and Shift.

Commonly, these keys are used to modify commands. For example, pressing the C key usually inserts a "c" into a text field, but hold the Ctrl key and that same button press means "Copy the selection." In Windows Explorer, holding down the Ctrl key while dragging and dropping a file turns the function from a Move into a Copy. These keys are also commonly used to adjust mouse behavior. Holding down Shift while dragging often constrains cursor movement to a single direction (either up/down or right/left). We'll discuss these conventions more later in the chapter.

Apple has a history of well-articulated standards for the use of modifier keys in combination with the mouse, and there tends to be a fair amount of consistency in their usage. In the Windows world, no single source championed modifier key standards in the same way, but some conventions (often rather similar to Apple's) have emerged.

Using cursor hinting to dynamically show the meanings of modifier keys for non text-related functions is a good idea. While the modifier key is pressed, the cursor should change to reflect the idiom's new function.

> **DESIGN PRINCIPLE**   *Use cursor hinting to show the meanings of modifier keys.*

Pointing

This simple operation is a cornerstone of the graphical user interface and is the basis of nearly every mouse operation. The user moves the mouse until the onscreen cursor is pointing to, or placed over, the desired object. Objects in the interface can know when they are being pointed at, even when they are not clicked. Objects that can be directly manipulated often change their appearance subtly to indicate this attribute when the mouse cursor moves over them. This property is called *pliancy* and is discussed in detail later in this chapter.

Clicking

While the user holds the cursor over a target, he clicks and releases the mouse button. In general, this action is defined to trigger a state change in a control or selecting an

object. In a matrix of text or cells, the click means "Bring the selection point over here." For a pushbutton control, a state change means that while the mouse button is down and directly over the control, the button enters and remains in the pushed state. When the mouse button is released, the button is triggered, and its associated action occurs.

Single-clicking selects data or an object or changes the control state.

However, if the user moves the cursor off the control while still holding down the mouse button, the pushbutton control returns to its unpushed state (but the input focus remains on the control until the mouse button is released). When the user releases the mouse button, input focus is severed, and nothing happens. This provides a convenient escape route if the user changes his mind or inadvertently clicks the wrong button. The mechanics of mouse-down and mouse-up events in clicking are discussed in more detail later in this chapter.

Point-and-click combinations

You can perform two basic operations with the mouse: You can move it to point at different things, and you can click the buttons. Most mouse actions beyond pointing and clicking are a combination of these actions. The following list summarizes the complete set of common mouse actions that can be accomplished without using modifier keys. For the sake of discussion, we have assigned a short name to each of the actions (shown in parentheses):

- Point (point)
- Point, click left button, release (click)
- Point, click right button, release (right click)
- Point, click and hold down left button, drag, release (click and drag)
- Point, click left button, release, quickly again click left button, release (double click)
- Point, click left button and right button simultaneously, release both buttons (chord click)
- Point, double-click without releasing the mouse button, drag, release (double drag)

An expert mouse user may perform all seven actions, but most users will not perform the last two.

Clicking and dragging

This versatile operation has many common uses, including selecting, reshaping, repositioning, drawing, and dragging and dropping. We'll discuss all of these in this chapter and the rest of the book.

As with clicking, it's often important to have an escape hatch for users who become disoriented or have made an error. The Windows scrollbar provides a good example of this: It allows users to scroll successfully without having the mouse directly over the scrollbar, as long as it was first clicked within the scrollbar. (Imagine how hard it would be to use if it behaved like a button.) However, if the user drags too far from the scrollbar, it resets itself to the position it was in before being clicked. This behavior makes sense, since scrolling over long distances requires gross motor movements that make it difficult to stay within the bounds of the narrow scrollbar control. If the drag is too far off base, the scrollbar makes the reasonable assumption that the user didn't mean to scroll. Some applications set this limit too close, resulting in frustratingly temperamental scroll behavior.

Clicking and dragging on a trackpad, while possible, is hardly ideal, especially in the scenarios just described. Drag actions on capacitive surfaces aren't as robust as mouse dragging, and the relatively small surface area of most track pads doesn't help. Apple has gradually enlarged its trackpads while supporting more touch gestures, probably for this very reason.

Double-clicking

If double-clicking is composed of single-clicking twice, it seems logical that the first thing double-clicking should do is the same thing that a single click does. This is indeed its meaning when the mouse is pointing at data. Single-clicking selects something; double-clicking selects something and then takes action on it.

> **DESIGN PRINCIPLE**   *Double-clicking means single-clicking plus action.*

This fundamental interpretation comes from the Xerox Alto/Star by way of the Macintosh, and it remains a standard in contemporary GUI applications. The fact that double-clicking is difficult for less-dexterous users—painful for some and impossible for a few—was largely ignored. The solution to this accessibility problem is to include double-click idioms but ensure that their functions have equivalent single-click idioms.

Although double-clicking file and application icons is well defined, double-clicking most controls has no meaning, and the extra click is discarded. Or, more often, it is interpreted as a second, independent click. Depending on the control, this can be benign or problematic. If the control is a toggle button, you may find that you've just returned it to the state it started in (rapidly turning it on and then off). If the control is one that goes away after the first click, like the OK button in a dialog box, for example, the results can be unpredictable. Whatever was directly below the pushbutton gets the second button-down message. There are also no affordances that indicate if an object is double-clickable. Generally speaking, double-clicking should be avoided where a single click would suffice.

Chord-clicking

Chord-clicking means clicking two buttons simultaneously, although they don't really have to be clicked or released at precisely the same time. To qualify as a chord click, the second mouse button must be clicked before the first mouse button is released.

Chord-clicking can be done in two ways. The first is the simplest: The user merely points to something and clicks both buttons at the same time. This idiom is clumsy and has not found much currency in existing software, although some creative and desperate developers have implemented it as a substitute for the Shift key on selection.

The second method is using chord-clicking to cancel a drag. The drag begins as a simple, one-button drag, and then the user adds the second button. Although this technique sounds more obscure than the first, it actually has found wider acceptance in the industry.

Double-clicking and dragging

This is another expert-only idiom. Faultlessly executing a double-click-and-drag gesture can be like patting your head while rubbing your stomach. Like triple-clicking, it is useful only in specialized sovereign applications. Use it as a variant of selection extension. In Microsoft Word, for example, you can double-click text to select an entire word; so, expanding that function, you can extend the selection word by word by double-dragging.

In a big sovereign application that has many permutations of selection, idioms like this one are appropriate. But for most products, we recommend that you stick with more basic mouse actions.

Mouse-up and mouse-down events

Each time the user clicks a mouse button (or taps a trackpad), the application must deal with two discrete events: the mouse-down event and the mouse-up event. How these events are interpreted varies from platform to platform and product to product. Within a given product (and ideally a platform), these actions should be made rigidly consistent.

When an object is selected, the selection should always take place on mouse-down. The button click may be the first step in a dragging sequence, and you can't drag something without first selecting it.

> **DESIGN PRINCIPLE**
>
> *Mouse-down over an object or data should select the object or data.*

On the other hand, if the cursor is positioned over a control rather than selectable data, the action on the mouse-down event is to *tentatively* activate the control's state transition. When the control finally sees the button-up event, it then commits to the state transition, as shown in Figure 18-16.

Figure 18-16: These images depict feedback and state change of a check box in Windows 8. The first image shows an unselected check box. The second is the mouseover state (or hover). The third shows the feedback to the click (or mouse-down). The fourth shows what happens when the button is released (mouse-up) but with a hover. The final image shows the selected state of the check box without a hover. Notice that although the click has visual feedback, the check box control doesn't register a state change until the mouse-up or release.

> **DESIGN PRINCIPLE**
>
> *Mouse-down over controls means proposing an action; mouse-up means committing to an action.*

This mechanism allows users to gracefully bow out of an inadvertent click. After clicking a button, for example, the user can just move the mouse outside of the button and release the mouse button. For a check box, the meaning is similar: On mouse-down the check box shows that it has been activated, but the check doesn't actually appear until

the mouse-up transition. This idiom is called pliant response hinting and is further described in Chapter 13.

Trackpads, trackballs, and gesture sensors

Almost anybody who has used a laptop has used a trackpad. Many people eschew their mouse when they take their laptop with them to meetings, coffee shops, kitchen tables, and bed. Keep in mind that trackpads are a bit more prone to glitchy behavior than mice, since they rely on finger contact on their capacitive surface, and that drag and drop or fine positioning control are difficult. This shouldn't affect your design of typical desktop apps much, but it's something to consider if you know your users will be making heavy or exclusive use of trackpads.

Windows trackpads typically include distinct left and right buttons in addition to the trackpad. Recent Apple trackpads have cleverly and invisibly built these buttons into the trackpad itself. Pressing the Apple trackpad yields a satisfying button click and activates either a left or right mouse button action. A one-finger tap equals a left-click, and a two-finger tap equals a right-click.

Trackballs are uncommon but are still used for specialized applications where fine movement control is desirable and space is at a premium, or where the ball's rotational movement maps well to the manipulation of objects onscreen (3D modeling applications). Click-and-drag operations are somewhat awkward using trackballs, so any dedicated application using a trackball as input probably should be designed to minimize the need for such interactions.

Mice (and trackpads) with multi-touch gesture sensors are becoming more and more common and are the standard for Apple's computers. The operating system typically reserves supported gestures for its own use. But in the event that gestures are made available for use by your application, think carefully about their implementation. Such gestures shouldn't interfere with OS gestures without excellent reason, and they shouldn't be the primary method of accessing functionality or performing navigation. Because gestures lack affordance, they should be considered power-user features, rather like keyboard accelerators and other command shortcuts.

Cursors

Pointing and selection on the desktop are achieved via the *cursor*, the visible representation of the mouse's position onscreen. By convention, it is normally a small arrow pointing diagonally up and left, but under application control it can change to any shape as long as it stays relatively small (32 × 32 pixels in Windows 8). Because the cursor frequently must resolve to a single pixel to point at small things, there must be some way for

the cursor to indicate precisely which pixel is the one pointed to. This is accomplished by designating a single pixel of any cursor as the actual locus of pointing, called the *hotspot*. For the standard arrow, the hotspot is, logically, the tip of the arrow. Regardless of which shape the cursor assumes, it always has a single hotspot pixel.

As discussed, the key to successful direct manipulation is rich visual feedback. It should be obvious to users which aspects of the interface can be manipulated, which are informational, and which are décor. Especially important for creating effective interaction idioms is attention to mouse cursor hinting, as discussed in Chapter 13.

Selection

The act of choosing an object or control is called *selection*. This is a simple idiom, typically accomplished by pointing to and clicking the item in question (although there are other keyboard- and button-actuated ways to do this). Selection is often the basis for more-complex interactions. After the user chooses something, she is in the appropriate context to perform an action on that thing. The sequence of events implied by such an idiom is called *object verb ordering*.

Command ordering and selection

At the foundation of every user interface is the way in which the user can express commands. Almost every command has a *verb* that describes the action and an *object* that describes what will be acted on.

If you think about it, you can express a command in two ways: with the verb first, followed by the object, or with the object first, followed by the verb. ("Throw me that ball" vs. "That ball, throw it to me.") These are commonly called *verb-object* and *object-verb* orders, respectively. Modern user interfaces use both orders.

Verb-object ordering is consistent with how commands are formed in English. As a result, it was only logical that command-line systems mimic this structure in their syntax. (For example, to **rem**ove a file in UNIX, you type **rm *filename*.txt**.)

When graphical user interfaces first emerged, it became clear that verb-object ordering created a problem. Without the rigid, formal structures of command-line idioms, graphical user interfaces must use the construct of *state* to tie together different interactions in a command. If the user chooses a verb, the system must then enter a state—a mode—to indicate that it is waiting for the user to select an object to act on. In the simple case, the user then chooses a single object, and all is well. However, if the user wants to act on more than one object, the system can know this only if the user tells it in advance how many operands he will enter, or if the user enters a second command indicating that he has completed his object list. These are both clumsy interactions and require users to

express themselves in an unnatural manner that is difficult to learn. What works just fine in a highly structured linguistic environment falls apart in the looser universe of the graphical user interface.

With an object-verb command order, we don't need to worry about termination. Users select which objects will be operated on and then indicate which verb to execute on them. The application then executes the indicated function on the selected data. A benefit of this is that users can easily execute a series of verbs on the same complex selection. A second benefit is that when the user chooses an object, the application can show only appropriate commands. This potentially reduces the user's cognitive load and reduces the amount of visual work required to find the command. (In a visual interface, all commands should be represented visually.)

Notice that a new concept has crept into the equation—one that doesn't exist and that isn't needed in a verb-object world. That new concept is called *selection*. Because the identification of the objects and the verb are not part of the same interaction, we need a mechanism to indicate which operands are selected.

The object-verb model can be difficult to understand in the abstract, but selection is an idiom that is easy to grasp and, once shown, is rarely forgotten. (Clicking an e-mail in Outlook and deleting it, for example, quickly becomes second nature.) Explained through the linguistic context of the English language, it doesn't sound too useful that we must choose an object first. On the other hand, we use this model frequently in our nonlinguistic actions. We pick up a can and then use a can opener on it.

In interfaces that don't employ direct manipulation, such as some modal dialog boxes, the concept of selection isn't always needed. Dialog boxes naturally come with one of those object-list-completion commands: the OK button. Here, users may choose a function first and one or more objects second.

While object-verb orderings are more consistent with the notion of direct manipulation, there are certainly cases where the verb-object command order is more useful or usable. These are cases where it isn't possible or reasonable to define the objects up front without the context of the command. An example is mapping software, where the user probably can't always select the address he wants to map from a list (although we should allow this for his address book). Instead, it is most useful for him to say "I want to see a map for the following address...."

Discrete and contiguous selection

Selection is a pretty simple concept, but a couple of basic variants are worth discussing. Because selection typically is concerned with objects, these variants are driven by two broad categories of selectable data.

In some cases, data is represented by distinct visual objects that can be manipulated independently of other objects. Icons on the desktop and vector objects in drawing applications are examples. These objects are also commonly selected independently of their spatial relationships with each other. We refer to these as *discrete data* and to their selection as *discrete selection*. Discrete data is not necessarily homogeneous, and discrete selection is not necessarily contiguous.

Conversely, some applications represent data as a matrix of many small, contiguous pieces of data. The text in a word processor or the cells in a spreadsheet are made up of hundreds or thousands of similar little objects that together form a coherent whole. These objects are often selected in contiguous groups, so we call them *contiguous data* and selection within them *contiguous selection*.

Both contiguous selection and discrete selection support single-click selection and click-and-drag selection. Single-clicking typically selects the smallest useful discrete amount, and clicking and dragging selects a larger quantity, but there are other significant differences.

There is a natural order to the text in a word processor's document—it consists of contiguous data. Scrambling the order of the letters destroys the document's sense. The characters flow from the beginning to the end in a meaningful continuum, and selecting a word or paragraph makes sense in the context of the data. Random, disconnected selections generally are meaningless. Theoretically it is possible to allow a discrete, discontiguous selection, such as several disconnected paragraphs. However, the user's task of visualizing the selections and avoiding inadvertent, unwanted operations on them is more trouble than it's worth.

Discrete data, on the other hand, has no inherent order. Many meaningful orders can be imposed on discrete objects, such as sorting a list of files by their modification dates. However, the lack of a single inherent relationship means that users are likely to want to make discrete selections, such as Ctrl+clicking multiple files that are not listed adjacently. Of course, users may also want to make contiguous selections based on some organizing principle (such as the old files at the bottom of that chronologically ordered list). The utility of both approaches is evident in a vector drawing application (such as Illustrator or PowerPoint). In some cases, the user will want to perform a contiguous selection on objects that are close together, and in other cases, she will want to select a single object.

Mutual exclusion

Typically, when a selection is made, any previous selection is unmade. This behavior is called *mutual exclusion*, because the selection of one excludes the selection of the other.

Typically, the user clicks an object and it becomes selected. That object remains selected until the user selects something else. Mutual exclusion is the rule in both discrete and contiguous selection.

Some applications allow users to deselect a selected object by clicking it a second time. This can lead to a curious condition in which nothing is selected, and there is no insertion point. You must decide whether this condition is appropriate for your product.

Additive selection

Mutual exclusion is often appropriate for contiguous selection because users cannot see or know what effect their actions will have if selections can readily be scrolled off the screen. Selecting several independent paragraphs of text in a long document might be useful, but it isn't easily controllable. It's also easy for users to get into situations where they cause unintended changes because they cannot see all the data they are acting on. Scrolling—not contiguous selection—creates the problem, but most applications that manage contiguous data are scrollable.

However, if there is no mutual exclusion for interactions involving discrete selection, the user can select many independent objects by clicking them sequentially, in what is called *additive selection*. A list box, for example, can allow users to make as many selections as desired and to deselect them by clicking them a second time.

Most discrete-selection systems implement mutual exclusion by default and allow additive selection only by using a modifier key. In Windows, the Shift key is used most frequently for this task in contiguous selection; the Ctrl key is frequently used for discrete selection. In a drawing application, for example, after you've clicked to select a graphical object, typically you can add another one to your selection by Shift-clicking.

Interfaces employing contiguous selection should not, generally speaking, allow additive selection (at least not without an overview mechanism to make additive selections manageable). However, contiguous-selection interfaces do need to allow selection to be *extended*. Again, modifier keys should be used. In Word, the Shift key causes everything between the initial selection and the Shift+click to be selected.

Some list boxes, as well as the file views in Windows (both examples of discrete data), do something a bit strange with additive selection. They use the Ctrl key to implement "normal" discrete additive selection, but then they use the Shift key to *extend* the selection, as if it were contiguous, not discrete data. In most cases this mapping adds confusion, because it conflicts with the common idiom for discrete additive selection.

Group selection

The click-and-drag operation is also the basis for group selection. For contiguous data, it means "extend the selection" from the mouse-down point to the mouse-up point. This can also be modified with modifier keys. In Word, for example, Ctrl+click selects a complete sentence, so Ctrl+drag extends the selection sentence by sentence. Sovereign applications should rightly enrich their interaction with these sorts of variants as appropriate. Experienced users will eventually come to memorize and use them, as long as the variants are manually simple.

In a collection of discrete objects, the click-and-drag operation generally begins a drag-and-drop move. If the mouse button is clicked in an area between objects, rather than on any specific object, it has a special meaning. It creates a *drag rectangle*, as shown in Figure 18-17.

Figure 18-17: When the cursor is not on any particular object at mouse-down time, the click-and-drag operation normally creates a drag rectangle that selects any object wholly enclosed by it when the mouse button is released. This is a familiar idiom to users of drawing applications and many word processors. This example is taken from Windows Explorer. The rectangle has been dragged from the upper left to the lower right.

A drag rectangle is a dynamically sizable rectangle whose upper-left corner is the mouse-down point and whose lower-right corner is the mouse-up point. When the mouse button is released, any and all objects enclosed within the drag rectangle are selected as a group.

Visual indication of selection

Selected objects must be clearly, boldly indicated as such to users. The selected state must be easy to spot on a crowded screen, must be unambiguous, and must not obscure normally visible details of the object.

You must ensure that, in particular, users can easily tell which items are selected and which are not. It's not good enough just to be able to see that the items are different. Keep in mind that a significant portion of the population is color-blind, so color alone is insufficient to distinguish between selections.

Historically, inversion has been used to indicate selection (such as making white pixels black and black pixels white). Although this is visually bold, it is not necessarily very readable, especially when it comes to full-color interfaces. Other approaches include colored backgrounds, outlines, pseudo-3D depression, handles, and animated marquees.

In drawing, painting, animation, and presentation applications, where users deal with visually rich objects, it's easy for selections to get lost. The best solution here is to add selection indicators to the object, rather than merely indicating selection by changing any of the selected object's visual properties. Most drawing applications take this approach, with *handles*: little boxes that surround the selected object, providing points of control.

With irregularly shaped selections (such as those in an image-manipulation application like Adobe Photoshop), handles can be confusing and get lost in the clutter. However, there is one way to ensure that the selection will always be visible, regardless of the colors used: Indicate the selection by movement.

One of the first applications on the Macintosh, MacPaint, had a wonderful idiom in which a selected object was outlined with a simple dashed line, and the dashes all moved in synchrony around the object. The dashes looked like ants in a row; thus, this effect earned the colorful sobriquet *marching ants*. Today, this is commonly called a *marquee*, after the flashing lights on old cinema signs that exhibited similar behavior.

Adobe Photoshop uses this idiom to show selected regions of photographs, and it works extremely well. (Expert users can toggle it off and on with a keystroke so that they can see their work without visual distraction.) The animation is not hard to do, although it takes some care to get it right, and it works regardless of the color mix and intensity of the background.

Insertion and replacement

As we've established, selection indicates on which object subsequent actions will operate. If that action involves creating or pasting new data or objects (via keystrokes or

a Paste command), they are somehow added to the selected object. In discrete selection, one or more discrete objects are selected, and the incoming data is handed to the selected discrete objects, which process the data in their own ways. This may cause a *replacement* action, in which the incoming data replaces the selected object. Alternatively, the selected object may treat the incoming data in some predetermined way. In PowerPoint, for example, when a shape is selected, incoming keystrokes result in a text annotation of the selected shape.

In contiguous selection, however, the incoming data always replaces the currently selected data. When you type in a word processor or text-entry box, you replace what is selected with what you are typing. Contiguous selection exhibits a unique quirk: The selection can simply indicate a location *between* two elements of contiguous data, rather than any particular element of the data. This in-between place is called the *insertion point*.

In a word processor, the *caret* (usually a blinking vertical line that indicates where the next character will go) indicates a position between two characters in the text, without actually selecting either one. By pointing and clicking anywhere else, you can easily move the caret, but if you drag to extend the selection, the caret disappears and is replaced by the contiguous selection of text.

Spreadsheets also use contiguous selection but implement it somewhat differently than word processors do. The selection is contiguous because the cells form a contiguous matrix of data, but there is no concept of selecting the space between two cells. In the spreadsheet, a single click selects exactly one whole cell. There is currently no concept of an insertion point in a spreadsheet, although the design possibilities are intriguing. (That is, select the line between the top and bottom of two vertically adjacent cells and start typing to insert a row and fill a new cell in a single action.)

A blend of these two idioms is possible as well. In PowerPoint's slide-sorter view, insertion-point selection is allowed, but single slides can be selected too. If you click a slide, that slide is selected, but if you click in between two slides, a blinking insertion-point caret is placed there.

If an application allows an insertion point, contiguous objects must be selected by either clicking and dragging, or if they are part of the same logical group, by double- or triple-clicking. Most people select text by dragging the mouse across it. This means that the user will be doing quite a bit of clicking and dragging in the normal course of using the application, with the side effect that any drag-and-drop idiom will be more difficult to express. You can see this in Word, where dragging and dropping text involves first a click-and-drag operation to make the selection, and then another mouse move back into the selection to click and drag again for the actual move. To do the same thing, Excel makes you find a special pliant zone (only a pixel or two wide) on the border of

the selected cell. To move a discrete selection, the user must click and drag the object in a single motion. To relieve the click-and-drag burden of selection in word processors, other direct-manipulation shortcuts are also implemented, like double-clicking to select a word.

Drag and drop

Of all the direct-manipulation idioms, nothing distinguishes a WIMP interface more than the drag-and-drop operation: clicking and holding the button while moving an object across the screen and releasing it in a meaningful location. Surprisingly, drag and drop isn't used as widely as we'd like to think, and it certainly hasn't lived up to its full potential.

In particular, the popularity of the web and the myth that web-like behavior is synonymous with superior ease of use have set back the development of drag and drop on the desktop. Developers have mistakenly emulated the crippled interactions of web browsers in other, far less appropriate contexts. Luckily, as web technology has been refined, developers have been able to provide rich drag-and-drop behavior in the browser. Although this task is still somewhat challenging, it seems that there has been a resurgence in rich, expressive command idioms for all platforms.

We might define *drag and drop* as "clicking an object and moving it to a new location," although that definition is somewhat narrow in scope for such a broad idiom. A more accurate description of drag and drop is "clicking an object and moving it to imply a transformation."

The Macintosh was the first successful system to offer drag and drop. It raised a lot of expectations with the idiom that were never fully realized for two simple reasons. First, drag and drop wasn't a systemwide facility, but rather an artifact of the Finder, a single application. Second, because the Mac was at the time a single-tasking computer, the concept of drag and drop between applications didn't surface as an issue for many years.

To Apple's credit, it described drag and drop in its first user-interface standards guide. On the other side of the fence, Microsoft not only failed to put drag-and-drop aids in its early releases of Windows but also didn't even describe the procedure in its developer documentation. However, Microsoft eventually caught up and even pioneered some novel uses of the idiom, such as movable toolbars and dockable palettes.

Generally we use the term "direct manipulation" to refer to all kinds of GUI interaction idioms, but drag and drop has two levels of directness. First, with true direct-manipulation idioms, dragging and dropping represents putting the object somewhere. Examples include moving a file between two directories, opening a file in a specific

application (by dropping a file icon onto an application icon), or arranging objects on a canvas in drawing applications.

The second type of drag-and-drop idiom is a little more indirect: The user drags the object to a specific area or onto another object to perform a function. These idioms are less popular but can be very useful. A good example of this can be found in the OS X Automator, as shown in Figure 18-18.
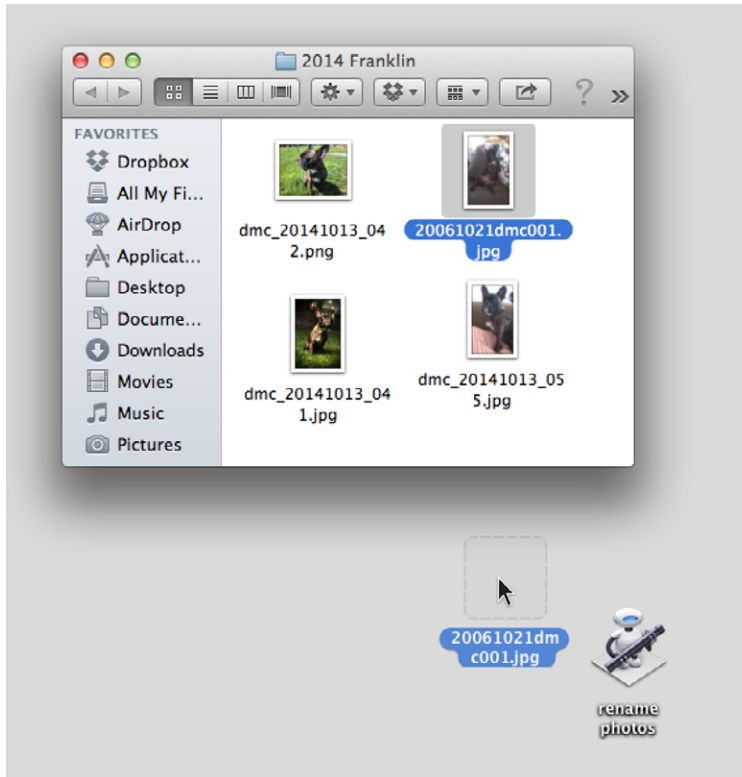
Figure 18-18: Apple's Automator tool in OS X allows users to set up common workflows, such as renaming an image, that are then represented as an icon. Users can then drag and drop files or folders onto the workflow icon to perform the function. While strictly speaking this isn't direct manipulation, it does provide a reasonably direct way to invoke a command.

Visual feedback for drag and drop

As we've discussed, an interface should visually hint at its pliancy—either statically, in how it is drawn, or actively, by becoming animated as the cursor passes over it. The idea that an object can be dragged is easily learned idiomatically. It is difficult for the user to forget that an icon, selected text, or other distinct object can be directly manipulated

after he learns the behavior. However, he may forget the details of the action, so feedback is very important *after* the user clicks the object and starts dragging. The first-timer or very infrequent user will probably also require some additional help to get started (such as textual hints built into the interface). Forgiving interactions and Undo encourage users to try direct manipulation without trepidation.

As soon as the user clicks the mouse button with the cursor on an object, that object becomes the source object for the duration of the drag and drop. As the user moves the mouse around with the button held down, the cursor passes over a variety of objects. It should be obvious which of these objects are meaningful drop targets. Until the button is released, these are called *drop candidates*. A drag can have only one source and one target, but there may be many drop candidates.

The only task of each drop candidate is to visually indicate that the hotspot of the captive cursor is over it. This means that it will accept the drop—or at least comprehend it—if the user releases the mouse button. Such an indication is, by its nature, active visual hinting.

> **DESIGN PRINCIPLE**
>
> *Drop candidates must visually indicate their receptivity.*

The weakest way to offer the visual indication of receptivity to being dropped upon is by changing the cursor. It is the cursor's primary job to represent what is being dragged. It is best to leave indication of drop candidacy to the drop candidate itself.

> **DESIGN PRINCIPLE**
>
> *The drag cursor must visually identify the source object.*

It is important that these two visual functions not be confused. Unfortunately, Microsoft seems to have done so in Windows, with its use of cursor hinting to indicate that something *is not* a drop target. This decision was likely made more for the ease of coding than for any design considerations. It is much easier to change the cursor than it is to highlight drop candidates to show their drop receptivity. The cursor's role is to represent the master—the dragged object. It should not be used to represent the drop candidate.

As if that wasn't bad enough, Microsoft performs this cursor hinting using the detestable circle with a bar—the universal icon for Not Permitted. This symbol is an unpleasant idiom, because it tells users what they can't do. It is negative feedback. The user can easily construe its meaning to be "Don't let go of the mouse now, or you'll do irreversible

damage" instead of "Go ahead and let go now; nothing will happen." Adding the Not Permitted symbol to cursor hinting is an unfortunate combination of two weak idioms and should be avoided, regardless of what the Microsoft style guide says.

After the user finally releases the mouse button, the current drop candidate becomes the *target*. If the user releases the mouse button in the interstice between valid drop candidates, or over an invalid drop candidate, there is no target, and the drag-and-drop operation ends with no action. Silence, or visual inactivity, is a good way to indicate this termination. It isn't a cancelation, exactly, so there is no need to show a cancel indicator.

Indicating drag pliancy

Active cursor hinting to indicate drag pliancy is problematic. In an increasingly object-oriented world, more things can be dragged than not. A cursor flicking and changing rapidly can be more visual distraction than help. One solution is to just assume that things can be dragged and let users experiment. This method is reasonably successful in the Windows Explorer and Macintosh Finder windows. Without cursor hinting, drag pliancy can be a hard-to-discover idiom, so you might consider building some other indication into the interface, such as a textual hint or ToolTip-style pop-up.

After the source object is picked up and the drag operation begins, there must be some visual indication of this. The most visually rich method is to fully animate the drag operation, showing the entire source object moving in real time.

One problem is that a drag-and-drop operation can require a pretty precise pointer. For example, the source object may be 6-centimeters square, but it must be dropped on a target that is 1-centimeter square. The source object must not obscure the target, and because the source object is big enough to span multiple drop candidates, we need to use a cursor hotspot to precisely indicate which candidate it will be dropped on. This means that dragging a transparent outline or thumbnail of the object may be much better than actually dragging an exact image of the source object or data. It also means that the dragged object can't obscure the normal arrow cursor. The tip of the arrow is needed to indicate the exact hotspot.

Dragging an outline also is appropriate for most repositioning, because the outline can be moved relative to the source object, still visible in its original position.

Indicating drop candidacy

As the cursor traverses the screen, carrying with it an outline of the source object, it passes over one drop candidate after another. These drop candidates must visually

indicate that they are aware of being considered as potential drop targets. By visually changing, the drop candidate alerts users that they can do something constructive with the dropped object. (Of course, this requires that the software be smart enough to identify meaningful source-target combinations.)

A point so obvious that it is difficult to see is that the only objects that can be drop candidates are ones that are currently visible. A running application doesn't have to worry about visually indicating its readiness to be a target if it isn't visible. Usually, the number of objects occupying screen real estate is very small—a couple dozen at most. This means that the implementation burden should not be overwhelming.

Insertion targets

In some applications, the source object can be dropped into the spaces between other objects. Dragging text in Word is such an operation, as are most reordering operations in lists or arrays. In these cases, a special type of visual hinting is drawn on the background "behind" the GUI objects of the application or in its contiguous data. This is an *insertion target*.

Rearranging slides in PowerPoint's slide-sorter view is a good example of this type of drag and drop. The user can pick up a slide and drag it into a different place in the presentation. As the user drags, the insertion target (a vertical black bar that looks like a big text edit caret) appears between slides. Word, too, shows an insertion target when you drag text. Not only is the loaded cursor apparent, but you also see a vertical dotted-line bar showing the precise location, between characters, where the dropped text will land.

Whenever something can be dragged and dropped on the space between other objects, the application must show an insertion target. Like a drop candidate in source-target drag and drop, the application must visually indicate where the dragged object can be dropped.

Visual feedback at completion

If the source object is dropped onto a valid drop candidate, the appropriate operation then takes place. A vital step at this point is visual feedback that the operation has occurred. For example, if you're dragging a file from one directory to another, the source object must disappear from its source and reappear in the target. If the target represents a function rather than a container (such as a print icon), the icon must visually hint that it received the drop and is now printing. It can do this with animation or by otherwise changing its visual state.

Auto-scrolling

What action should the application take when the selected object is dragged beyond the border of the enclosing application? Of course, the object is being dragged to a new position, but is that new position inside or outside of the enclosing application?

Take Microsoft Word, for example. When a piece of selected text is dragged outside the visible text window, is the user saying "I want to put this piece of text into another application" or is he saying "I want to put this piece of text somewhere else in this same document, but that place is currently scrolled off the screen"? If it's the former, we proceed as already discussed. But if the user desires the latter, the application must automatically scroll (*auto-scroll*) in the direction of the drag to reposition the selection at a distant, not currently visible location in the same document.

Auto-scroll is a very important adjunct to drag and drop. Wherever the drop target can possibly be scrolled offscreen, the application needs to auto-scroll.

> **DESIGN PRINCIPLE**
>
> *Any scrollable drag-and-drop target must auto-scroll.*

In early implementations, auto-scrolling worked if you dragged outside the application's window. This had two fatal flaws. First, if the application filled the screen, how could you get the cursor outside the app? Second, if you wanted to drag the object to another application, how could the app tell the difference between that and the desire to auto-scroll?

Microsoft developed an intelligent solution to this problem. Basically, it begins auto-scrolling just *inside* the application's border instead of *outside* the border. As the drag cursor approaches the borders of the scrollable window—but is still inside it—a scroll in the direction of the drag is initiated. If the drag cursor comes within about 30 pixels of the bottom of the text area, Word begins to scroll the window's contents upward. If the drag cursor comes equally close to the top edge of the text area, Word scrolls down.

Thankfully, in recent times developers have commonly implemented a variable auto-scroll rate, as shown in Figure 18-19. The automatic scrolling increases in speed as the cursor gets closer to the window edge. For example, when the cursor is 30 pixels from the upper edge, the text scrolls down at one line per second. At 15 pixels, the text scrolls at two lines per second, and so on. This gives the user sufficient control over the auto-scroll to make it useful in a variety of situations.

Another important detail required by auto-scrolling is a time delay. If auto-scrolling begins as soon as the cursor enters the sensitive zone around the edges, it is too easy for a slow-moving user to inadvertently auto-scroll. To cure this, auto-scrolling should

begin only after the drag cursor has been in the auto-scroll zone for a reasonable amount of time—about a half-second.
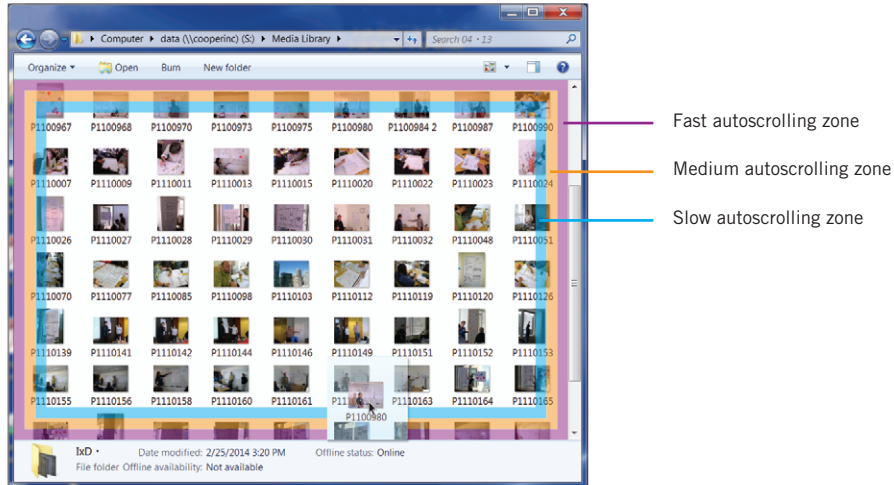
Figure 18-19: This image expresses the concept of variable-speed auto-scroll, as it could be applied to Windows Explorer. Unfortunately, auto-scroll moves at a single speed that is impossible to control. It would be better if the auto-scroll went faster the closer the cursor gets to the window's edge. (But it's also important to have a speed limit. Auto-scroll doesn't help anyone if it goes too fast.) To its credit, Microsoft's idea of auto-scrolling as the cursor approaches the *inside* edges of the enclosing scrollbox, rather than the outside, is clever indeed.

If the user drags the cursor completely outside Word's scrollable text window, no auto-scrolling occurs. Instead, the repositioning operation terminates in an application other than Word. For example, if the drag cursor goes outside Word and is positioned over PowerPoint, when the user releases the mouse button, the selection is pasted into the PowerPoint slide at the position indicated by the mouse. Furthermore, if the drag cursor moves within 3 or 4 millimeters of any of the borders of the PowerPoint Edit window, PowerPoint begins auto-scrolling in the appropriate direction. This is a convenient feature, because the confines of contemporary screens mean that we often find ourselves with a loaded drag cursor and no place to drop its contents.

Avoiding drag-and-drop twitchiness

When an object can be either selected or dragged, it is vital that the mouse be biased toward the selection operation. Because it is so difficult to click something without inadvertently moving the cursor a pixel or two, the frequent act of selecting something must not accidentally cause the application to misinterpret the action as the beginning of a drag-and-drop operation. Users rarely want to drag an object only one or two pixels. (And even in cases where they do, such as in drawing applications, it's useful to require a little extra effort to do so, to prevent frequent accidental repositioning.)

In the hardware world, controls like pushbuttons that have mechanical contacts can exhibit what engineers call *bounce*. This means that the switch's tiny metal contacts literally bounce when someone presses them. For electrical circuits like doorbells, the milliseconds the bounce takes aren't meaningful, but in modern electronics, those extra clicks can be significant. The circuitry backing up such switches has special logic to ignore extra transitions if they occur within a few milliseconds of the first one. This keeps your stereo from turning back off a thousandth of a second after you turned it on. This situation is analogous to the oversensitive mouse problem. The solution is to copy switch makers and *debounce* the mouse.

To avoid inadvertent repositioning, applications should establish a *drag threshold*. All mouse-movement messages that arrive after the mouse-down event are ignored unless the movement exceeds a small threshold amount, such as 3 pixels. This provides some protection against initiating an inadvertent drag operation. If the user can keep the mouse button within 3 pixels of the mouse-down point, the entire click action is interpreted as a selection command, and all tiny, spurious moves are ignored. As soon as the mouse moves beyond the 3-pixel threshold, the application can confidently change the operation into a drag, as shown in Figure 18-20. Whenever an object can be selected and dragged, the drag operation should be debounced.
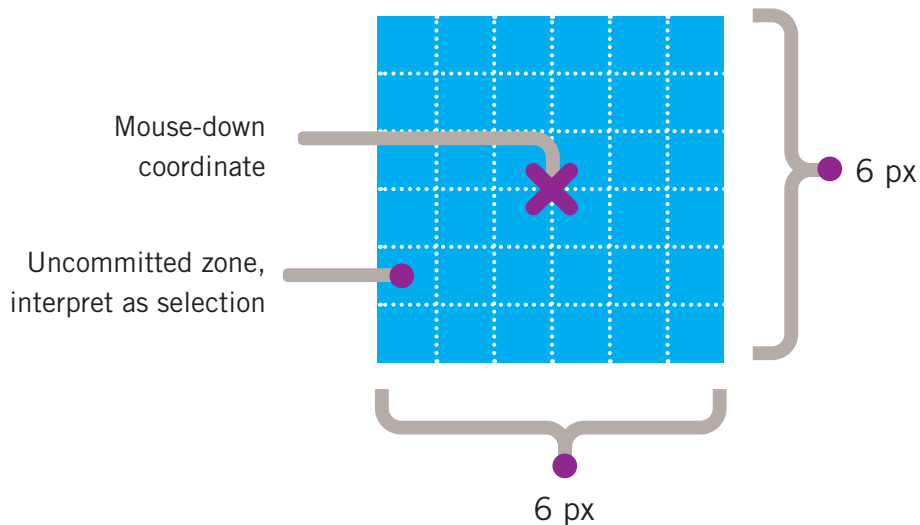
Figure 18-20: Any object that can be both selected and dragged must be debounced. When the user clicks the object, the action must be interpreted as a selection rather than a drag, even if the user accidentally moves the mouse a pixel or two between the click and the release. The application must ignore any mouse movement as long as it stays within the uncommitted zone, which extends 3 pixels in each direction. After the cursor moves more than 3 pixels from the mouse-down coordinate, the action changes to a drag, and the object is considered "in play." This is called a drag threshold.

Debounce all drags.

Some applications may require more-complex drag thresholds. Three-dimensional applications often require drag thresholds that enable movement in three projected axes on the screen. Another such example arose in the design of a report generator for one of our clients. The user could reposition columns on the report by dragging them horizontally. For example, he could put the First Name column to the left of the Last Name column by dragging it into position from anywhere in the column. This was by far the most frequently used drag-and-drop idiom. However, another infrequently used drag operation allowed the values in one column to be interspersed *vertically* with the values of another column—for example, an address field and a state field (see Figure 18-21).

Name	Address	City
1 Ginger Beef	342 Easton Lane	Waltham
2 C.U. Lator	339 Disk Drive	Borham
3 Justin Case	83 Elm	Alboin
4 Creighton Barrel	9348 N. Blenheim	Five Island
5 Dewey Decimal	1003 Water St.	Freeport

Name	Address
1 Ginger Beef	342 Easton Lane Waltham
2 C. U. Lator	339 Disk Drive Borham
3 Justin Case	83 Elm Alboin
4 Creighton Barrel	9348 N. Blenheim Five Island
5 Dewey Decimal	1003 Water St. Freeport

Figure 18-21: This report-generator application offered an interesting feature that enabled the contents of one column to be interspersed with the contents of another by dragging and dropping it. This direct-manipulation action conflicted with the more-frequent drag-and-drop action of reordering the columns (like moving City to the left of Address). We used a special two-axis drag threshold to accomplish this.

We wanted to follow the persona's mental model and enable him to drag the values of one column on top of the values of another to perform this stacking operation. However, this conflicted with the simple horizontal reordering of columns. We solved the problem by differentiating between horizontal drags and vertical drags. If the user dragged the column left or right, it meant that he was repositioning the column as a unit. If the user dragged the column up or down, it meant that he was interspersing the values of one column with the values of another.

Because the horizontal drag was the predominant user action, and vertical drags were rare, we biased the drag threshold toward the horizontal axis. Instead of a square uncommitted zone, we created the spool-shaped zone shown in Figure 18-22. Because the horizontal-motion threshold was set to 4 pixels, it didn't take a big movement to commit users to the normal horizontal move while still insulating users from an inadvertent vertical move. To commit to the far less frequent vertical move, the user had to move the cursor 8 pixels on the vertical axis without deviating more than 4 pixels left or right. That motion is quite natural and easily learned.
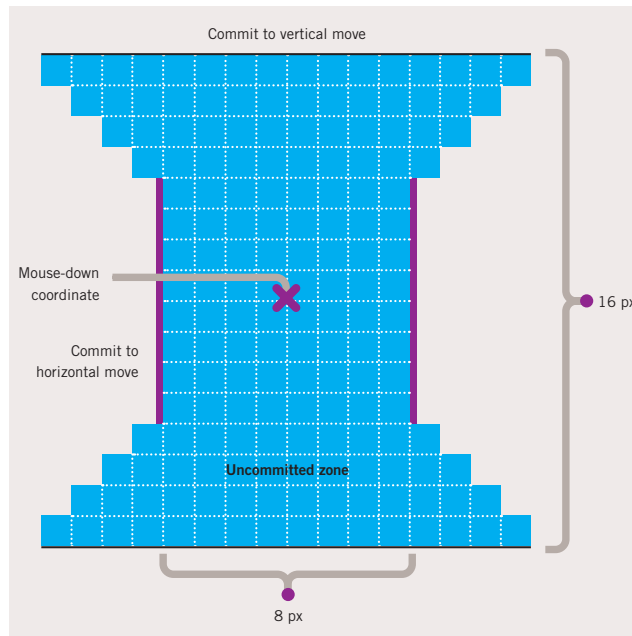
Figure 18-22: This spool-shaped drag threshold allowed a bias toward horizontal dragging in a client's application. Horizontal dragging was by far the most frequently used type of drag in this application. This drag threshold made it difficult for the user to inadvertently begin a vertical drag. However, if the user really wanted to drag vertically, a bold move either up or down would cause the application to commit to the vertical mode with a minimum of excise.

This axially asymmetric threshold can be used in other ways, too. Visio implements a similar idiom to differentiate between drawing a straight line and a curved line.

Fine scrolling

The weakness of the mouse as a precision pointing tool is readily apparent, particularly when dragging objects around in drawing applications. It is hard to drag something to the exact desired spot, especially when the screen resolution is 72 pixels per inch (or sometimes much more) and the mouse is running at a 6-to-1 ratio to the screen. To move the cursor 1 pixel, you must move the mouse about 1/500th of an inch.

This is solved by adding a *fine scrolling* function, whereby users can quickly shift into a mode that allows much finer resolution for mouse-based manipulation of objects. During a drag, if the user decides that he needs more-precise maneuvering, he can change the ratio of the mouse's movement to the object's movement on the screen. Any application that might demand precise alignment must offer a fine scrolling facility. This includes, at a minimum, all drawing and painting, presentation, and image-manipulation applications. This idiom has several variants. Commonly, using a modifier key while dragging puts the mouse into vernier mode, in which every 10 pixels of mouse movement are interpreted as a single pixel of object movement.

DESIGN
PRINCIPLE

Any program application that demands precise alignment must offer a vernier.

Another effective method is to make the arrow keys active during a drag operation. While holding down the mouse button, the user can manipulate the arrow keys to move the selection up, down, left, or right—1 pixel at a time. The drag operation is still terminated by releasing the mouse button. Many pixel-pushing applications like Adobe Photoshop let users move selections by single pixels with the arrow keys, and by 10x the standard amount when modified with a Shift key.

The problem with a vernier is that the simple act of releasing the mouse button can often cause the user's hand to shift a pixel or two. This causes the perfectly placed object to slip out of alignment just at the moment of acceptance. The solution is, upon receipt of the first vernier keystroke, to *desensitize* the mouse. You do so by making the mouse ignore all subsequent movements under some reasonable threshold, such as 5 pixels. This means that the user can make the initial gross movements with the mouse; and then make a final, precise placement with the arrow keys; and then release the mouse button without disturbing the placement. If the user wants to make additional gross movements after beginning the vernier, he simply moves the mouse beyond the threshold, and the system shifts back out of vernier mode.

If the arrow keys are not otherwise spoken for in the interface, as in a drawing application, they can be used to control vernier movement of the selected object. This means that the user does not have to hold down the mouse button. Adobe Illustrator and Photoshop

do this, as does PowerPoint. In PowerPoint, the arrow keys move the selected object one step on the grid—about 2 millimeters using the default grid settings. If you hold down the Alt key while using the arrow keys, the movement is 1 pixel per arrow keystroke.

Control manipulation

Controls are the fundamental building blocks of the modern graphical user interface. While we discuss the topic in detail in Chapter 21, in our current discussion of direct manipulation it is worth addressing the mouse interactions required by several controls.

Many controls, particularly menus, require the moderately difficult motion of a click and drag rather than a mere click. This direct-manipulation operation demands more of users because of its juxtaposition of fine motions with gross motions to click, drag, and then release the mouse button. Although menus are not used as frequently as toolbar controls, they are still used often, particularly by new and infrequent users. Thus, we find one of the more intractable conundrums of GUI design: The menu is the primary control for beginners, yet it is one of the more difficult controls to operate.

There is no solution to this problem other than to provide additional idioms to accomplish the same task. If a function is available from the menu, and it is one that will be used more than just rarely, be sure to provide idioms for invoking the function that don't require a click-and-drag operation, such as a toolbar button.

One nice feature in Windows, which Mac OS has also adopted, is the ability to work its menus with a series of single clicks rather than clicking and dragging. You click the menu, and it drops. You point to the desired item, click to select it, and close the menu. Microsoft further extended this idea by putting applications into a sort of *menu mode* as soon as you click any menu. In this mode, all the top-level menus in the application and all the items on those menus are active, just as though you were clicking and dragging. As you move the mouse around, each menu, in turn, drops without your having to click the mouse.

Modal tools and palettes

With *modal tools*, the user selects a tool from a tool palette, as discussed earlier in this chapter. The application's display area is then completely in that tool's mode: It does only that one tool's job. The cursor usually changes to indicate the active tool.

When the user clicks and drags with the tool on the drawing area, the tool does its thing. If the active tool is a spray can, for example, the application enters Spray Can mode, and it can only spray. The tool can be used repeatedly, spraying as much ink as the user wants until he clicks a different tool. If the user wants to use some other tool on the graphic, like an eraser, he must return to the toolbox and select the eraser tool. The application

then enters Eraser mode, and on the canvas, the cursor erases things only until the user chooses another tool. There is usually a selection-cursor tool on the palette to let the user return the cursor to a selection-oriented pointer, as in Adobe Photoshop, for example.

Modal tools work for tools that perform *actions* on drawings, such as an eraser, or for *shapes* that can be drawn, such as ellipses. The cursor can become an eraser tool and erase anything previously entered, or it can become an ellipse tool and draw any number of new ellipses. The mouse-down event anchors a corner or center of the shape (or its bounding box), the user drags to stretch the shape to the desired size and aspect, and the mouse-up event confirms the draw.

Modal tools are not bothersome in an application like Paint, where the number of drawing tools is very small. In a more advanced drawing application, such as Adobe Photoshop, however, the modality is disruptive. As users get more agile with the cursor and tools, the percentage of time and motion devoted to selecting and deselecting tools—the excise—increases dramatically. Modal tools are excellent idioms for introducing users to the range of features of such an application, but they usually don't scale well for intermediate users of more sophisticated applications. Luckily, Photoshop makes extensive use of keyboard commands for power users.

The difficulty of managing a modal tool application isn't caused by the modality as much as it is by the sheer quantity of tools. More precisely, the efficiencies break down when the quantity of tools in the user's working set gets too large. A working set of more than a handful of modal tools tends to become hard to manage. If the number of necessary tools in Adobe Illustrator could be reduced from 24 to eight, for example, its user interface problems might diminish below the threshold of user pain.

To compensate for the profusion of modal tools, products like Adobe Illustrator use modifier keys to modify the various modes. The Shift key is commonly used for constrained drags, but Illustrator adds many nonstandard modifier keys and uses them in nonstandard ways. For example, holding down the Alt key while dragging an object drags away a *copy* of that object, but the Alt key is also used to promote the selector tool from single-vertex selection to object selection. The distinction between these uses is subtle: If you click something and then hold down the Alt key, you drag away a copy of it. Alternatively, if you hold down the Alt key and *then* click something, you select all of it, rather than a single vertex of it. But then, to further confuse matters, you must *release* the Alt key, or you will drag away a copy of the entire object. To do something as simple as selecting an entire object and dragging it to a new position, you must hold down the Alt key, point to the object, click and hold down the mouse button without moving the mouse, release the Alt key, and then drag the object to the desired position! What were these people thinking?

Admittedly, the possible combinations are powerful, but they are hard to learn, hard to remember, and hard to use. If you are a graphic arts professional working with Illustrator

for eight hours a day, you can turn these shortcomings into benefits in the same way that a race car driver can turn the cantankerous behavior of a finely tuned automobile into an asset on the track. The casual user of Illustrator, however, is like an average driver behind the wheel of an IndyCar: way out of his league with a temperamental and unsuitable tool.

Charged cursor tools

With *charged cursor* tools, users again select a tool or shape from a palette. But this time, rather than switching permanently (until the user switches again) to the selected tool, the cursor becomes loaded—or *charged*—with a single instance of the selected object.

When the user clicks in the drawing area, an instance of the object is created on the screen at the mouse-up point. The charged cursor doesn't work too well for functions (even though Microsoft uses it ubiquitously for its Format Painter function), but it is nicely suited for graphic objects. PowerPoint, for example, uses it extensively. The user selects a rectangle from the graphics palette, and the cursor then becomes a modal rectangle tool charged with exactly one rectangle.

In many charged cursor applications like PowerPoint, the user cannot always deposit the object with a simple click. Instead, she must drag a bounding rectangle to determine the size of the deposited object. Some applications, like Visual Basic, allow either method. A single click of a charged cursor creates a single instance of the object in a default size. The new object is created in a state of selection, surrounded by *handles* (which we'll discuss in the section "Resizing and Reshaping") and ready for immediate precision reshaping and resizing. This dual mode, allowing either a single click for a default-sized object or dragging a rectangle for a custom-sized object, is certainly the most flexible and discoverable method that will satisfy the most users.

Sometimes charged-cursor applications forget to change the cursor's appearance. For example, although Visual Basic changes the cursor to crosshairs when it's charged, Delphi doesn't change it at all. If the cursor has assumed a modal behavior—if clicking it somewhere will create something—it is important that it visually indicate this state. A charged cursor also demands good cancel idioms. Otherwise, how do you harmlessly discharge the cursor? Pressing the Esc key is one widely used and effective discharge idiom.

2D object manipulation

Like controls, data objects on the screen, particularly 2D graphical objects in drawing and modeling applications, can be manipulated by clicking and dragging. Objects (other than icons, which were discussed earlier in this chapter) depend on click-and-drag motions for four main operations: repositioning, resizing, reshaping, and connecting.

Repositioning

Repositioning is the simple act of clicking an object and dragging it to a new location. The most significant design issue regarding repositioning is that it usurps the place of other direct-manipulation idioms. The repositioning function demands the click-and-drag action, making it unavailable for other purposes. This is not an issue for content in an application, since the direct manipulation is a likely intention of a drag and drop, but it can mean problems for objects in the interface.

The most common solution to this conflict is to dedicate a specific physical area of the object to the repositioning function. For example, you can reposition a window in Windows or on the Macintosh by clicking and dragging its title bar. The rest of the window is not pliant for repositioning, so the click-and-drag idiom is available for functions within the window, as you would expect. The only hints that the window can be dragged are its color and the slight dimensionality of the title bar, a subtle visual hint that is purely idiomatic. (Thankfully, the idiom is very effective.)

In general, however, you should provide more explicit visual hinting about an area's pliancy. For a title bar, you could use cursor hinting or a gripable texture as a pliancy hint.

Before you move an object, you must select it. This is why selection must take place on the mouse-down transition: The user can drag without having to first click and release an object to select it, and then click and drag it to reposition it. It feels so much more natural to simply click it and then drag it to where you want it in one easy motion.

This creates a problem for moving contiguous data. In Word, for example, Microsoft uses this clumsy click-wait-click operation to drag chunks of text. You must click and drag to select a section of text, wait a second or so and click, and then drag to move it. This is unfortunate, but there is no good alternative for contiguous selection. If Microsoft were willing to dispense with its modifier key idioms for extending the selection, those same modifier keys could be used to select a sentence and drag it in a single movement. But this still wouldn't solve the problem of selecting and moving arbitrary chunks of text.

When you do repositioning, a modifier key (such as Shift) is often used to constrain the drag to a single dimension (either horizontal or vertical). This type of drag is called a *constrained drag*. Constrained drags are extremely helpful in drawing applications, particularly when you draw neatly organized diagrams. The predominant motion of the first 5 or 10 pixels of the drag determines the angle of the drag. If the user begins dragging on a predominantly horizontal axis, for example, the drag henceforth is constrained to the horizontal axis. Some applications interpret constraints differently, letting users shift angles in mid-drag by dragging the mouse across a threshold.

Another way to assist users as they move objects around onscreen is by providing *guides*. In the most common implementations (such as in Adobe Illustrator), they are special lines that the user may place as references to be used when positioning objects. Commonly, the user may tell the application to "snap" to the guides. This means that if an object is dragged within a certain distance of the guide, the application will assume that it should be aligned directly with the guide. Typically this can be overridden with a keyboard nudge.

A novel and useful variation on this concept is OmniGraffle's Smart Guides. They provide dynamic visual feedback on and assistance with positioning objects. This is based on the (very reasonable) assumption that users are likely to want to align objects to each other and to create evenly spaced rows and columns of these aligned objects. Google's SketchUp (described in greater detail later in the chapter) provides similar help with three-dimensional sketches.

Resizing and reshaping

When it comes to windows in a GUI, there isn't really any functional difference between resizing and reshaping. The user can adjust a window's size and aspect ratio at the same time by dragging a control on a window's lower-right corner. It is also possible to drag any window edge. These interactions typically are supported by clear cursor hinting.

Such idioms are appropriate for resizing windows. But when the object to be resized is a graphical element (as in a drawing or modeling application), it is important to communicate clearly which object is selected and where the user must click to resize or reshape the object. A resizing idiom for graphical objects must be visually bold to differentiate itself from parts of the drawing, especially the object it controls. It also must not obscure the user's view of the object and the area around it. The resizer also must not obscure the resizing action.

A popular set of idioms accomplishes these goals; Shown in Figure 18-23, they are called *resize handles* (or, simply, *handles*). Handles serve double-duty because they can also indicate selection. This is a naturally symbiotic relationship, because an object usually must be selected to be resizable.

The handle centered on each side moves only that side, while the other sides remain motionless. The handles on the corners simultaneously move both the sides they touch, an interaction that is quite visually intuitive.

Handles tend to obscure the object they represent, so they don't make very good permanent controls. This is why we don't see them on top-level resizable windows. For that situation, frame or corner resizers are better idioms. If the selected object is larger than

the screen, the handles may not be visible. If they are hidden offscreen, not only are they unavailable for direct manipulation, but they are also useless as selection indicators.
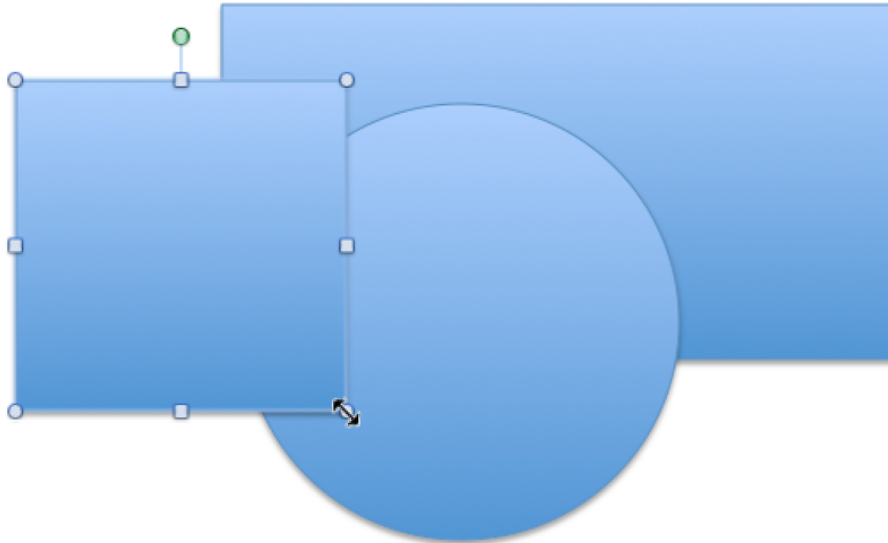
Figure 18-23: The selected object has eight handles, one at each corner and one centered on each side. The handles indicate selection and are a convenient idiom for resizing and reshaping the object. Handles are sometimes implemented with pixel inversion, but in a multicolor universe they can get lost in the clutter. These handles from Microsoft PowerPoint 2010 feature a small amount of dimensional rendering to help them stand out on the slide. Non-rectangular objects display their drag handles in a rectangular bounding box around the object.

As with dragging, a modifier key is often used to constrain the direction of a resize interaction. Another example of a constrained drag idiom, Shift is again used to force the resize to maintain the object's original aspect ratio. This can be quite useful. In some cases, it's also useful to constrain the resize to either a vertical, horizontal, or locked aspect ratio.

Notice that the assumption in this discussion of handles is that the object in question is rectangular or can be easily bounded by a rectangle. If the user is creating an organization chart, this may be fine, but what about reshaping more complex objects? A very powerful and useful variant of the resize handle is a *vertex handle*.

Many applications draw objects on the screen with *polylines*. A polyline is a graphics developer's term for a multisegment line defined by an array of vertices. If the last segment connects back to the first vertex, it is a closed form, and the polyline forms a polygon. When the object is selected, the application, rather than placing eight handles as

it does on a rectangle, places one handle on top of every vertex of the polyline. The user can then drag any vertex of the polyline independently and actually change one small aspect of the object's internal shape rather than affecting it as a whole. This is shown in Figure 18-24.

Figure 18-24: These are vertex handles, so named because each vertex of the polygon has one handle. The user can click and drag any handle to reshape the polygon, one segment at a time. This idiom is primarily useful for drawing applications.

Freeform objects in PowerPoint are rendered with polylines. If you click a freeform object, it is given a bounding rectangle with the standard eight handles. If you right-click the freeform object and choose Edit Points from the context menu, the bounding rectangle disappears, and vertex handles appear instead. It is important that both these idioms are available. The former is necessary to scale the image in proportion, and the latter is necessary to fine-tune the shape.

If the object in question is curved, rather than a collection of straight lines, the best mechanism to allow for reshaping is the Bézier handle. Like a vertex of a polyline, it expresses a point on the object, but it also expresses the shape of the curve at the point. Bézier curves require a good deal of skill to operate effectively and are probably best reserved for specialized drawing and modeling applications.

Connecting

A direct-manipulation idiom that can be very powerful in some applications is *connection*. The user clicks and drags from one object to another. But instead of dragging the first

object onto the second, a connecting line or arrow is drawn from the first object to the second.

If you use project management or organization chart applications, you are undoubtedly familiar with this idiom. For example, to connect one task box in a project manager's network diagram (often called a PERT chart) with another, you click and drag an arrow between them. In this case the direction of the connection is significant: The task where the mouse button went down is the *from* task, and the task where the mouse button is released is the *to* task.

As a connection is dragged between objects, it provides visual feedback in the form of *rubber-banding*: The arrow forms a line that extends from the first object to the current cursor position. The line is animated, following the movement of the cursor with one end while remaining anchored at its other end. As the user moves the cursor over connection candidates, cursor hinting should suggest that the two objects may be connected. After the user releases the mouse button over a valid target, the application draws a more permanent line or arrow between the two objects. In some applications, it also links the objects logically. As with drag and drop, it's vital to provide a convenient means of canceling the action, such as the Esc key or chord-clicking.

Connections can also be full-fledged objects themselves, with reshape handles and editable properties. This sort of implementation would mean connections could be independently selected, moved, and deleted as well. For applications where connections between objects need to contain information (such as in a project-planning application), it makes sense for connections to be first-class citizens.

Connection doesn't require as much cursor hinting as other idioms do because the rubber-banding effect is so clearly visible. However, it would be a big help in applications where objects are adjacent and connected logically, to show which currently pointed-to objects are valid targets for the arrow. In other words, if the user drags an arrow until it points to some icon or widget on the screen, how can he tell if that icon or widget can be connected to? The answer is to have the potential drop target visually hint at its pliancy. This hinting for potential targets can be quite subtle, or even eschewed completely when all objects in the application are equally valid targets for any connection. Target objects should always highlight, however, when a connection is dragged over them, to indicate willingness to accept the connection.

3D object manipulation

Working with precision on three-dimensional objects presents considerable interaction challenges for users equipped with 2D input devices and displays. Some of the most interesting research in UI design involves trying to develop better paradigms for 3D input

and control. So far, however, there seem to be no real revolutions—merely evolutions of 2D idioms extended into the world of 3D.

Most 3D applications are concerned with either precision drafting (for example, architectural CAD) or 3D animation. When models are being created, animation presents problems similar to those of drafting. An additional layer of complexity is added, however, in making these models move and change over time. Often, animators create models in specialized applications and then load these models into different animation tools.

There is such a depth of information about 3D-manipulation idioms that an entire chapter or even an entire book could be written about them. We will thus only briefly address some of the broader issues of 3D object manipulation.

Display issues and idioms

Perhaps the most significant issue in 3D interaction on a 2D screen is the lack of parallax, the binocular ability to perceive depth. Without resorting to expensive, esoteric goggle peripherals, designers are left with a small bag of tricks with which to conquer this problem. Another important issue is one of occlusion: near objects obscuring far objects. These navigational issues, along with some of the input issues discussed later, are probably a large part of the reason virtual reality hasn't yet become the GUI of the future.

Multiple viewpoints

Use of *multiple viewpoints* is perhaps the oldest method of dealing with both of these issues, but it is, in many ways, the least effective from an interaction standpoint. Nonetheless, most 3D modeling applications present multiple views on the screen, each displaying the same object or scene from a different angle. Typically, there is a top view, a front view, and a side view, each aligned on an absolute axis, which can be zoomed in or out. There is also usually a fourth view—an orthographic or perspective projection of the scene, the precise parameters of which the user can adjust. When these views are provided in completely separate windows, each with its own frame and controls, this idiom becomes quite cumbersome: Windows invariably overlap each other, getting in each others' way, and valuable screen real estate is squandered with repetitive controls and window frames. A better approach is to use a multipane window that permits one-, two-, three-, and four-pane configurations (the three-pane configuration has one big pane and two smaller panes). Configuration of these views should be as close to single-click actions as possible, using a toolbar or keyboard shortcut.

The shortcoming of multiple viewpoint displays is that they require users to look in several places at the same time to figure out an object's position. Forcing the user to locate something in a complex scene by looking at it from the top, side, and front, and then expecting him to triangulate in his head in real time, is a bit much to expect, even from

modeling whizzes. Nonetheless, multiple viewpoints *are* helpful for precisely aligning objects along a particular axis.

Baseline grids, depthcueing, shadows, and poles

Baseline grids, depthcueing, shadows, and poles are idioms that help get around some of the problems created by multiple viewpoints. The idea behind these idioms is to allow users to successfully perceive the location and movement of objects in a 3D scene projected in an orthographic or perspective view.

Baseline grids provide virtual floors and walls to a scene, one for each axis, which orient users. This is especially useful when (as is usually the case) the camera viewpoint can be freely rotated.

Depthcueing is a means by which objects deeper in the field of view appear dimmer. This effect typically is continuous, so even a single object's surface will exhibit depthcueing, giving useful clues about its size, shape, and extent. Depthcueing, when used on grids, helps disambiguate the orientation of the grid in the view.

One method used by some 3D applications to position objects is the idea of *shadows*— outlines of selected objects projected onto the grids as if a light is shining perpendicular to each grid. As the user moves the object in 3D space, she can track, by virtue of these shadows or silhouettes, how she is moving (or sizing) the object in each dimension.

Shadows work pretty well, but all those grids and shadows can get in the way visually. An alternative is the use of a single *floor grid* and a *pole*. Poles work in conjunction with a horizontally oriented grid. When the user selects an object, a vertical line extends from the center of the object to the grid. As she moves the object, the pole moves with it, but the pole remains vertical. The user can see where in 3D space she is moving the object by watching where the base of the pole moves on the surface of the grid (x- and y-axes) and also by watching the length and orientation of the pole in relation to the grid (z-axis).

Guidelines and other rich visual hints

The idioms described in the previous section are all examples of rich visual modeless feedback, which we will discuss in detail in Chapter 15. However, for some applications, lots of grids and poles may be overkill. For example, Google's SketchUp is an architectural sketching application that lets users lay down their own drafting lines using a tape measure and protractor. As they draw their sketches, they get color-coded hinting that keeps them oriented to the right axes. Users can also turn on a blue-gradient sky and a ground color to help keep them oriented. Because the application is focused on architectural sketching, not general-purpose 3D modeling or animation, the designers were able to pull off a spare, powerful, and simple interface that is easy to both learn and use (see Figure 18-25).
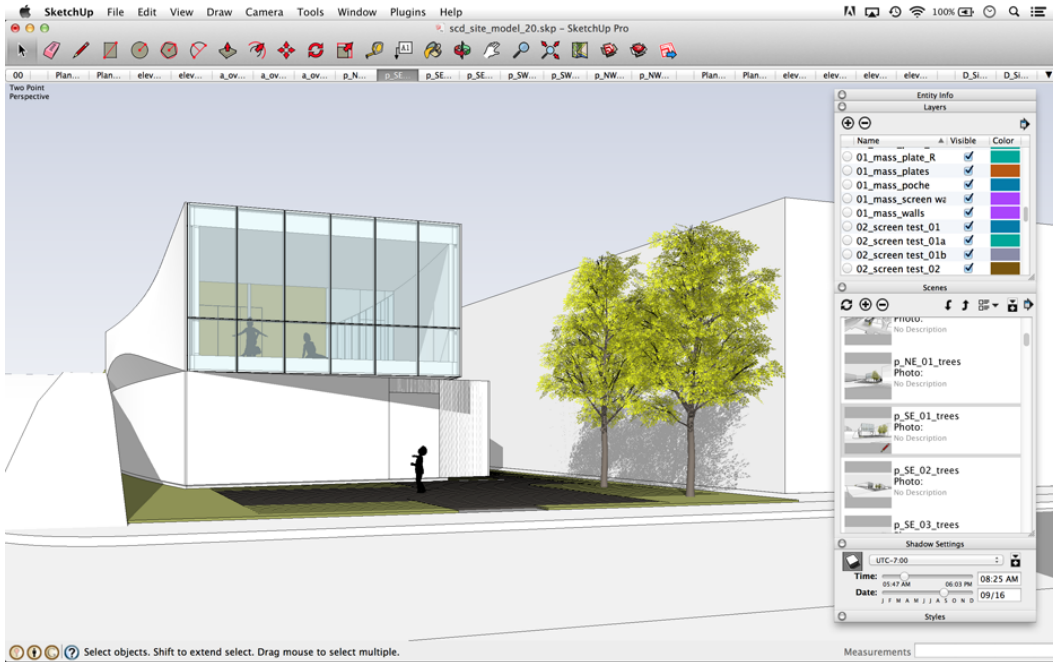
Figure 18-25: SketchUp is a gem of an application that combines powerful 3D architectural sketching capability with smooth interaction, rich feedback, and a manageable set of design tools. Users can set sky color and real-world shadows according to location, orientation, and time of day and year. These help not only in presentation but also in orienting users. Users also can lay down 3D grid and measurement guides just as in a 2D sketching application. Camera rotate and zoom functions are cleverly mapped to the mouse scroll wheel, allowing fluid access while using other tools. ToolTips provide textual hints that help users draw lines and align objects.

Wireframes and bounding boxes

Wireframes and *bounding boxes* solve problems of object visibility. In the days of slower processors, all objects needed to be represented as wireframes, because computers weren't fast enough to render solid surfaces in real time. It is fairly common these days for modeling applications to render a rough surface for selected objects while leaving unselected objects as wireframes. Transparency would also work, but it can be very processor-intensive. In highly complex scenes, it is sometimes necessary or desirable, but not ideal, to render only the bounding boxes of unselected objects.

Input issues and idioms

3D applications use many idioms such as drag handles and vertex handles that have been adapted from 2D to 3D. However, some special issues surround 3D input.

Drag thresholds

One of the fundamental problems with direct manipulation in a 2D projection of a 3D scene is the problem of translating 2D cursor motions in the plane of the screen into a more meaningful movement in the virtual 3D space.

In a 3D projection, a different kind of drag threshold is required to differentiate between movement along three, not just two, axes. Typically, up and down mouse movements translate into movement along one axis, whereas 45-degree-angle drags are used for each of the other two axes. SketchUp provides color-coded hinting in the form of dotted lines when the user drags parallel to a particular axis, and it also hints with ToolTips. In a 3D environment, rich feedback in the form of a cursor and other types of hinting becomes a necessity.

The picking problem

The other significant problem in 3D manipulation is known as the *picking problem*. Because objects need to be in wireframe or need to be otherwise transparent when assembling scenes, it becomes difficult to know which of many overlapping items the user wants to select when she mouses over it. Locate highlighting can help but is insufficient because the object may be completely occluded by others. Group selection is even trickier.

Many 3D applications resort to less-direct techniques, such as an object list or object hierarchy that users can select from outside of the 3D view. Although this kind of interaction has its uses, there are more direct approaches.

One obvious approach is to let users type in or speak the name of the object they wish to select. If there's only one cube in the scene, it's easy for the system to distinguish. Another might be by attribute. "Select the green shape." If the user has bothered to name objects in the interface, their unique ID could also be used. Since most 3D manipulations are performed with the mouse, though, this forces a bit of mode-shifting that is less than ideal. There are more mouse-centric ways to handle it.

For example, hovering over part of a scene could open a ToolTip-like menu that lets users select one or more overlapping objects. (This menu would be unnecessary in the simple case of one unambiguous object.) If individual facets, vertices, or edges can be selected, each should hint at its pliancy as the mouse rolls over it.

Although it doesn't address the issue directly, a smooth and simple way to navigate around a scene can also ameliorate the picking problem. SketchUp has mapped both zoom and *orbit* functions to the mouse scroll wheel. Spin the wheel to zoom in toward or away from the central zero point in 3D space. Press and hold the wheel to switch from

whatever tool you are using to orbit mode, which allows the camera to circle around the central axes in any direction. This fluid navigation makes manipulating an architectural model almost as easy as rotating it in your hand.

Object rotation, camera movement, rotation, and zoom

One more issue specific to 3D applications is the number of spatial manipulation functions that can be performed. Objects can be repositioned, resized, and reshaped in three axes. They also can be rotated in three axes. Beyond this, the camera viewpoint can be rotated in place or revolved around a focal point, also in three axes. Finally, the camera's field of view can be zoomed in and out.

Not only does this mean that assignment of modifier keys and keyboard shortcuts is critical in 3D applications, but another problem also occurs: It can be difficult to tell the difference between camera transformations and object transformations by looking at a camera viewpoint, even though the actual difference between the two can be quite significant. One way around this problem is to include a thumbnail, absolute view of the scene in a corner of the screen. It could be enlarged or reduced as needed and could provide a reality check and global navigation method in case the user gets lost in space. (Note that this kind of thumbnail view is useful for navigating large 2D diagrams as well.)

DESIGNING FOR MOBILE AND OTHER DEVICES

The mobile device user experience changed forever in June of 2007, when Apple introduced the iPhone. Almost overnight, the definition of what it meant to be a mobile information device experienced a radical reboot. Before the iPhone's introduction, the mobile user experience meant tiny hardware keyboards on the device surface or hidden within a sliding drawer. It also meant small, clumsy, resistive mono-touchscreens that more often than not required a stylus to operate effectively without resorting to a miniature five-way D-pad that was equally cumbersome to use.

The iPhone replaced this mess of a user experience with the following:

- A giant, high-resolution, multi-touchscreen, an OS that specified on-screen controls big enough for fingers to use successfully

- A set of now-iconic gestural idioms that were relatively easy to discover and learn

- A set of sensors delivering contextual information about orientation, location, ambient light, and movement that added an extraordinary level of intelligence to a new generation of mobile apps

Scarcely more than a year later, Google introduced its own multi-touch mobile operating system, Android, borrowing many of its gestural and navigational idioms from its iOS competitor. Although it took Google several years of iteration to begin matching iOS from an aesthetic and user experience refinement perspective, it has become

the Windows of the mobile world, taking the majority share of the smartphone market. (Windows Phone OS, ironically, was quite late to the game; it lags severely in market share.) As of this writing, the basic mobile user experience is remarkably uniform across "smart" mobile platforms. More than 90 percent of these devices sport large, gestural, multi-touchscreens with similar idioms, similar sensor-laden hardware, and even (as of iOS 7) converging "flat" visual styles.

Almost exactly the same story can be told regarding the release of the iPad. This device rewrote the story on tablet devices, a market that Microsoft and many others had repeatedly tried and abandoned. However, the success of the iPhone practically ensured that the iPad (despite early naysaying from the desktop computing world) would be an instant success.

Today, iPad, Android, and Microsoft multi-touch tablet sales are seriously eroding the sales of low-end laptop computers, and this trend will most likely continue. For most people, a computing device that turns on instantly, saves its state automatically on shut-down, manages its own software updates in the background, installs from the cloud, eliminates window management excise, and allows direct interaction with fingertips is a rather dramatic improvement over the complexities of desktop software and pointer-based input. It isn't difficult to imagine what this means for the future of desktop and even laptop computers.

The majority of this chapter describes some of the most important design concerns and design patterns for phone and tablet format mobile devices. We'll also briefly discuss other device platform interfaces at the end of the chapter, including public kiosk, device, and automotive interfaces.

Anatomy of a Mobile App

While the posture of desktop applications is most often sovereign (see Chapter 9), mobile apps, by contrast, are by their very nature *transient*. The on-the-go and highly context-driven nature of the majority of mobile apps (games perhaps being the exception, but the interaction design of games in general is a unique topic in itself) dictates a transient stance, especially on handheld mobile devices. The fact that these transient apps take up their host device's entire screen makes them no less transient. Transience here is dictated by the character of the user's interaction with the app: brief, intermittent, and focused on particular tasks.

> **DESIGN PRINCIPLE**
>
> *Most mobile apps have transient posture.*

The other major factor that contributes to the transient nature of mobile apps is the physical form factor of the host device. Phone-sized screens that support multi-touch interactions require onscreen objects to be large enough that they can be activated easily with fingers, without the user accidentally triggering other interactions while doing so. Tablet-sized screens have a bit more breathing room but still need finger-scale controls.

These two factors lead to an information and control density on mobile screens similar to the information and control density of dialogs on the desktop. While high-resolution display technology does help allow for detailed information graphics and crisp text on mobile devices, the number and spacing of individual objects on the screen remain fairly limited if usability and readability are to be maintained. The alternative solution of zooming (see Chapter 12), while technically possible, would only add a layer of complexity and confusion, since logical zoom—itself a bit problematic already—is the typical (if awkward) method of navigating between apps on many mobile platforms.

Mobile form factors

While it's safe to say that mobile apps are almost always transient, the form factor of the mobile device has a significant effect on the navigation, the layout, and even the behavioral strategies and patterns employed.

Modern multi-touch mobile devices fall primarily into three form factor categories:

- **Handhelds** consist of phones and media devices like the iPod Touch. They are characterized by tall, narrow (typically 16:9 aspect ratio) screens that are 4 to 6 inches diagonally and are used most frequently in portrait orientation.

- **Tablets** consist of devices sporting 9-to-10-inch diagonal screens. (Apple's tablets are 4:3; Google and Microsoft's are 16:9.) Android and Windows tablets seem biased toward landscape use in their design, while Apple tablets seem to be used more frequently in both orientations.

- **Mini-tablets** consist of devices sporting screens that are 7 to 8 inches diagonally. Like their larger cousins, they have either a 4:3 aspect ratio (Apple) or 16:9 aspect ratio (Google, Microsoft).

The next section focuses on the basic structural patterns for each of these form factors, the building blocks of which we'll discuss later in this chapter, as well as in Chapter 21.

Handheld format apps

Mobile touchscreen operating systems thankfully eschew the complexities of window management, opting instead for full-screen applications that make much better use of the limited available real estate. This decision dates back to the earliest handheld

touchscreen devices, including Apple's Newton and Palm's very popular PalmPilot. It continues to make sense even though modern mobile displays are many times the resolution of those now crude-looking screens.

Modern handheld-format devices also continue to use some of the same basic layout patterns employed in these early systems—vertical stacks of UI elements, including lists, grids, bars, and drawers. Newer structural patterns made possible by high-performance, high-resolution graphics and multi-touch screens—such as carousels, swimlanes, and cards—are now widely recognized mobile idioms.

Stacks

Stacks are perhaps the primary pattern used by most non-game mobile apps—especially on handheld devices. The tall and narrow form factor of smartphones and other handheld mobile devices dictates a list-like display for most types of content or control. The main exception is icons and thumbnails; more about these below. Stacks are vertically organized structures with a content area, usually arranged in a list or grid, with a top and/or bottom bar for navigating content and accessing functions. Most iOS, Android, and Windows Phone apps follow this top-level pattern, as shown in Figure 19-1.
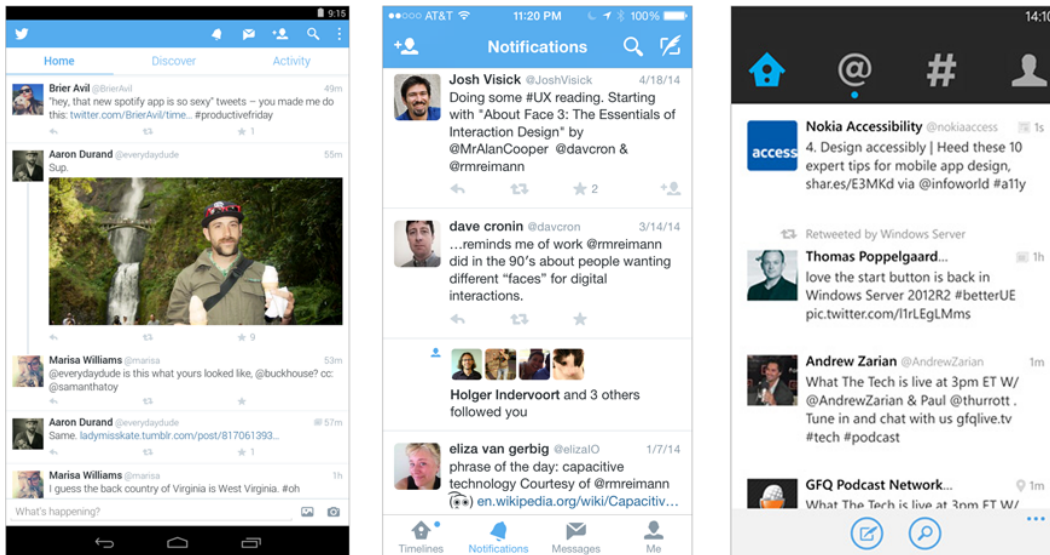
Figure 19-1: Typical mobile apps use a stack layout pattern including content, control, and navigation elements.

Screen carousels

Screen carousels are an alternative top-level pattern that is most appropriate for a dashboard-like display that has multiple instances or variants between which the user can quickly navigate via a swipe gesture to the left or right. The classic example of this pattern is the iOS Weather app, shown in Figure 19-2. The user swipes between identically laid-out cards or screens that, in the case of the Weather app, represent different locations. The few interactions on a carousel screen occur in place on the card; there usually is no drill-down navigation, as you typically see in the Stacks pattern. Carousels may or may not have top or bottom bars associated with them, but they usually do have a page marker widget that shows the user's place in the carousel content. Carousels often don't provide circular flow, but rather disallow further swiping at the far left and right. In most cases, there's no reason not to make it circular, which makes navigating between screens much easier.
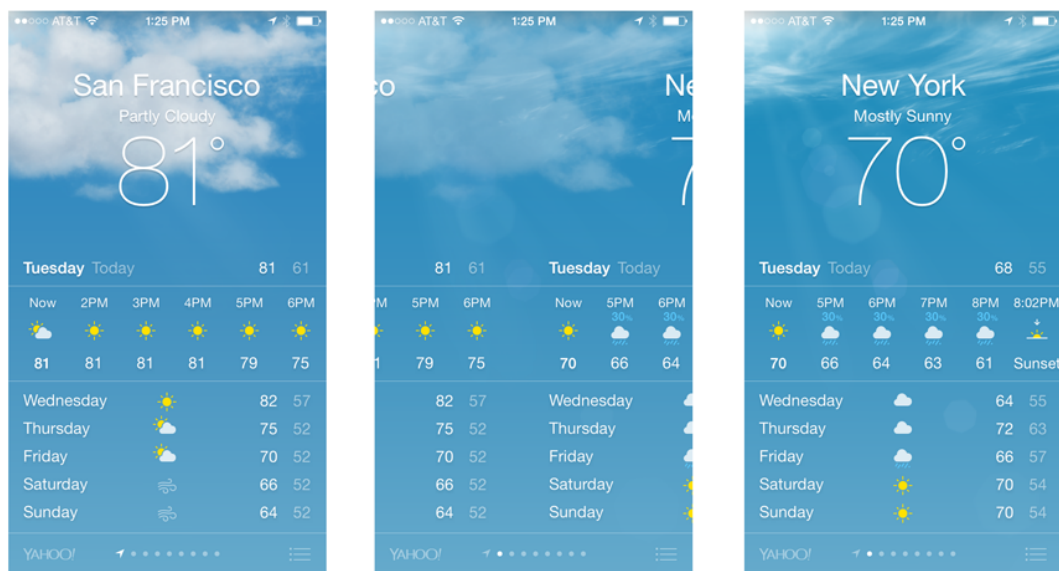
Figure 19-2: The iOS Weather app is the classic example of a screen carousel pattern, where you can navigate between several instances of self-contained dashboard-like screens by swiping left or right. A place marker widget in the bottom bar shows the user his current position in the sequence of screens. This implementation of the pattern doesn't wrap around from the end to the beginning of the carousel, making navigation across the set harder than it needs to be.

Orientation and layout

Most modern mobile devices can detect their screen orientation (portrait or landscape), which means that the app can dynamically rearrange its layout to better suit the current orientation. The majority of apps, however, stick with portrait orientation even when rotated. For list- and grid-based content browsing (more on these topics in the section

"Browse Controls"), assuming portrait orientation is a good bet because users usually operate the phone one-handed in portrait orientation.

However, for applications such as photo or video capture and editing, it makes sense to allow rotation to landscape orientation, since the medium itself can be in that orientation. For these sorts of apps, iconic controls make the most sense, since they can simply be rotated right along with the screen and thus minimize user disorientation (see Figure 19-3). However, this means that extra care must be taken to ensure that users can easily figure out what the controls mean.
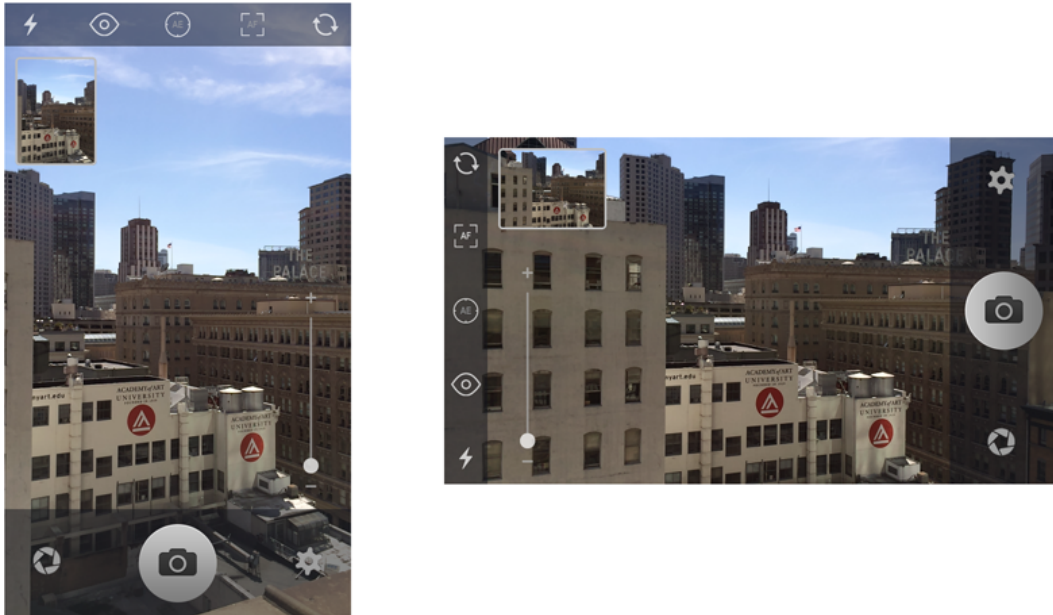
Figure 19-3: The Slow Shutter app on iOS does a great job of making a smooth transition from portrait to landscape. iOS's native Camera app also allows this sort of transition. But since it uses a scrolling selection bar containing text labels, the result is difficult-to-read, rotated text when the app is in landscape orientation.

Tablet format apps

Tablet format apps have considerably more breathing room than handheld-format apps as far as screen real estate is concerned. The iPad's 4:3 aspect ratio and large screen size ensures plenty of room for navigational and functional controls, but Windows and Android tablets also manage quite serviceably with the movie-like 16:9 aspect ratios.

Stacks and index panes

Like handheld format mobile apps, tablet format apps also rely on the stack pattern, vertically stacking a primary area and one or more tab, navigation, and action bars. However, the extra real estate available on a tablet also allows the addition of one or more supporting panes, should the app require it. Typically the additional pane is an index pane (see Chapter 18) that lists content items, such as your e-mail inbox or search results, the current selection of which is displayed in detail in the main content pane. This is a good use of display real estate, because it eliminates one level of drill down and allows users to quickly navigate and inspect a long list of content.

These additional *index panes* can themselves have navigation and functions associated with them, which are housed in bars at the top or bottom of the pane. Frequently, the list of objects in the index pane can come from several sources such as e-mail (see Figure 19-4). This requires either tabbed or drill-down approaches to navigation. We discuss these in more detail later in the chapter. Search and filter widgets are also common controls in tablet index panes.

In portrait mode, index panes typically are launched from a button and overlap the main content area. Unless the content of your app's index pane is particularly narrow, you will probably want to opt for this approach (see Figure 19-4). Non-overlapping panes provide a superior interaction, even in portrait, if they are narrow enough to fit.
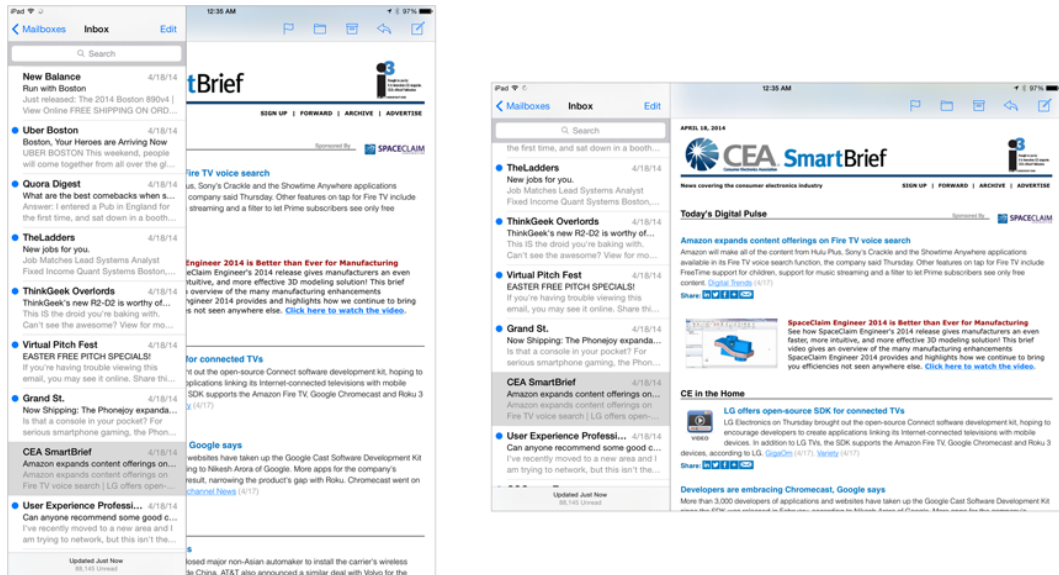
Figure 19-4: iOS's iPad Mail app presents a navigable index pane containing mail folders and their contents. In portrait mode, the pane is launched from a button on the left of the app's nav bar and overlaps the rest of the screen until it is dismissed. In landscape mode, the pane is permanently placed adjacent to the detailed content pane.

When rotated to landscape, the standard pattern calls for the overlapping pane to become a permanent adjacent pane.

Pop-up control panels

Tablet screens are large enough to support *pop-up panels* that don't overlay the entire screen and that can replace navigation to a full-screen control panel screen, as is usually required in handheld format apps. These pop-ups, when used judiciously, can improve task flow by retaining the context of the background screen and not feel as much to the user like going to a different room (see Chapter 18).

Pop-up panels are different from dialogs, in that they are attached to a particular control or content object and are used to make changes to parameters associated with that object. This association typically is shown via a speech balloon caret that extends from the pop-up canvas to the control it is associated with, as shown in Figure 19-5.

Figure 19-5: The Procreate digital painting app on iOS makes extensive use of pop-up control panels as a means of configuring the drawing tools in the app's tool bar. These pop-up panels show their connection to the tool via a speech-balloon-like caret that emerges from the otherwise rectangular panel.

Orientation-based layout

More so than on handheld format apps, tablet apps need to be concerned about orientation. On tablet apps, rotating controls in place usually is an insufficient or undesirable approach. Instead, tabs, navigation, and tool bars need to reorient themselves on the screen in a sensible way by relocating to the top or sides of the screen as appropriate. Overlapping panes that were available from a button need to be laid out adjacent to the main content pane (see Figure 19-4). This approach makes sense for simple apps. But more complex apps, or those dedicated to activities that have a heavy bias toward one orientation—such as a streaming video app (landscape), e-reader (portrait), or any authoring tool that relies on a fixed layout of complex controls—may select to support only one orientation. The next two sections elaborate on two of these cases.

Mobile versus desktop-like layout

The high-resolution displays on modern touchscreen tablets rival those of many laptop and even desktop displays, shrunk to fit a 10-inch diagonal screen. It might be tempting to treat your tablet app like a shrunken, touch-enabled version of your desktop app. In most cases, however, this isn't a good idea. For media browsing and other search, browse, and view/purchase types of apps, the approaches outlined in this chapter are appropriate.

However, due to the complexity of productivity and creative authoring apps that seek to replace similar desktop apps, there is more of a case for adopting more desktop-like tool bars and panes. Audio and video production apps in particular seem well-suited to a more desktop-like approach, as shown in Figure 19-6. Here relatively dense control layouts, multiple panes, complex tool bars and control panels, large pop-up panels or drawers, and drag-and-drop idioms make sense.
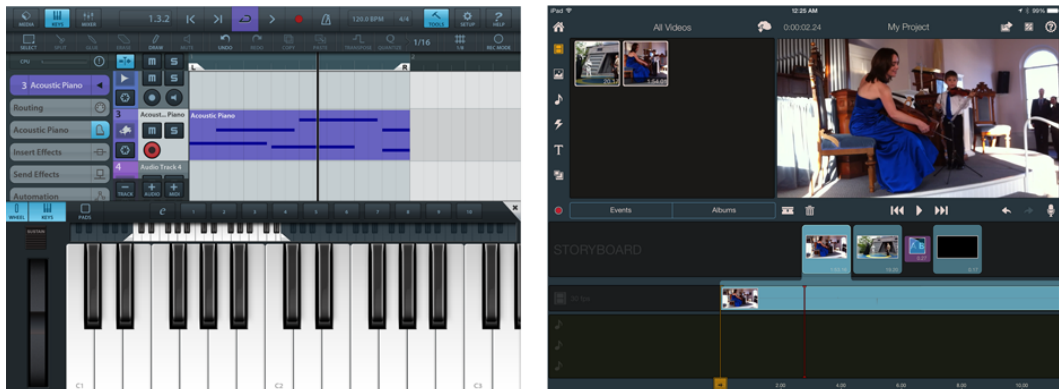
Figure 19-6: Steinberg's Cubasis and Corel's Pinnacle Studio are examples of media production apps that are well suited to a more complex layout that more closely resembles desktop apps.

If your app is in this category, keep the following principles in mind:

- Make sure that tool bar, control panel, and menu items have areas that register taps (known as *hit areas*) and inter-item spacing properly scaled for finger use.

- Drag and drop is prone to accidental drops on a touchscreen, so either avoid it or be forgiving of them.

- Pop-up panels should point to where they came from when possible and have clearly labeled headers.

- Pay close attention to function hierarchy so as to keep workflow as linear as possible. Put the user on a single path to accomplish a task whenever possible.

- Apps with complex layouts should in most cases choose an orientation and stick to it, rather than trying to support both portrait and landscape. Consider the alternate orientation an opportunity for a completely different display.

Hardware-like control layout

For certain apps, especially those in the domain of music production, interfaces that resemble hardware-based control surfaces appeal to users in that domain. While such interfaces can be quite awkward on the desktop due to the necessity of operating faux-hardware controls with a mouse or trackpad, the introduction of a multi-touch input surface on the display changes this equation substantially. Designers should still take into account fingertip use for hit areas and spacing. They should allow horizontal or vertical drag gestures (and be consistent about their use) to operate rotary controls in addition to circular drag gestures. Also remember not to let your key interactions be artificially limited by hardware metaphors. A knob or slider might make sense for setting volume or mixing tracks, but using direct manipulation for activities such as shaping and scrubbing audio waveforms can provide a whole new level of richness to the creative process, as shown in Figure 19-7.

Mini-tablet format apps

Mini-tablets such as Google's Nexus 7 and the Kindle Fire are popular, inexpensive mobile devices that handily fit into a large purse or pocket, making them popular with consumers. From a user experience perspective, however, their combination of a narrow 16:9 aspect ratio, support for both screen orientations, and a small size represent challenges for a designer of touch-based experience. There simply isn't as much room for finger-sized controls as on a full-sized tablet. But at the same time, there's a bit too much room for apps designed for phones to look aesthetically well proportioned, especially using standard OS widgetry.

Figure 19-7: Positive Grid's Final Touch app provides pro-quality audio mastering on the iPad. While making extensive use of hardware control metaphors, its smart layout and workflow, along with judicious use of direct-manipulation idioms in combination with the hardware-like controls, make it both extremely powerful and easy to use.

Navigation and layout strategies employed by handheld and tablet format apps will work for mini-tablets, with some caveats:

- **Adjacent panes**—Generally not a good idea on full-sized tablets in portrait orientation, adjacent panes are usually far too cramped to consider on mini-tablets. In landscape, at most two adjacent panes (and perhaps a vertical tab bar) can be supported. In portrait in particular, overlapping pop-up panes and drawers make more sense. Drawers are discussed in the next section.

- **Tool bars**—In portrait view, these can feel distant from the action due to the tall, narrow form factor and increased screen size over handhelds. In landscape orientation, tool bars stacked with navigation bars leave little vertical space for content. Vertical tool bars may sometimes make more sense on mini-tablets. Tool bars are discussed in the next section.

- **Lists**—Single-column lists tend to look out of proportion on mini-tablets, even in portrait orientation. Grid, swimlane, and card approaches tend to work better for the

user's sense of flow in both orientations. If your content is truly list-based, consider using vertical tabs or tool bars in portrait and adjacent index and detail panes in landscape. Each of these idioms is discussed in more detail in the next section.

- **Pop-up versus full-screen dialogs**—Mini-tablets are big enough that using phone-style full-screen idioms for menus and dialogs won't work; these should be implemented as pop-up dialogs as on full-size tablets. Pop-up control panels for tool bar tools will also work, but these may take up the majority of the screen when open.

Mobile Navigation, Content, and Control Idioms

Mobile applications share many controls with desktop and web applications, as discussed at length in Chapter 21. However, due to the unique form factor and multi-touch input technology employed by the majority of modern mobile devices, they have evolved a unique set of idioms especially suited to mobile app use.

We'll enumerate the most common and important of these in this section.

Browse controls

Most mobile apps are optimized for *browsing*. Whether it's music, videos, social networking updates, restaurant reviews, e-mail, shopping, or search results, we do an awful lot of casual surveying in our mobile apps. Due to the limitations of the form factor and input options, it is much easier on mobile devices to browse and select content than to input data. Given this situation, it's unsurprising that mobile apps have developed a rich set of patterns around browsing through content.

Lists

Lists are the most frequently used pattern for organizing content on handheld format touchscreen devices. List content often includes line items or blocks of text, controls (such as check boxes or buttons) and their labels, and image or video thumbnails.

Tapping a content item in a list typically drills down a level in the hierarchy, revealing either the content or the next level of grouping. Sometimes tapping a list item may also launch a modal pop-up or screen that provides a set of options for controlling the item, or it may navigate to a detailed view of the item itself.

As we'll discuss in a minute, list views often work in conjunction with tab bars to provide access to multiple screens of content, each in its separately maintained list. Apple's Music app is a good example of this, with lists of albums, artists, and songs available on different tabs, each with its own (slightly different, but related) drill-down hierarchies (see Figure 19-8).
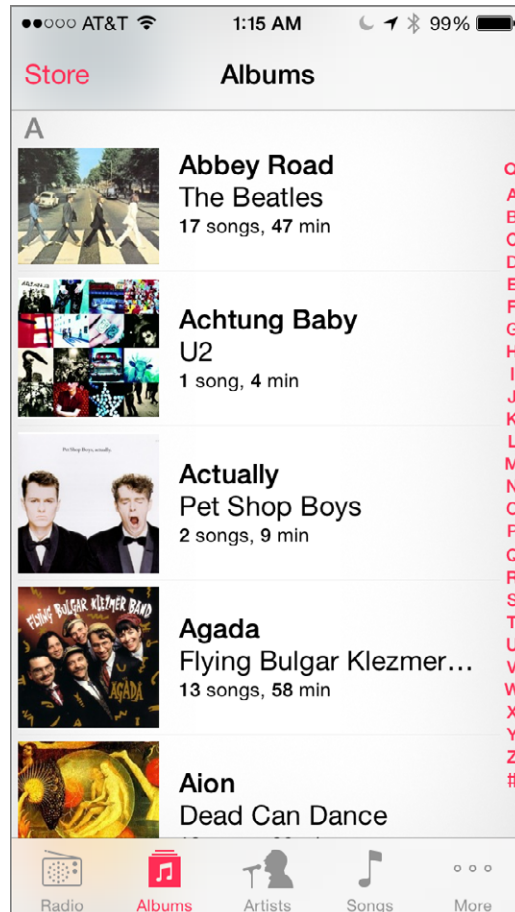
Figure 19-8: The iOS Music app has tabbed lists of albums, artists, and songs, among others. Navigating between lists is accomplished via a bottom tab bar.

Lists can either be finite in length or allow *infinite scrolling*. This kind of scrolling presents an initial subset of items from a very large set (such as web search results) and then presents an additional block of results each time the user reaches the bottom of the list. While infinite scrolling is a necessary compromise due to limited computing resources, it is a reasonably elegant solution, as long as the incremental load time can be kept under a second or so.

Grids

Grids are used to organize content such as apps, thumbnails, and function icons into regular rows and columns. The most obvious example of this is the home screen of the iPhone, with its editable grid of app icons. Android supports a similar interface. Microsoft has taken the idea of the app icon grid and transformed it into the more innovative Start screen grid. It mixes apps and notifications in an aesthetically pleasing and useful way, as shown in Figure 19-9.
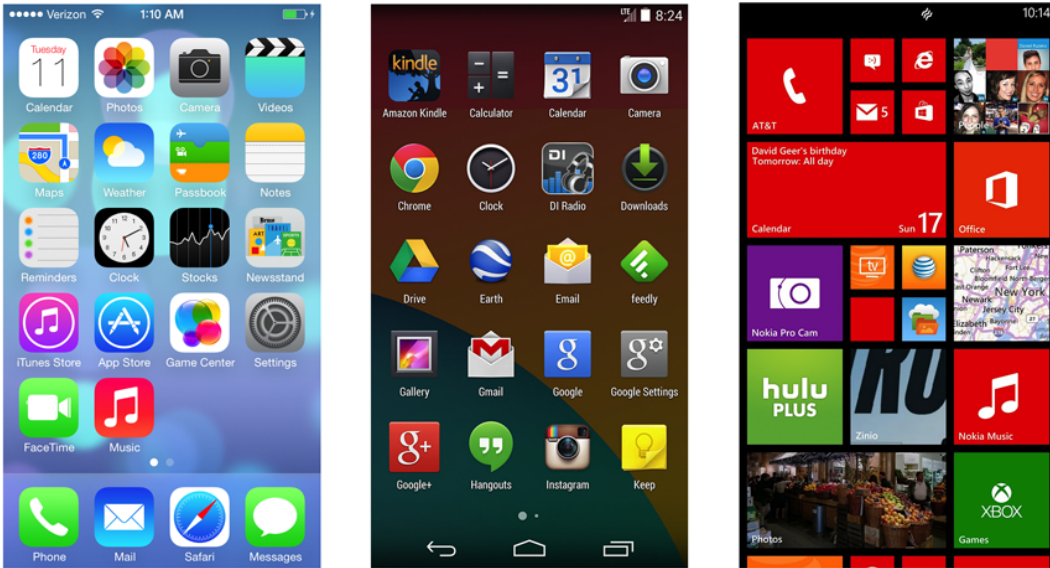
Figure 19-9: iOS and Android home screens use a similar app grid, both of which are derived from the original Palm Pilot. Microsoft, on the other hand, evolved its Zune interface into the Metro UI, with its unique Start screen grid that seamlessly—and beautifully—combines apps and notifications.

Within an app, grid views (also called gallery views) often are used to present media objects. These include photos, videos, and music albums (with cover art), or small, encapsulated cards (more on this later) containing image, text, and sometimes button or link elements. One challenge with presenting grids of content objects is making sure users understand how to navigate them. The iPhone's home screen uses horizontal swipes to navigate between grid "pages." Most apps that use grids as a primary navigation and selection mechanism, such as Rdio (see Figure 19-10), use pageless, and sometimes infinite, vertical scrolling to expose more grid objects (albums in this case). The direction of scrolling is nicely disambiguated by sizing the album art so that the bottom-most visible row is partly cut off. This provides the necessary visual hint that a vertical swipe up will reveal more choices.
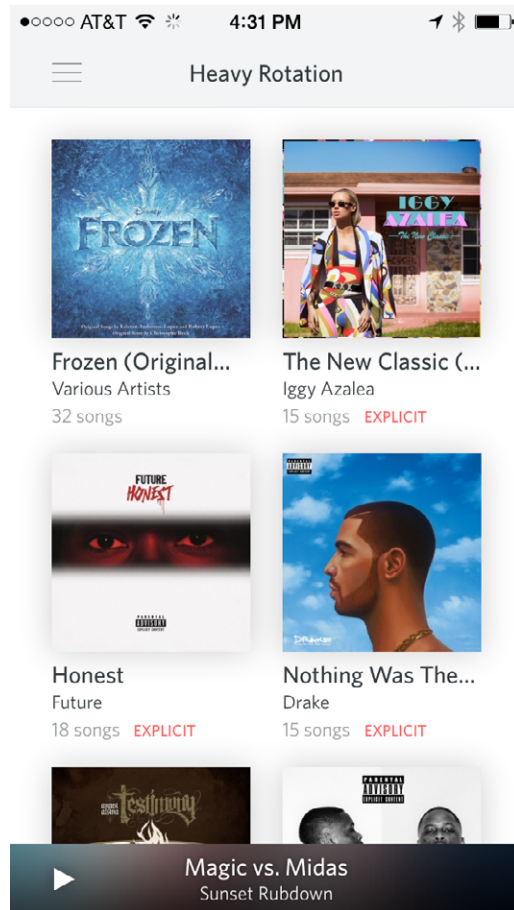
Figure 19-10: The Rdio streaming music app using a two-column scrolling grid to display album choices. The bottom-most visible row is cut off, which hints that scrolling is vertical.

Apple's Photos app, shown in Figure 19-11, uses a much tighter four-column grid for the Camera Roll, which also scrolls vertically.

Grids also can scroll horizontally, as in Apple's Music app when the iPhone is rotated to landscape orientation, as shown in Figure 19-12.

Figure 19-11: The Camera Roll in Apple's Photos app uses a tighter grid with four vertically scrolling columns on the iPhone.

It might be tempting to allow zooming in and out on the grid via a pinch gesture, but generally this is not a good idea, especially in the narrow portrait orientation of the handheld form factor. Issues quickly arise concerning the legibility and hit area of the icons or thumbnails, as well as column width of text labels and metadata.

As with lists, tapping a content item in a grid typically drills down into a hierarchy, revealing either another grid or list of content items or controls. Or it launches a modal pop-up that provides a set of options for controlling the item. Or it opens a detailed view of the item itself, as shown in Figure 19-12.

Like lists, grids can be either finite or infinitely scrolling, where rows or columns of additional items are added incrementally when the end of the grid is reached.
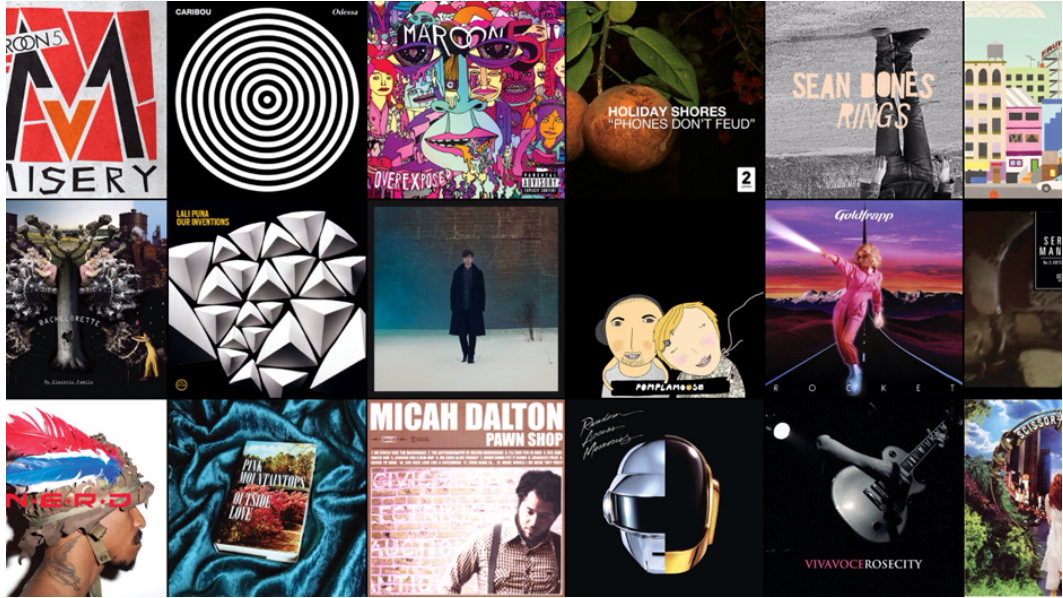
Figure 19-12: When rotated to landscape view on an iPhone, Apple's Music app displays a horizontally scrolling grid of album art. Tapping the art drills down to a view of the album that includes the album art, vertically scrolling track list, and transport controls.

Content carousels

Screen carousels use a horizontal swipe gesture to navigate between similar full-screen layouts containing different data. *Content carousels* live within a single screen layout but use the same type of gesture to allow navigation between different content objects that are presented within that screen. Often they are media thumbnails (or larger images), but they also can be textual or cards containing both media and formatted text.

Content carousels present a row of content items carefully sized and spaced so that they bleed off the edge of the screen. Or they fill the screen from the left to right edge and use either arrows near the left and right screen edges or a page marker widget. Some carousels, such as the one at the top of the iPad's App Store app, make use of a 3D layering effect that puts the focus item in the carousel in front of the others.

Properly designed carousels should wrap circularly from end to beginning, rather than making the user swipe all the way back to the start. They also make it visually clear when the last item in the carousel has been reached.

Typically, content carousels are used to present a relatively small set of objects that the app is meant to feature or highlight. As such, only one carousel should be employed on a screen, and it should take the most prominent position in the layout. An excellent

example of this is the Crackle app on iPhone, shown in Figure 19-13. The app has a large carousel at the top of its Featured tab. It wraps and includes a page marker widget so that users know when they've returned to the beginning. (This trick works only with carousels containing a short list of items.) It also auto-advances the carousel every few seconds—a common variation of the idiom. This helps users understand the behavior, as well as helping the app feel more dynamic and ensuring that users are exposed to the featured items. Care must be taken not to auto-advance a carousel so fast that users have trouble reading or focusing on the content. This animation should also pause while the user is interacting with other elements on the screen to avoid disorienting transitions.
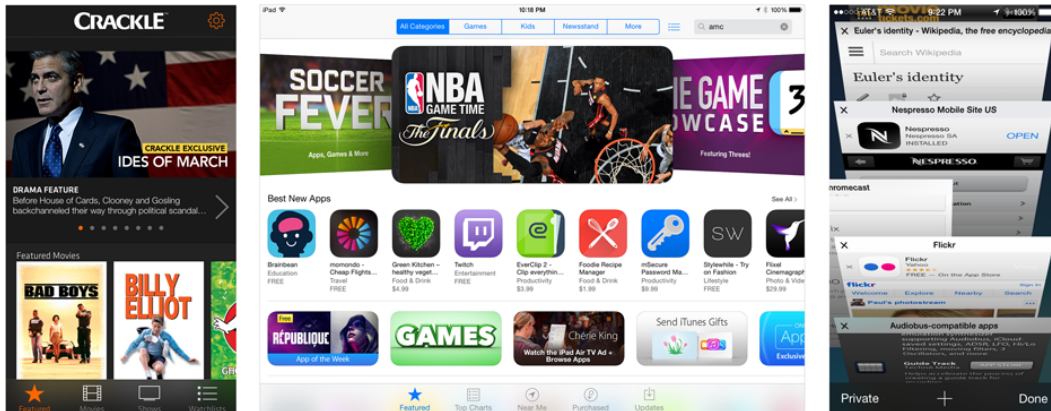
Figure 19-13: The Crackle app on the iPhone (left) offers a good example of a content carousel on its Featured tab. It works well. But the arrow that indicates a drilldown to the details of each carousel item is sized and positioned such that it looks more like a way to navigate to the next carousel item. Safari on the iPhone (right) offers an example of a vertical carousel in place of browser tabs. The iPad App Store app (center) makes use of a 3D layered effect.

Much less common is a vertically oriented carousel. Apple employs one in Safari for the iPhone in iOS 7 in place of browser tabs. The user swipes up and down to browse, taps to select, and swipes left to delete a tab (see Figure 19-13).

Swimlanes

Swimlanes are a clever mash-up of the carousel concept with a grid. They combine the carousel's natural browsability with the data density that only a grid can permit.

Simply put, swimlanes are a vertical stack of carousels, each of which can be scrolled horizontally, independent of the others. Navigating to other swimlanes is a simple matter of vertical scrolling. Swimlanes thus are a clever way to allow the user to browse multiple categories of content with minimal navigation. Swipe through one category, and the next is waiting pristinely below it. This is a big advantage over using a fixed grid that moves columns of content objects all at once.

The Netflix app makes great use of swimlanes for category-based content. Users scroll vertically through the categories and horizontally to browse a category. It works well even though the screen's portrait orientation makes for a narrow viewport on the content, as shown in Figure 19-14. The Apple App Store uses both a carousel and a set of swimlanes on its Featured tab. This combination works well, because the navigational gestures are identical for all elements on the screen.
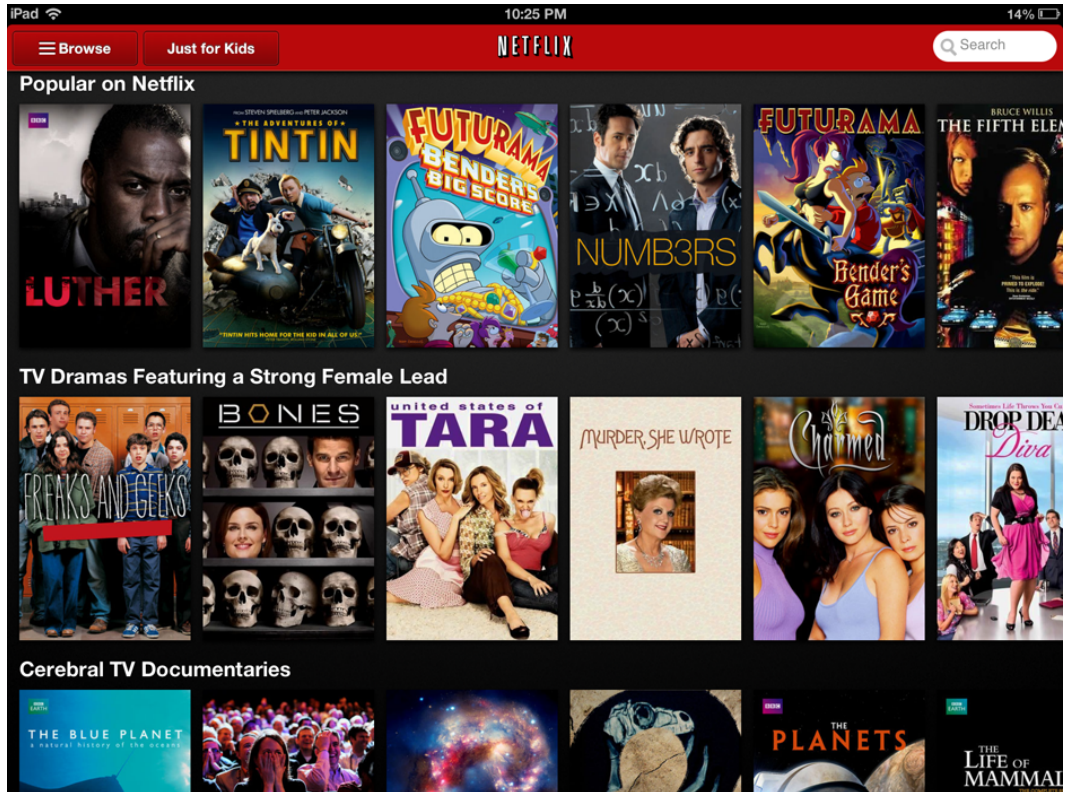
Figure 19-14: The Netflix app uses swimlanes as its primary browse paradigm. Apple's App Store combines the use of a carousel and a set of swimlanes for items highlighted on its Featured tab.

The authors have seldom seen swimlanes that auto-wrap back to the beginning of the list. But they probably should, with a marker of some kind between the end and beginning object so that users receive visual feedback that indicates when they have returned to the starting point in the list. While swimlanes typically are used for finite lists of featured items, you can imagine them being used with infinitely scrolling lists as well. (Imagine categorized search results, for example.) However, swimlanes—unlike carousels—should never auto-advance.

Cards

Cards are a relatively new idiom for mobile that can perhaps in some ways trace their roots to the original HyperCard on the Mac. Back then, Macs had low resolution (much lower than current mobile devices). Therefore, it seemed to make sense to be able to combine text and visual media into nicely formatted chunks of information suited to the display's limitations. HyperCard was meant to be an authoring environment for the masses, but it ended up being a way for developers to easily create rich-media, content-centric interactions.

Fast-forward to modern mobile applications, and the same need arises: How do you present meaningful chunks of rich-media content for easy consumption on a constrained display? Add to this the social and contextual nature of most mobile interactions, and you have what exemplifies the modern card-based UI—a self-contained interactive object combining media, text, web links, and social actions such as commenting, sharing, tagging, and adding media. Facebook and LinkedIn both use cards as a central idiom in their handheld apps, as shown in Figure 19-15.
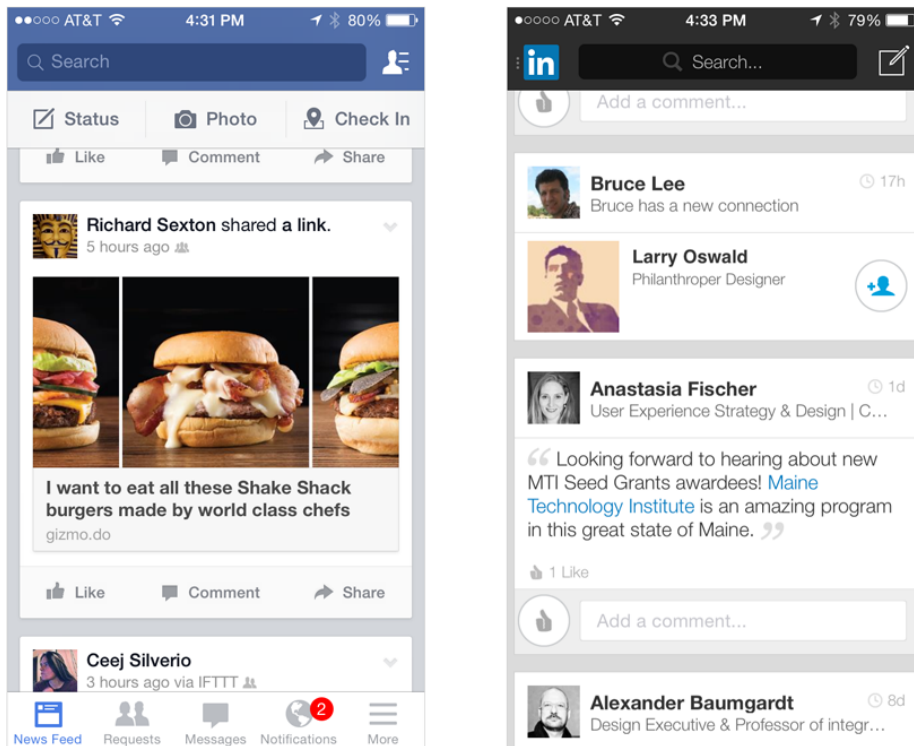
Figure 19-15: Facebook and LinkedIn's apps both use cards as a central idiom.

The Google Search app's Google Now feature has a different approach to cards. It is more focused on contextual information (time, location, and information pulled from

the usage of other Google apps) than it is on social interaction. Google's cards are small encapsulations of data pulled from other Google services, such as weather, maps, stocks, restaurant reviews, and notifications pulled from calendar and e-mail data. Tapping their content takes the user to the full app or web interface from which they originated, providing an avenue for deeper interaction if desired (see Figure 19-16). Google's cards also have individual settings that can be accessed by tapping an icon in the upper-right corner. Doing so flips over the card, revealing access to a configuration interface.
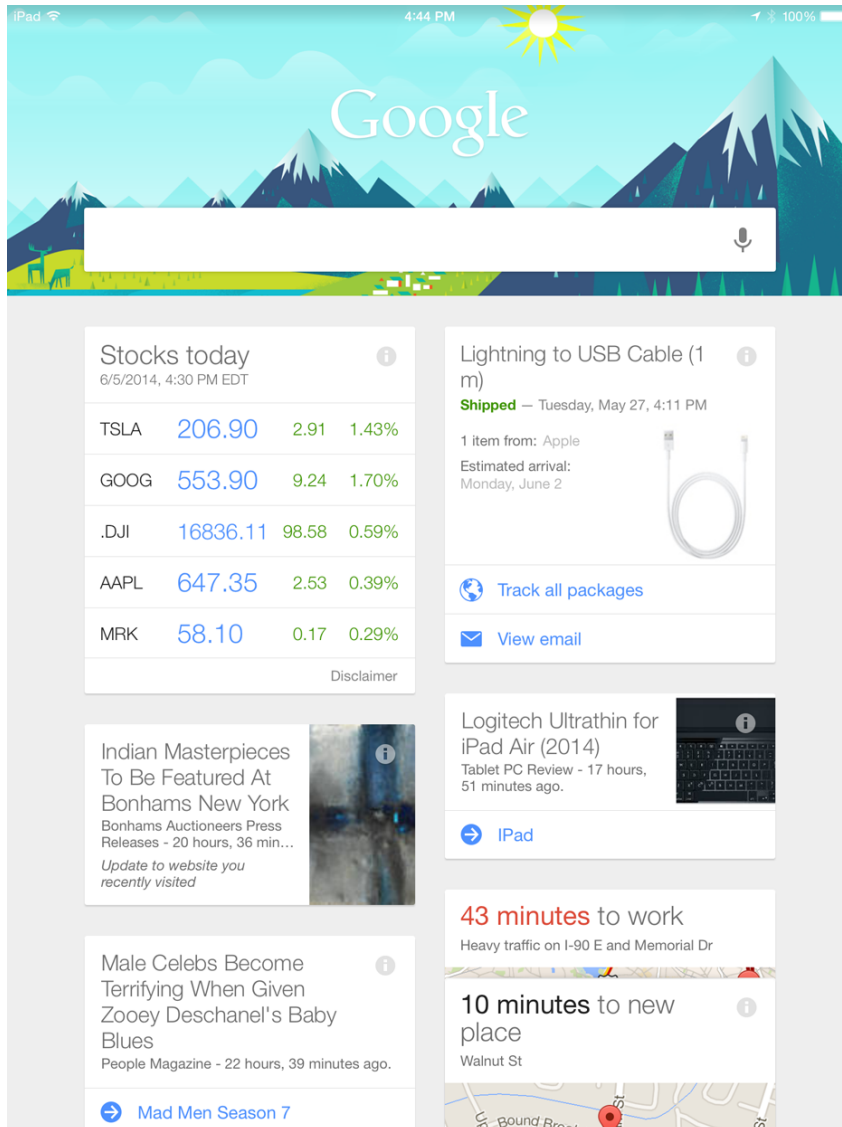
Figure 19-16: The Google Search app uses cards that return encapsulated snippets of useful information based on the user's current context, including location, time, and relevant related information pulled from other Google services.

Cards are most often displayed in a scrolling vertical list, but they also lend themselves to grid, carousel, and swimlane layouts. Facebook's Paper app provides a good example of the use of cards in a nonstandard layout: The top half of the screen is a category card that cycles through individual posts. If it is tapped, the post is expanded on a full-screen card, as shown in Figure 19-17. Under the category card is an infinitely scrolling swimlane of posts fitting the category. Swiping up on these expands the swimlane to full-screen height, making more detailed content visible. (Swiping down returns them to the bottom of the screen.) Tapping any shared content inside the expanded card takes the user to the content's original source.

Figure 19-17: Facebook's Paper app is a good example of cards used in a nonlist layout. Content navigation in the app is achieved via a card carousel of content categories (each of which automatically cycles through recent content) and an infinite card swimlane that the user can browse through.

Navigation and tool bars

Bars are the primary mechanism for navigating to the different functional and content areas of handheld mobile apps. Like lists and grids, they date back to the earliest days of mobile touchscreens. Bars are narrow horizontal regions at the top or bottom of the screen that consist of tab-like or button-like controls with either icons or text labels (and sometimes both, as in many iOS apps). The affordance of these controls used to be more prominent. However, probably to their detriment, the major mobile operating systems have gravitated toward a flat visual style. Although this significantly reduces visual clutter, it also has the unfortunate side effect of requiring more cognitive work for users to identify active controls. At this point, most users have been trained to assume that any text or icon living within a bar is a navigational control of some kind.

Tab bars

Tab bars contain a set of text and/or icon buttons. (iOS tab buttons frequently sport an icon with a text label beneath it.) Tapping a tab button switches to a different list or grid view in the main content area, as you'd expect a tab to do. Each tab in a tab bar maintains its own content hierarchy of associated lists and grids and typically preserves the state of that hierarchy while the app is running. Tab bars are frequently found at the bottom of iOS screens and, more frequently, at or near the top of Android and Windows Phone screens, as shown in Figure 19-18.
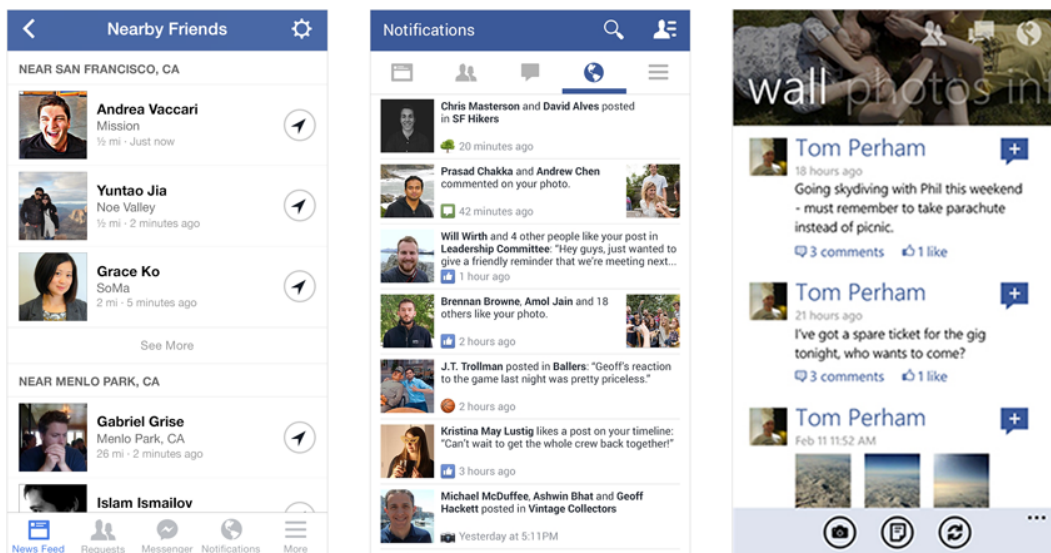
Figure 19-18: Use of tab bars in iOS, Android, and Windows Phone. iOS tab bars typically are at the bottom of the screen, and Android tab bars generally form a secondary navigation beneath a nav bar (or action bar, in Android terms). Windows Phone uses a tab bar that is purely textual, without rendering a bar rectangle.

Some tablet apps use vertical tab bars aligned to the left edge of the screen. Spotify and Twitter currently use this tab bar variant in their iOS tablet apps, as shown in Figure 19-19.
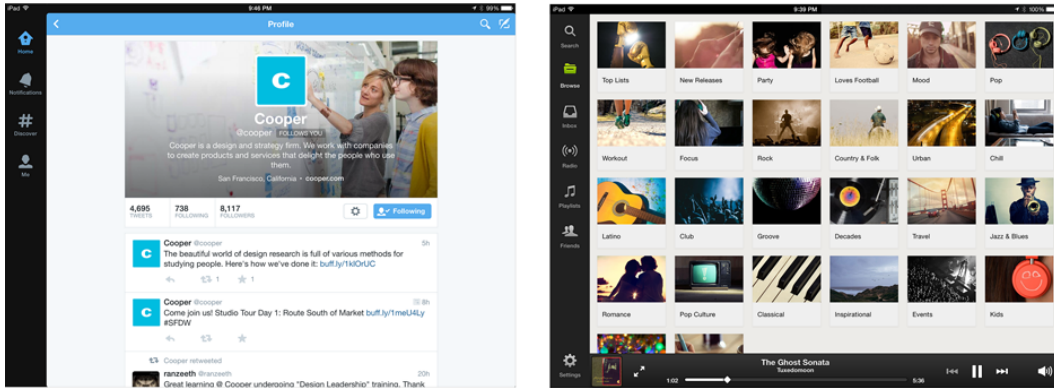
Figure 19-19: Twitter and Spotify use vertical tab bars in their tablet apps. They use buttons containing both icons and text for clarity, which works well given the large amount of vertical space available.

More… controls

The narrow aspect ratio of most handheld screens, as well as the need to provide fingertip-sized hit areas, limits the practical number of controls that can live in a bar to no more than about five. Both iOS and Android deal with this limitation using two strategies.
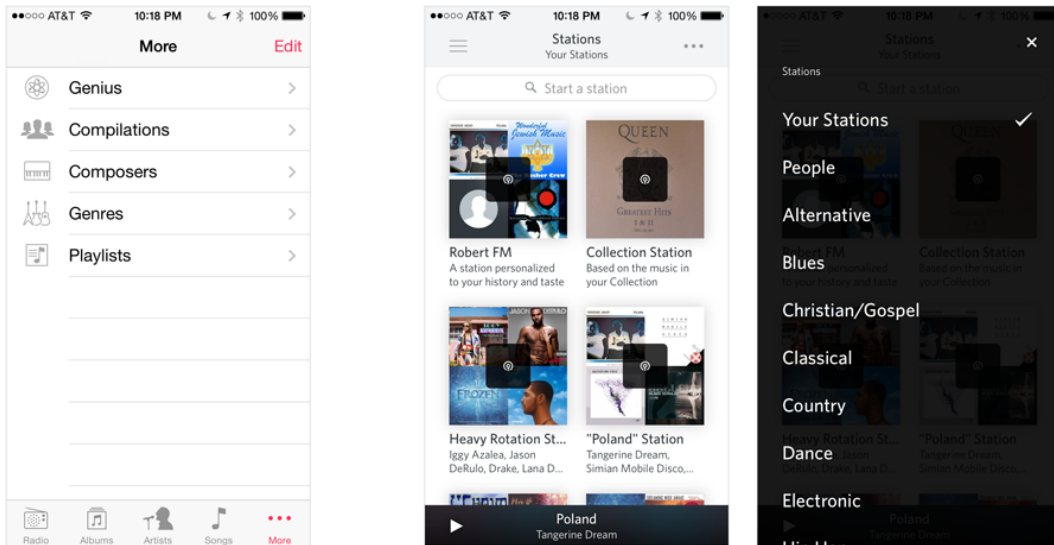
Figure 19-20: More… controls in iOS's Music app (left), and Rdio (right). Rdio's More… control launches a modal pop-up that allows genre selection of radio stations.

The *More... control*, shown in Figure 19-20, is a tab bar or action bar control that gets around the limited screen real estate of mobile apps. In iOS this is usually a tab that shows a screen of additional navigation options. It often has an edit mode that allows the user to drag an option from that screen onto the bar, which swaps the dragged option with the one occupying the slot in the bar that the new option was dropped on. In Android, a More... control lives on the right side of the action menu (see the section on nav and action bars later in this chapter) and opens a pop-up menu of additional navigation options or (more typically) functions. Some iOS apps, such as the Rdio streaming music app for iPhone, use a similar idiom in the upper right of the screen as a way to select additional options via a full-screen modal pop-up.

Tab carousels

A different approach to the same problem that the More... control addresses is the *tab carousel*, which elegantly marries the concept of tabs with that of horizontally swipable carousels. Tabs are shown in the tab bar as usual but extend off the edges of the screen. The selected tab is centered or otherwise highlighted in the tab bar. Tapping another tab selects it. Swiping the tab bar (and, in some cases, the view it controls) selects the adjacent tab on the left or right and slides the contents into view, as shown in Figure 19-21.
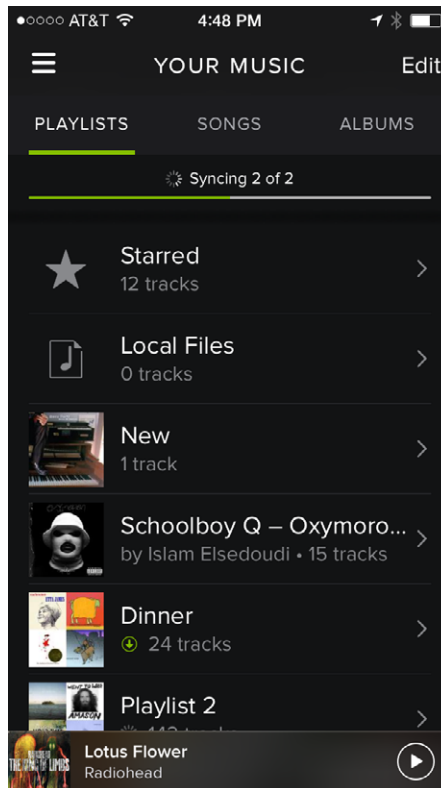
Figure 19-21: Spotify's iPhone app uses a tab carousel in its Your Music section, which is accessed via its main navigation drawer.

As with other carousel views, it is important that at least one tab label is initially shown extending off the edge of the screen, to provide the hint of scrollability in the tab bar. Windows Phone uses a variant of the tab carousel as a primary navigation mechanism in its apps. The tab bar is not rendered, but purely textual tabs are employed (see Figure 19-18).

Nav bars and action bars

Nav bars, located at the top of the screen, provide a way to navigate a list or grid hierarchy, as shown in Figure 19-22. Typically they contain at the very least a back button on the left and the title of the current list, grid, or other type of content screen in the center. Android calls this set of controls an *action bar*. Frequently, function menus or buttons are included on the right.
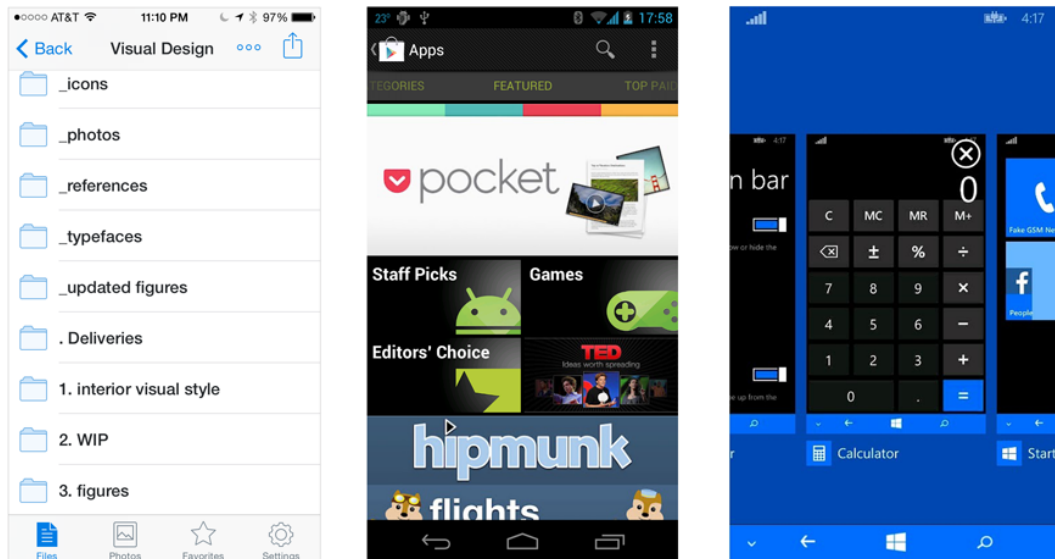
Figure 19-22: Use of nav bars in iOS (left), Android (center), and Windows phone (right). Android encourages an action bar at the top of the screen, which incorporates navigation and access to functions. Android and Windows Phone also makes use of a system-level nav bar at the bottom of the screen. Windows Phone's Metro design language discourages use of top nav bars.

Most versions of Android (and Windows Phone, as of 8.1) have a system-level navigation bar at the bottom. It contains a back control (which takes the user to the previously viewed screen, regardless of app or hierarchy), a home control, and a "recents" control (Windows Phone also includes search). The presence of a ubiquitous bar at the bottom means that Android apps typically place most of their app navigation at the top of the screen.

Tool bars and palettes

Tool bars contain buttons that execute functions on the current or selected app content. Windows Phone permits four action buttons in its action bar (called an *app bar*), which typically is placed at the bottom of the screen.

iOS apps often place an action button or two on the right side of their nav bar, but apps designed to let you author or edit media rather than simply viewing or sharing it often replace the standard bottom-of-screen tab bar with a tool bar.

Google encourages the use of its top-of-screen action bar, which combines back navigation with action buttons. It recommends adding a tab bar under this if the user needs to navigate multiple views. Google's action bar even supports view switching via a drop-down on the action bar itself if the stacked action and tab bars take up too much space.

Most audio playback apps place a transport bar or control pane containing playback-related controls at the bottom of their "now playing" screen.

Tool palettes, a variant of tool bars similar to their desktop brethren (see Chapter 18), use iconic buttons as a way to access tools that operate on a document. (Drawing and painting tools are the most obvious example.) Tool palettes on tablet apps make heavy use of pop-up control panels to allow the selection and configuration of tools.

Vertical tool bars and palettes

On tablets, more-complex tool bars supporting pop-up control panels and palettes are used at both the top and bottom of the screen. *Vertical tool bars* run along the left or right edge of the screen (and sometimes both). Art Studio, shown in Figure 19-23, is a good example of an app that makes heavy use of rich, complex tool bars.

Tool carousels

Just as carousels have crossbred with tab bars, they have also combined with tool bars, allowing more functions than can comfortably fit across the screen to be accessible with a horizontal swipe. *Tool carousels* seem particularly popular with image processing apps such as Google's Snapseed, shown in Figure 19-24. Each item in the tool carousel is a labeled thumbnail that both describes and shows a small example of the filter or effect applied to an image. (In an ideal world, the image would be the one you were actually editing at the time, but scale can become an issue.)
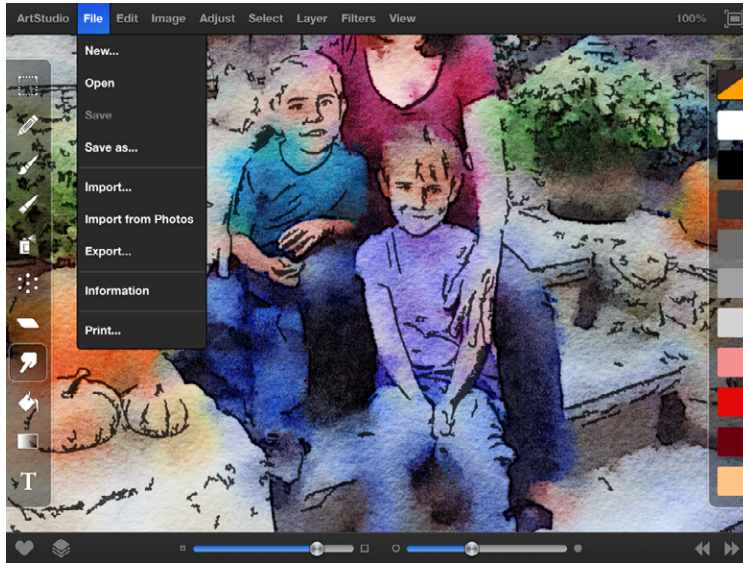
Figure 19-23: The Art Studio app uses vertical tool bars as well as a desktop-like menu bar and sliders embedded in its bottom tool bar. Authoring tools like this begin to rival the complexity of desktop applications. The tablet screen becomes quite cluttered with this many controls, so Art Studio lets you hide them while working, similar to desktop design tools such as Adobe Photoshop.

Figure 19-24: Google's Snapseed app uses a carousel to let you select the tool. After it is selected, the appropriate controls for the tool are shown, in some cases including a secondary tool carousel for choosing a specific setting.

By stacking two bars, you can build a rather complex set of features in a way that tames the complexity. A tool bar lets you select the category of tool (effects, filters, adjustments), and a tool carousel contains items for each specific tool or variant in a category.

Menu bars: an idiom to avoid in mobile

As complex authoring tools make their way onto tablets, more of the trappings of desktop applications are also making their way to tablet user interfaces. Apps like Art Studio (see Figure 19-23) and Cubasis (see Figure 19-6) for iOS use complex, desktop-like control layouts. Art Studio takes this a step too far by implementing a desktop-like menu bar.

This isn't a good idea for a couple reasons. First, a row of text labels in a bar typically means a tab bar, and a desktop-style menu bar interaction is unexpected on a tablet. Second, most of the functionality remains hidden in the menus. Once it is exposed, it still isn't clear from the menu label what the functions will do. An approach that uses both a tool bar and a tool carousel (as described in the preceding section) can accomplish most of what a menu bar can do, but in a less visually dense and more visually explanatory way.

Drawers

Drawers are a clever idiom that provides access to a vertical list of navigational elements similar to tabs. They use minimal screen real estate by hiding in a panel that lives in a layer under the main content area. The drawer icon is also called the hamburger menu icon due to its shape: three short, stacked lines. Tapping this icon—or, sometimes, swiping across the main content area—slides the content area horizontally to reveal the drawer under it. As with tabs, the current selection is highlighted. Tapping a drawer item simultaneously swaps what is displayed in the content area and snaps the drawer back shut. Items in the drawer are usually textual, but may have icons and other adornments. Additional controls may also live in the drawer. Google's Gmail app on the iPhone, shown in Figure 19-25, illustrates a typical use of the drawer idiom.
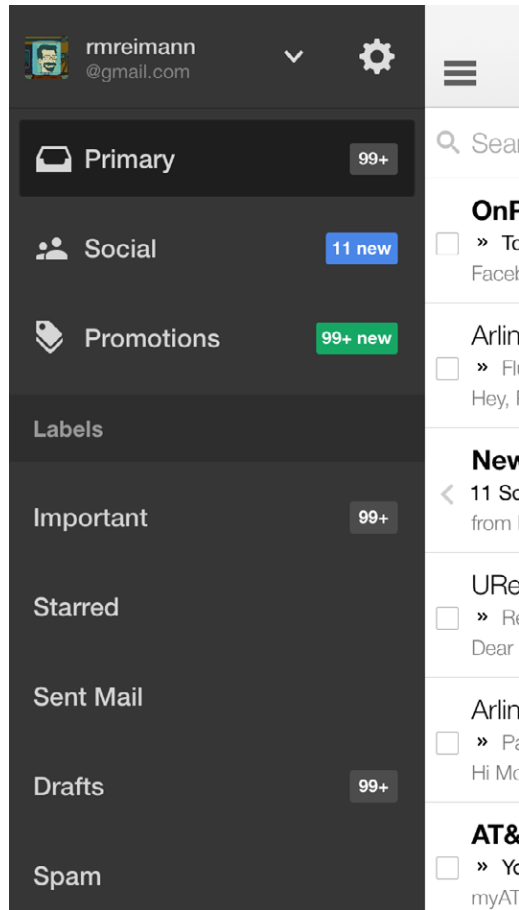
Figure 19-25: The Gmail app on the iPhone uses a drawer with additional navigation elements inside it. It's a little disconcerting that the account management UI slides down from the top while the settings UI slides up from the bottom (and takes up the full screen), even though both controls are next to each other in the drawer.

Secondary-action drawers

Drawers can be used to replace a navigational tab bar or can be used to interact with a secondary set of objects in the app. Drawers usually slide open from the left, but not always. Some secondary actions are put in a drawer that deploys from the right. The current version of the Facebook app for the iPhone uses a set of fairly standard bottom tabs (including a More... tab) for its main navigation. It also offers a right-hand drawer that gives you access to a list of online friends for chatting, as shown in Figure 19-26.
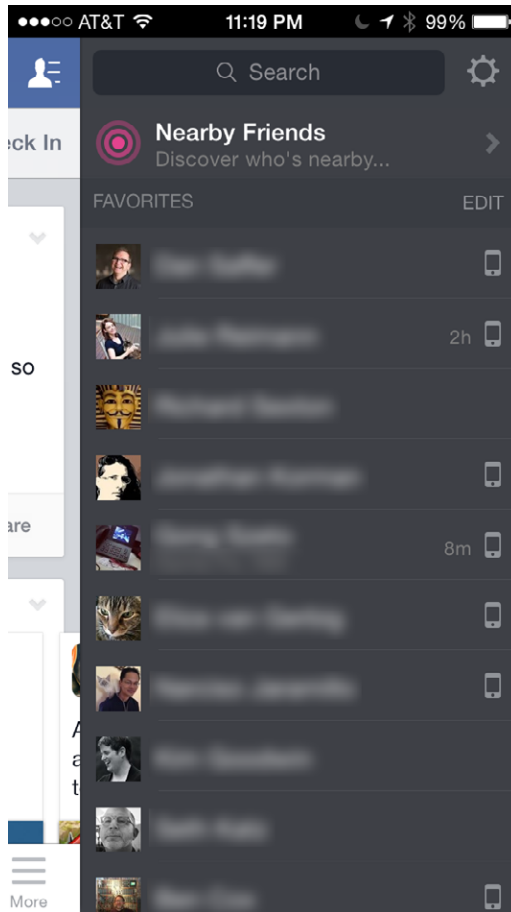
Figure 19-26: The Facebook app on the iPhone uses a right-hand drawer to let you access online friends for chatting.

Double drawers

Path, an intriguing timeline-based social networking app on iOS, has successfully opted to minimize its use of tab and tool bars in favor of idioms that take up less main screen real estate. The Path design, as shown in Figure 19-27, uses two drawers—a standard left-hand drawer for primary navigation between views, and a Facebook-like right-hand drawer for messaging friends. Path also uses a nonstandard but interesting tool menu control that fans open from the lower-left corner of the main content area when activated. Although it adds a tap to access these functions, the interaction is both clear and pleasing in its execution, and it allows the content area to shine.
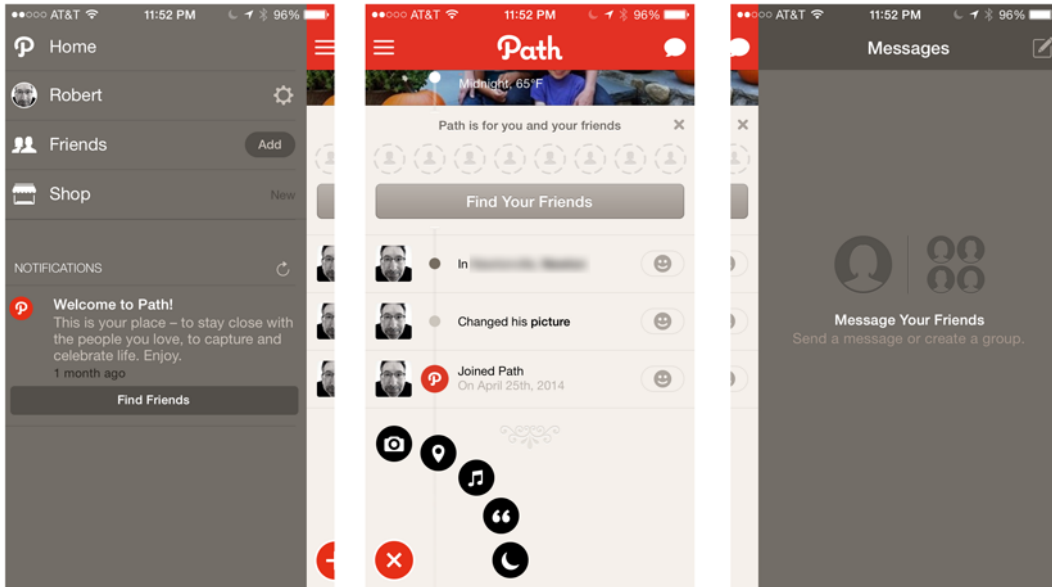
Figure 19-27: The Path app on the iPhone uses both a standard left-hand drawer for primary navigation and a right-hand drawer for messaging friends. In addition, Path uses a nonstandard pop-up action menu that fans out from the lower-left corner of the main content area when tapped.

Item-level drawers

Some handheld format apps have taken the concept of a slide-to-reveal drawer and applied it to individual items in a list. Sliding an item to the left or right (depending on the app) reveals a tool bar under the item, whose functions perform an action on that item. This avoids the need for a tool bar at the top or bottom of the content area. Although this approach may seem clever, it actually has a number of drawbacks:

- It is difficult for users to discover unless some sort of visual cue is added to the list item. But then it needs to be added to all items, wasting horizontal space. Desktop applications have the benefit of the hover state to reveal such controls without cluttering the interface, but mobile apps do not.

- The swiped item can be obscured when the drawer is open, so the user would need to remember what it is, adding to mnemonic (memory) work.

- The per-item swipe gesture means that other, more standard horizontal gestures, such as those for deleting an item or opening a global navigational drawer, may become either confusing or impossible.

The Slacker streaming music app on iPhone, shown in Figure 19-28, provides a workable example of item-level drawers for both list and grid items. Swiping to the left on grid items for artists, stations, or albums, and swiping list items for tracks, reveals a drawer

containing an info button. Tapping it takes the user to a detailed metadata screen. While this idea reduces UI clutter, its discoverability is low, because swiping individual grid items is a nonstandard interaction. Therefore, this type of interaction requires some explanation in a welcome or help UI, and even then it's questionable whether most users will find it.
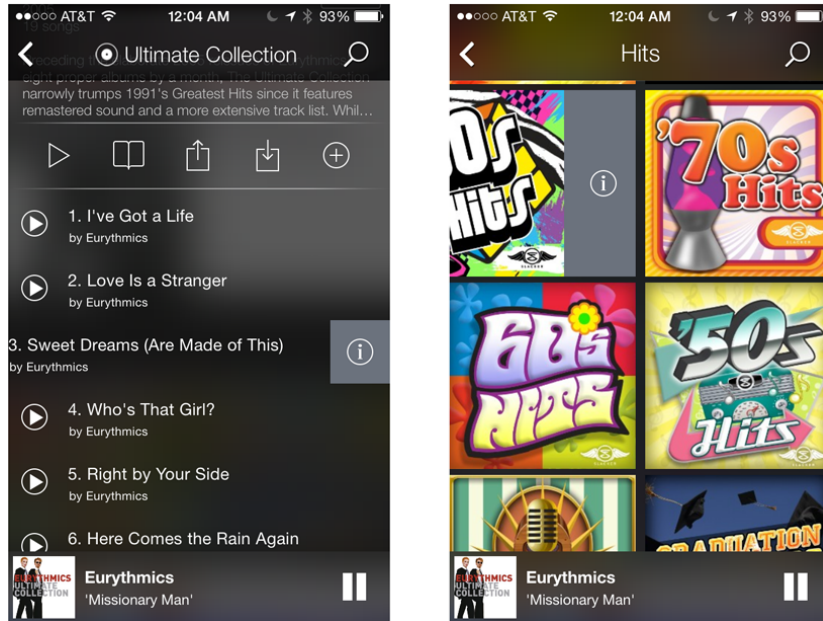
Figure 19-28: Slacker's streaming music app uses item-level drawers in both grid and list views to give access to an info button that takes users to a detailed metadata screen. Although it is elegant in terms of avoiding clutter, its discoverability is low.

Drawer behaviors to avoid

The Gmail app's drawer implementation, shown in Figure 19-25, also presents an object lesson on the need to carefully consider the overloading of animated transitions for accessing options.

The Gmail app's main drawer opens as expected, with the content pane sliding off to the right when the drawer icon is tapped or when the content pane is swiped to the right. Within it, the navigational choices (e-mail folders) scroll up and down as expected.

From there, things get complicated. The account management UI is a toggle control in the action bar at the top of the drawer. Activating it slides down a pane that covers the drawer's contents until either an account is chosen or the pane is dismissed. And, next to the account management toggle is a settings button, which launches yet another sliding

pane that slides up from the bottom of the screen, covering both the open drawer and what is visible of the main content area. Sound confusing? Well, it is.

This overloading of popping and sliding panes—each moving in a different direction and affecting different layers of the UI—can be both disconcerting and confusing for users.

Limit the number of animated screen transitions.

Unlike Google's Gmail app, the Google+ app for iOS, shown in Figure 19-29, breaks drawer convention. It slides the drawer open on top of the main content area, rather than having the content area slide over to reveal the drawer underneath. This type of behavior usually is seen on tablets when an index pane of content is opened in portrait mode. It's puzzling why Google didn't stick with the more appropriate drawer idiom it was already using for its Gmail app.
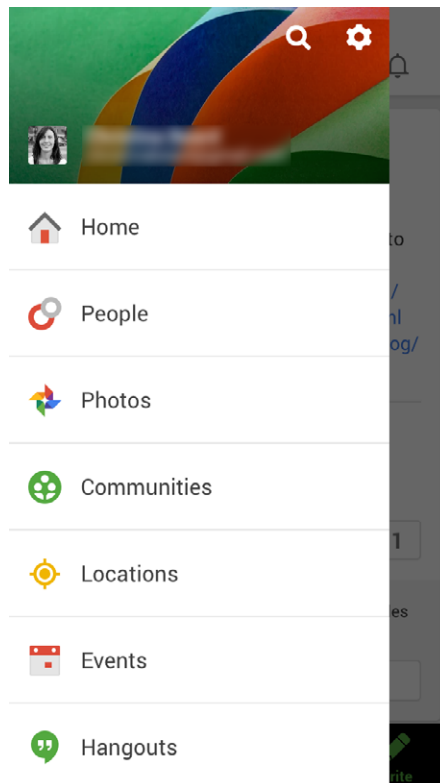
Figure 19-29: The Google+ app breaks the drawer pattern by sliding it over the content area—more like a content index pane—instead of sliding away the content area to reveal the drawer's contents.

The drawer controversy

Drawers using the hamburger menu have come under fire, the claim being that use of drawers hampers user engagement by hiding functionality. Some in the community have fiercely advocated for drawers to be abandoned entirely. We believe that this is throwing the baby out with the bathwater.

Certainly there is some truth in these claims: Hiding an entire nav hierarchy behind a single icon button does have its problems, but these can also be remedied via use of a text button (e.g., Menu) instead of—or in addition to—the hamburger icon, having the initial use state of the drawer be open, or making use of an initial help overlay (see the sections on welcome and help screens later in this chapter). In some cases the style of the hamburger might be the biggest problem—if the user doesn't register it as a control, then the failure is in visual communication.

Benefits of drawers include a cleaner main interface with more room for content, and a means of making almost any function a swipe (and possibly a scroll) and a tap away. For an app with a complex feature set, this can be a godsend.

For apps where you expect users to be constantly using many functions, drawers may also work well. And apps that have many infrequently used but occasionally necessary functions might benefit from a drawer approach. On the other hand, apps that are only casually used may best use tabs (or one of their variants) for navigation, since users will not be using the app with enough frequency or dedication to recall the existence of features hidden within a drawer.

Tap-to-reveal and direct manipulation

One of the main differentiators of touchscreen mobile apps from desktop apps is the ability to use your fingers to manipulate onscreen objects. Navigational constructs such as lists, carousels, and drawers allow users to navigate in a more immersive way, and the same principle can be applied to creating and editing content.

Tap-to-reveal controls

The iDraw app, shown in Figure 19-30, provides a good example tap-to-reveal: Tap an object, and the manipulation tools are revealed.

Similarly, streaming video apps use the tap-to-reveal idiom for controls that normally are hidden during playback. Tapping anywhere on the video playback area of the YouTube app (see Figure 19-31) launches transport, volume, and other controls.
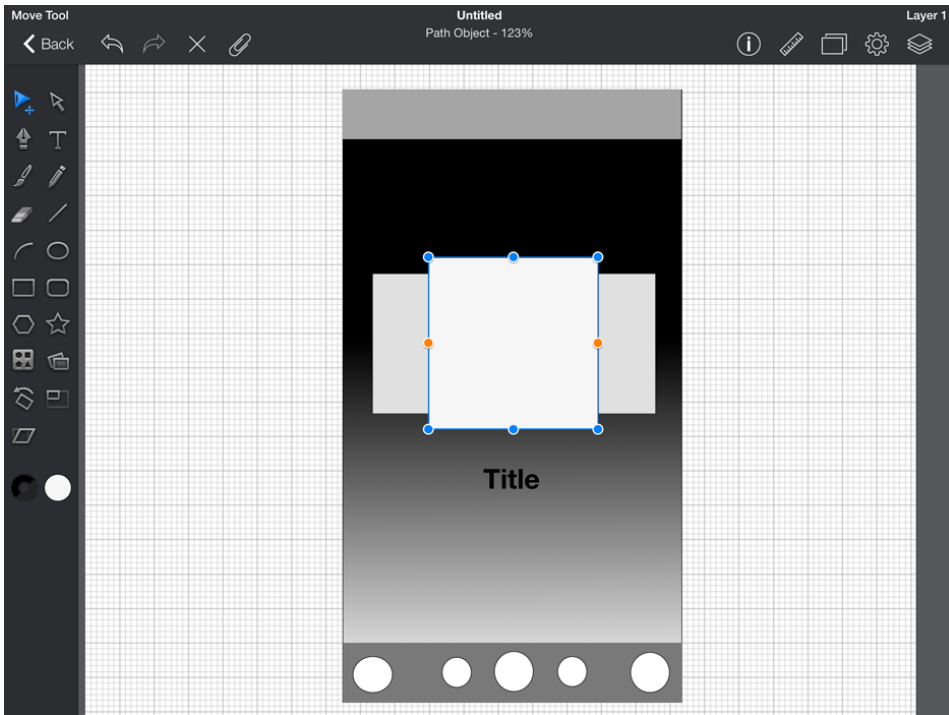
Figure 19-30: The iDraw app uses traditional desktop-style drag handles that appear when an object is tapped. An additive selection mode allows successive taps to select additional objects as a group.
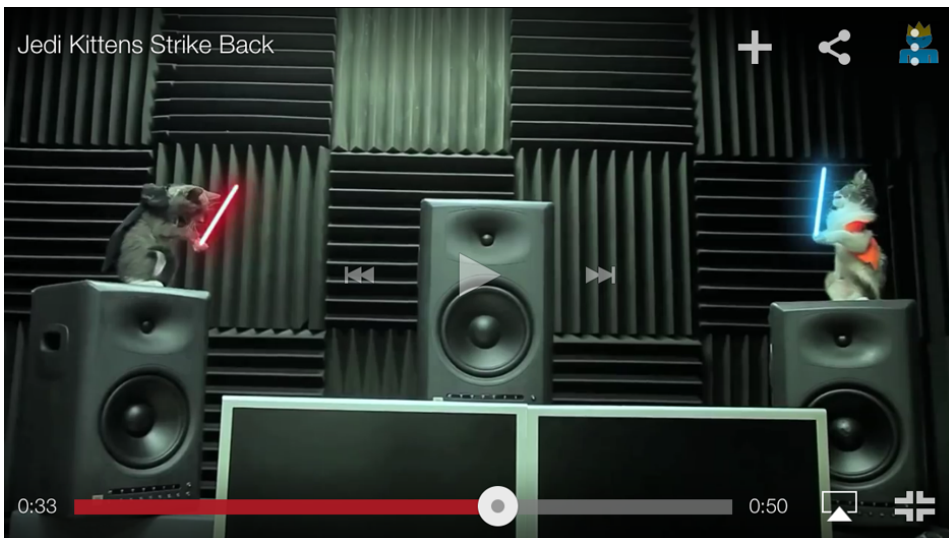
Figure 19-31: YouTube makes its transport, volume, and other controls temporarily available as icons superimposed on the video display area when it is tapped. This design method helps eliminate clutter, but it must be discovered. Luckily, most mobile video apps use this idiom, and tapping the playback area isn't that much of a stretch discovery-wise.

Direct manipulation controls

Some apps go to the next step of direct manipulation that touch-based screens permit—replacing cumbersome indirect-manipulation idioms such as sliders with gestures on the object being edited. The best of these, such as Google's Snapseed image editor, provide dynamic feedback hints that show roughly how the gestures will affect the object being edited. For instance, when you use the tilt-shift effect, tapping the image displays a center adjustment point, as well as sets of double lines indicating the effect's angle and transition interval (see Figure 19-32). The user can move the effect's center point, swipe horizontally to widen or narrow the transition area (also tracked by a thermometer-like display below), and twist his or her thumb and forefinger on the screen to change the angle of the effect. Although some discovery and learning are involved, it quickly becomes second nature and provides a tremendously immersive way of editing and correcting photos.

Figure 19-32: Snapseed provides innovative and highly immersive direct-manipulation tools for editing images, eliminating the need for the traditional banks of knobs and sliders that such interfaces usually entail. The price of this approach is a steeper discovery curve, but this disadvantage can be offset as Snapseed does—with one-time welcome/help screens for each tool.

Searching, sorting, and filtering

Searching is a key user activity on mobile apps. In fact, it is arguably the most important mobile activity besides making a phone call. People use mobile apps to find something, whether it's a recent e-mail, a song or video, something to buy, or something in the real world that's in their vicinity.

As mentioned above, complex data entry is not easy or practical in the on-the-go world of mobile apps. Luckily, you have a variety of helpful ways to minimize the effort in search, as well as contextual information that mobile apps can provide.

Implicit sorting versus explicit searching

As discussed earlier, mobile apps are, by and large, optimized for browsing. We can utilize that browsing behavior to help pre-empt a user's need to build search queries. A smart app might keep track of the kinds of things the user has viewed, liked, or purchased in the past. Then it could serve up those items (more on this first option below) items that share similar properties or are liked by people with tastes similar to those of the user. Netflix has based its mobile app on this principle (see Figure 19-14), providing swimlane categories of TV shows and movies gleaned from the user's watching habits. Search is still available but is not the focus of the interface.

Building search queries

Of course, even with the best possible browse options, the user's need to search for something specific is almost inevitable. The challenge in a mobile app context is to allow sufficient expression of search terms, but with a minimum of data entry for the user. Here are a few of the most useful approaches:

- **Voice search**—The three major mobile platforms support in-app voice search, and you should certainly make this an option in your app. Voice search can certainly ease entry for simple searches in supported domains. However, we're a long way off from completely reliable general searches, so the need to enter and modify search terms manually still remains.

- **Auto-complete**—As the user types, displaying a list of popular options matching the entered letters can dramatically decrease keyboard time and user frustration.

- **Tap-ahead**—This is a refinement on top of auto-complete. Tap-ahead allows users to take any auto-completed term option the app provides as the result of auto-suggest, load it into the search box, and run a new auto-complete query. This might be overkill for some searches, but it is certainly useful for web searches and in more technical domains where precision of search terms might be important. The Google Search app uses tap-ahead, as shown in Figure 19-33.
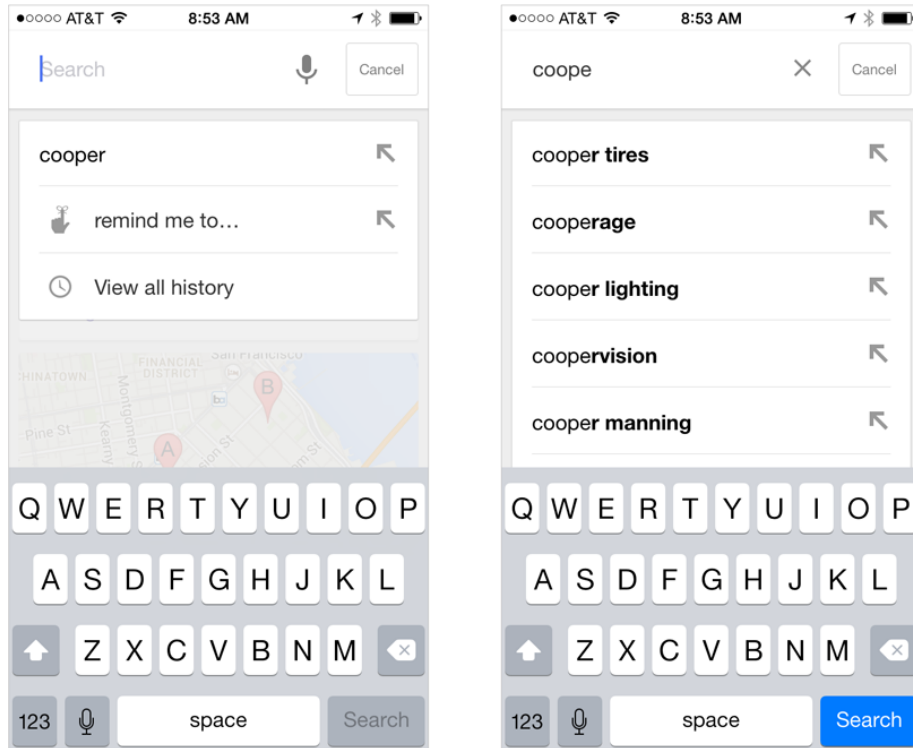
Figure 19-33: Google's Search app uses voice search and recent/frequent search suggestions (left), auto-complete (right), and tap-ahead (both).

- **Recent/frequent searches**—Humans are creatures of habit who typically search for the same things repeatedly. Any search functionality should remember past searches and present them as soon as the user taps the search box. Ideally these results should be organized in order of most frequent and most recent. They also should support tap-ahead so that they can be used to start a related search if desired, as the Google Search app does.

- **Auto-suggest**—A more sophisticated improvement on strict auto-complete, auto-suggest uses fuzzy matching techniques to provide spell-corrected, controlled-vocabulary, and synonym options in its option list. Typically, auto-suggest options include a small set of strict auto-complete options at the top, with a larger set of suggested results beneath.

- **Categorized suggestions**—Building on auto-suggest, an app that needs to search across several types of data can provide suggested options in each category. iOS's Spotlight search, shown in Figure 19-34, does this well. It provides instantaneous categorized suggestions (with thumbnail images where appropriate) pulled from apps, contacts, music, videos, mail, messages, calendar, notes, reminders, and more.
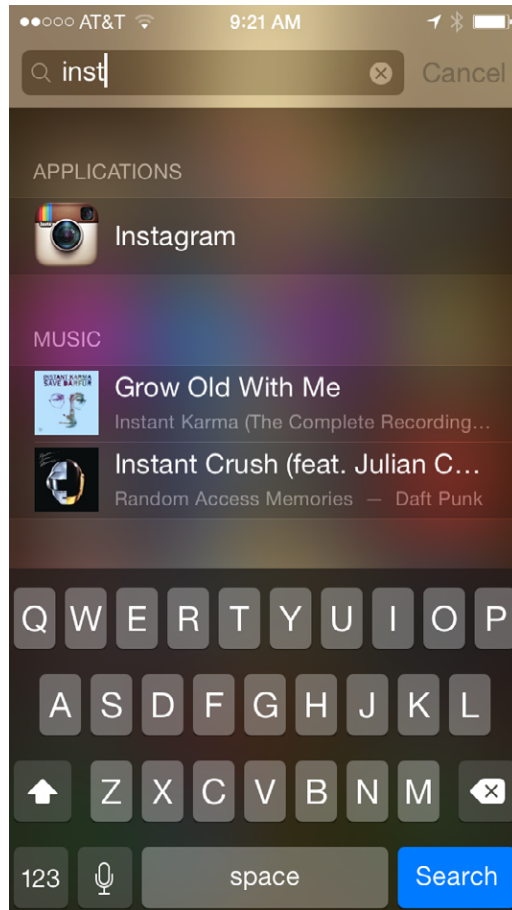
Figure 19-34: iOS's Spotlight search uses voice search, auto-suggest, and categorized suggestions.

Sorting and filtering

On mobile devices, sorting and filtering often amount to the same thing. This is because the combination of limited screen real estate and the limited amount of time users typically have in an on-the-go mobile context limits the number of search results users will want to scroll through to a few screens at most. Thus, sorting effectively results in filtering out items at the bottom of the sort. Add to this the fact that users don't always understand the difference between sorting and filtering, and you can anticipate the appropriate strategy for these functions on mobile: Merge them into a single set of controls. Unfortunately, many high-profile apps don't get this quite right.

Amazon's iPhone app, shown in Figure 19-35, has a straightforward search that remembers recent searches. It also has a clear Refine button in the nav bar of the search results;

this is good so far. But the refine UI infuriatingly forces you to choose a department before you can even see a sort by option (or any other filter options) *and* takes you back to the results page before you can choose it! The consequence is that users may not even realize that additional sort and filter options are available.
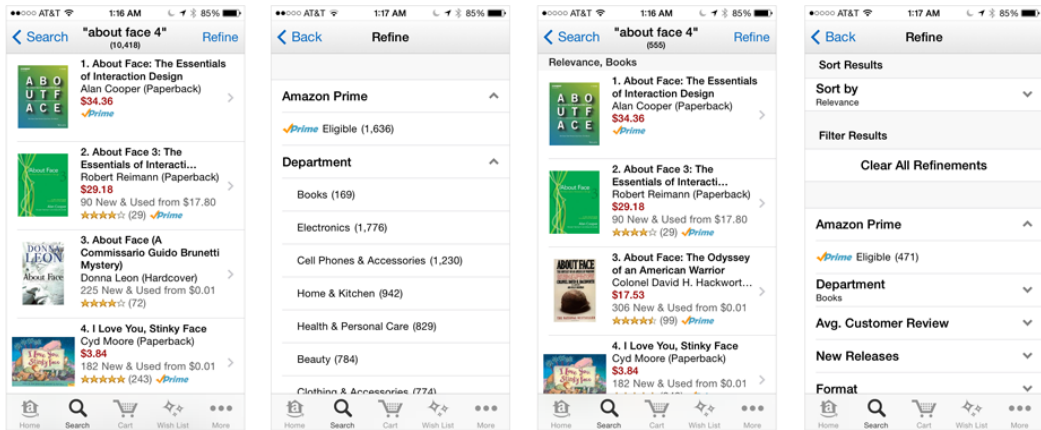
Figure 19-35: Amazon's iPhone app fails users by allowing them to choose only one refinement option at a time and by hiding most refinement options until a department filter is selected. Undoubtedly this is due to a database integration issue on Amazon's back end, but it is Amazon's customers who suffer.

The OpenTable app, shown in Figure 19-36, takes a better approach for users. The search portion of the interface has appropriate filters for a restaurant reservation app built in: time and location, as well as the expected keyword search for restaurants. Both time and location are also sensibly prepopulated. Search refinement options are clear and simple, with the most important at the top, and more fussy criteria collapsed at the bottom. The only *faux pas* OpenTable makes is placing its filter control behind a somewhat obscure icon in the lower right of the screen, where people are almost sure to miss it.

Yelp takes a no-nonsense approach to refinement in its app, with a prominent Refine button to the left of the search box on the results screen (see Figure 19-37). Tapping the button opens a full-screen dialog that mixes filter and sort controls, each of which is clearly labeled and appropriately prioritized from top to bottom.

Yelp and Amazon both get another detail right: Filtered results are indicated by a narrow filter bar anchored to the top of the results view. This bar contains a terse textual summary of all current filters on the results. A nice addition to this interface would be the ability to swipe horizontally to see a full list of active filters (the list is truncated in the Yelp interface). Another advantage would be the ability to tap to toggle the filters on and off without needing to return to the refine screen.
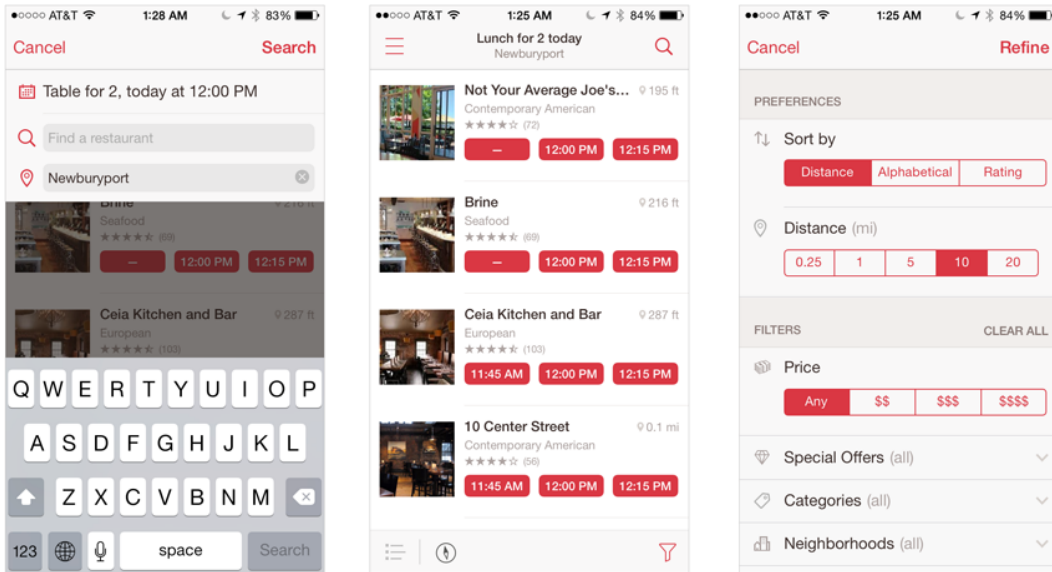
Figure 19-36: OpenTable's app does a great job searching (left) and filtering (right), except for the filter control placement (center). It's almost invisible in the lower right of the screen, especially since it disappears entirely when you scroll down (though it does come back when you scroll up).
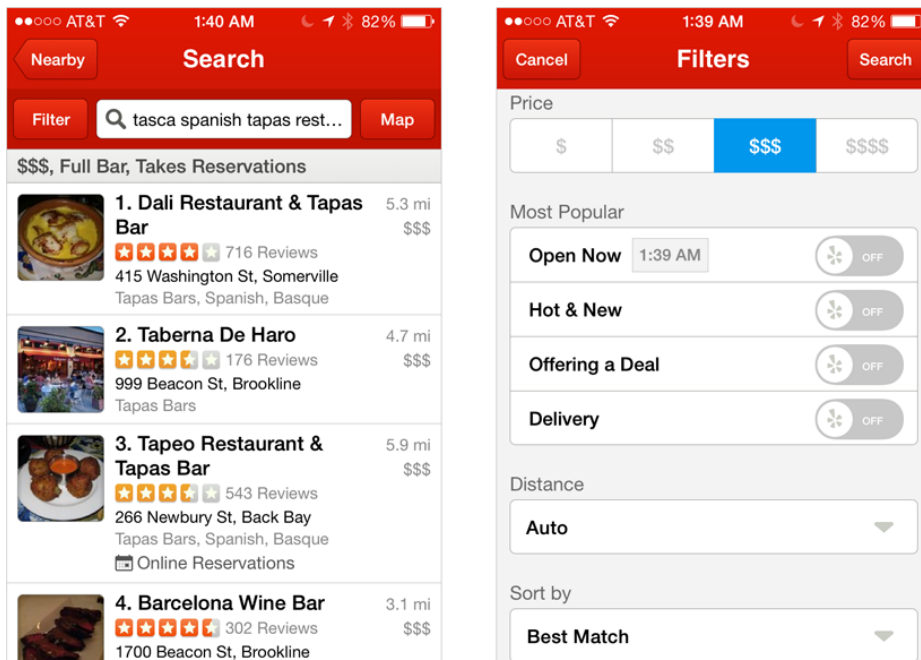
Figure 19-37: The Yelp app gets searching and filtering right. A clearly labeled Filter button is at the top left of the results screen (left), and a full-screen modal pop-up appropriately mixes filter and sort criteria (right). The results screen also shows a narrow filter bar identifying which filters have been set (left).

Welcome and help screens

While many mobile interfaces are quite easy to learn, some forces unique to mobile design act in opposition to ease of learnability:

- Limited screen real estate effectively limits the amount of textual labeling or instructive text that can be on the screen at any given time.
- Multi-touch interfaces center on the use of gestures to accomplish actions, which have no visible affordance until fingers touch or move across the screen.
- Unlike mouse-driven desktop interfaces, there is no hover state to afford tooltips and other contextual hints.

While some other alternatives exist, the simplest and most effective way to help users learn a mobile interface is through welcome and help screens.

Welcome and help interfaces are most often two sides of the same coin in mobile apps. On a user's first-time entry into an app after purchase and login, *welcome* screens provide guidance on what the important activities are in the app and how to perform them. *Help* in a mobile context provides much the same—and often identical—information, but on demand when the user requests it. Most mobile apps aren't complex enough to require separate welcome and help screens for different tabs or other views, but this might make sense for a more sophisticated authoring tool with many controls, options, and actions.

This section describes several popular welcome and help idioms that help users learn an app's primary gestures and interactions. For more detail on these mechanisms, see Chapter 16.

Guided tours

Guided tours usually consist of a carousel of cards, each of which contains text and images or video that describes the use of a particular function or set of related app functions. Guided tours are employed at first use and after a major release to highlight new functions, and secondarily, as help. Many apps allow users to relaunch the tour from an in-app settings dialog or, less frequently, from a help button or menu placed in a top or bottom navigation bar. Guided tours should allow users to exit the tour from any card or screen.

> **DESIGN PRINCIPLE** *Use guided tours to orient first-time users.*

Overlays

Overlays are another simple way to help the user get started. An overlay covers the entire screen with a semitransparent layer on which instructions—often rendered to look hand-drawn, and employing arrows to indicate gestures or highlight controls—are displayed. Tapping anywhere on the screen dismisses the overlay. (This is sometimes also accomplished with a close box.) As with guided tours, overlays can be activated again from a help button or settings dialog.

> **DESIGN PRINCIPLE**   *Use overlays to explicate gestures.*

ToolTip overlays

ToolTip overlays are an overlay variation that attempts to provide a ToolTip-like display of all primary functions on a single overlay screen, and is often used in the context of more complex authoring apps. As such, this idiom is best not used as a welcome screen, but rather as a help screen.

Multi-Touch Gestures

Gestures are at the heart of the mobile experience. Although the kind of experience afforded by gestures is quite rich and immersive, the actual number of core gestures is fairly small—and this is for the best. Users don't need a huge vocabulary of gestures to satisfy their needs; keeping gestures simple and straightforward makes them easy to discover and learn.

This section describes the primary uses of the most frequently used multi-touch gestures.

Tap to select, activate, or toggle

The tap is used to select objects and toggle the activation state of controls. Tapped items should get an appropriate selection highlight or activation/deactivation state or animation.

Tap-and-hold

Tap-and-hold is a gestural idiom that is falling out of favor, and probably rightfully so. It is typically used to open a contextual pop-up menu on an object, similar to the desktop

right-click idiom. However, this gesture isn't very discoverable, and few users are familiar with it. Therefore, this gesture isn't recommended.

Instead, a visible menu control should be placed on the object. Or a tap-to-select model, combined with an action menu, should be used.

Drag to scroll

Drag to scroll can work horizontally or vertically, and is a fundamental direct manipulation gesture.

Vertical dragging can be used to scroll lists or, in conjunction with drag handles, reorder objects in a list. Dragging downward on a list can initiate a refresh when the list has already been scrolled to the top. A drag upward can initiate an incremental addition of items after the last displayed item in a list.

The top and bottom drawers supported by some mobile OS's also can be accessed via vertical dragging.

Horizontal dragging can scroll a carousel or swimlane, or open a left-hand or right-hand drawer.

Drag to move

Dragging also can be used to move or copy an object from one list, pane, or container to another, or to move an object arbitrarily within a canvas or grid.

Drag to control

Dragging also can be used to control knobs, switches, sliders, virtual x-y control pads, and contextual touch controls, and to operate palette tools (such as brushes in a painting app) on a canvas.

Swipe up/down

Swiping up usually is synonymous with dragging up, although iOS uses a swipe up gesture in desktop edit mode to close a running app. Swiping a list or grid upward causes it to continue scrolling for a while with simulated momentum.

Swiping down usually is synonymous with dragging down. Swiping a list or grid downward causes it to continue scrolling for a while with simulated momentum.

Swipe left

Swiping left usually is synonymous with dragging left. Swiping a carousel or swimlane to the left causes it to continue scrolling for a while with simulated momentum.

Swiping left also can open a right-hand drawer or close a left-hand drawer.

Apple's Safari browser uses a swipe left to navigate like the forward button. Google's Chrome browser uses a swipe left to delete browser tabs when in tab edit mode.

Swipe right

Swiping right usually is synonymous with dragging right. Swiping a carousel or swimlane to the right causes it to continue scrolling for a while with simulated momentum.

Swiping right also can open a left-hand drawer or close a right-hand drawer.

Apple's Safari browser uses a swipe right to navigate like the back button. Google's Chrome browser uses a swipe right to delete browser tabs when in tab edit mode.

Pinch in/out

The pinch-in gesture is used to shrink or zoom out on objects physically (such as on a map view). Or you can perform a semantic zoom—zoom out or up one level in the hierarchy in a set of physically or conceptually nested structures.

The pinch-out gesture is used to expand or zoom in on objects physically (such as on a map view). Or you can perform a semantic zoom—zooming in or down one level in the hierarchy in a set of physically or conceptually nested structures.

Rotate

Rotate is a gesture employing the thumb and forefinger twisted clockwise or counterclockwise together on the touchscreen surface. This gesture can be used to actuate knob controls. But knobs probably should also support a horizontal or vertical drag action that starts on the knob as an alternative and more discoverable gesture. It can also be used to rotate objects, like a selection of pixels in an image editing app.

This gesture is somewhat awkward to carry out, given the anatomy of the human wrist. FiftyThree Inc.'s iOS app, Paper, uses this gesture to control Undo/Redo. Although this is

a novel approach, it seems inferior to the more standard Undo/Redo arrow icons from a usability standpoint.

Multifinger swipes

The various mobile OS's use various multifinger swipe gestures. For example, iOS supports an option that permits four-finger left/right swipes to switch between running apps.

On the whole, multifinger gestures are not very discoverable and when used in apps may interfere with OS-level gestures. They are best left unused, or reserved for specific needs.

Inter-App Integration

Modern smartphones, with their standalone app approach, have created a marvelous ecosystem in which users can easily add amazing functionality to their devices through an app store. This approach does have one Achilles heel: Standalone apps tend not to foster the useful integration of functionality and data between them. For example, the iPhone has a phone app, a contacts app, a calendar app, a messaging app, a memo app, and a reminders app. However, these apps are almost completely standalone and do not communicate with each other except in the most rudimentary ways.

The iPhone and other modern smartphones currently do a reasonable job of integrating the phone and contacts apps. When a call arrives, you can see the full name from the address book and, by tapping a button in the address book, you can dial it. However, this integration could be taken a step further. Clicking a name in an address book could give you a reverse chronological list (or set of lists) of all documents that are associated with that person: appointments, e-mails, phone calls from the log, memos including the caller's name, websites associated with the person, and so on.

Similarly, when an incoming call arrives, the phone could check your location (such as a cinema) or see if you are currently in a meeting that's on your calendar. It could automatically silence the ringer (and perhaps even send an "I'm busy and will call you back later" text message to the person calling you) unless the call is from someone on your VIP list of callers.

It's unfortunate that phone manufacturers haven't yet applied this kind of integration to the core suite of phone apps. However, some clever apps like IFTTT (If This Then That) do allow apps that participate in their service to be wired together with customizable rules that allow for some level of app integration (see Figure 19-38).
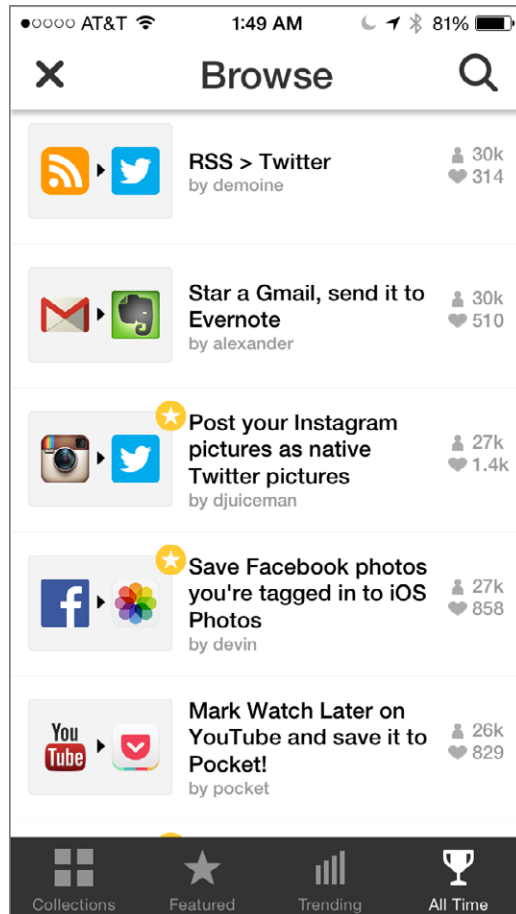
Figure 19-38: The IFTTT app lets users wire together apps by allowing them to specify output and input triggers, effectively allowing simple app integration.

For music production, Audiobus (see Figure 19-39) is an integration-oriented iOS app that allows other compatible iOS audio applications to route multiple input audio streams to multiple audio outputs. This effectively allows an entire virtual recording studio to exist within an iPhone or iPad.
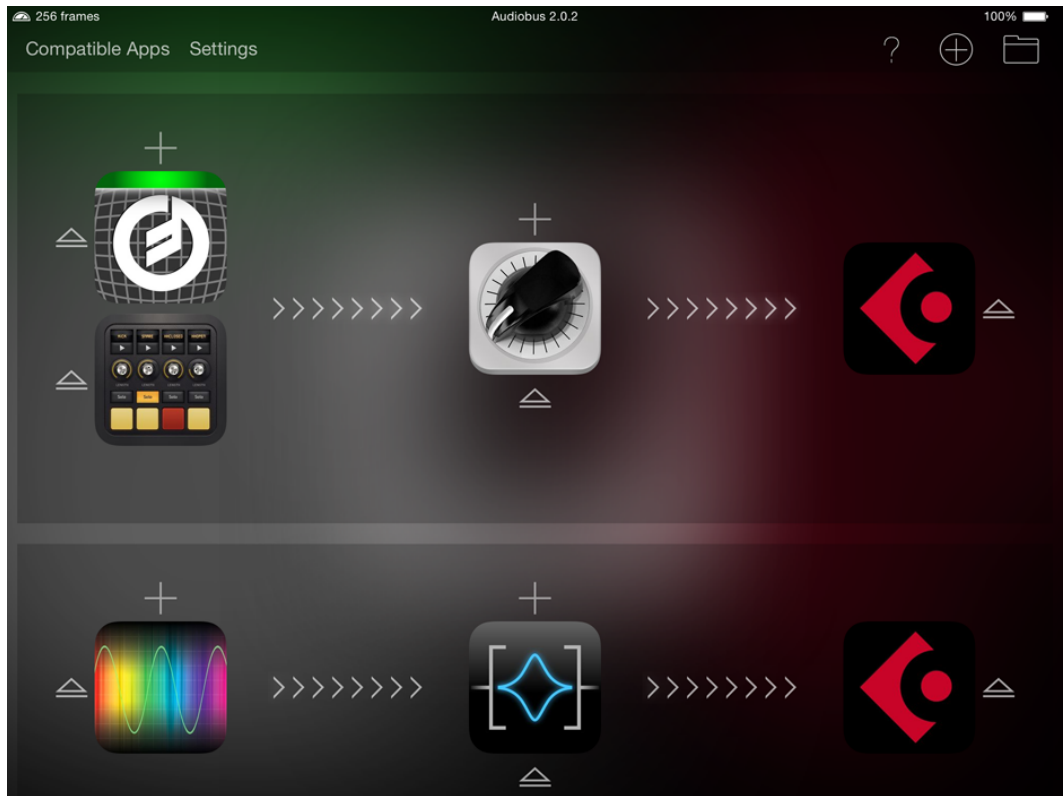
Figure 19-39: The Audiobus app allows users to chain together the audio streams from compatible running apps. Doing so supports input, output, and effects, allowing an entire virtual recording studio to exist within an iPad.

Other Devices

Unlike software running on a desktop computer or high resolution mobile device, interaction design for *device-embedded interfaces* requires special attention to creating an experience that coexists with the noise and activity of the real world happening around the product, combined with the typically limited screen and computational resources needed to render it. Kiosks and other embedded systems, such as TVs, microwave ovens, automobile dashboards, cameras, bank machines, and laboratory equipment, are unique platforms with their own opportunities and limitations.

General design principles

Embedded systems (physical devices with integrated software systems) involve some unique challenges that differentiate them from desktop systems, despite the fact that

they may include typical software interactions. Examples include smart appliances and medical informatics devices. When designing any embedded system, whether it is a smart appliance, kiosk system, or handheld device, keep these basic principles in mind:

- Don't think of your product as a computer.
- Integrate your hardware and software design.
- Let context drive the design.
- Use modes judiciously, if at all.
- Limit the scope.
- Balance navigation with display density.
- Minimize input complexity.

The following sections discuss each of these principles in more detail.

Don't think of your product as a computer

Perhaps the most critical principle to follow while designing an embedded system is that what you are designing is not a computer, even though its interface might be dominated by a computer-like bitmap display. Your users will approach your product with specific expectations about what the product can do (if it is an appliance or familiar handheld device) or with very few expectations (if you are designing a public kiosk). The last thing you want to do is bring all the baggage—the idioms and terminology—of the desktop computer world with you to a "simple" device like a camera or microwave oven. Similarly, users of scientific and other technical equipment expect quick and direct access to data and controls within their domain, without having to wade through a computer operating system or file system to find what they need.

Designers, especially those who have designed for desktop platforms, can easily forget that even though they are designing software, they are not always designing it for computers in the usual sense: devices with large color screens, lots of power and memory, full-size keyboards, and mouse pointing devices. Few, if any, of these assumptions are valid for most embedded devices. And, most importantly, these products are used in much different contexts than desktop computers. Idioms that have become accepted on a PC are inappropriate on an embedded device. "Cancel" is not an appropriate label for a button to turn off an oven, and requiring people to enter a "settings" mode to change the temperature on a thermostat is preposterous. Much better than trying to squeeze a computer interface into the form factor of a small-screen device is to see it for what it is and then figure out how digital technology can be applied to enhance the experience for its users.

Integrate your hardware and software design

Embedded systems often benefit from custom hardware. Desktop computers are meant to be all-purpose, but embedded systems are usually intended to help accomplish a very specific task. Due to cost, power, and form factor constraints, hardware-based navigation and input controls must often take the place of onscreen equivalents.

It is therefore critical to design the hardware and software elements of the system's interface—and the interactions between them—simultaneously, and from a goal-directed, ergonomic, and aesthetic perspective. Many of the best, most innovative digital devices of the last several decades, such as the TiVo and the original iPod, were designed from such a holistic perspective. Hardware and software combine seamlessly to create a compelling and effective experience for users, as shown in Figure 19-40. This doesn't occur often enough in the standard development process. Hardware engineering teams regularly hand off completed mechanical and industrial designs to the software teams, who must then accommodate them, regardless of what is best from the user's perspective.

Figure 19-40: A Cooper design for a smart desktop phone, exhibiting strong integration of hardware and software controls. Users can easily adjust volume/speakerphone, dial new numbers, and control playback of voicemail messages with hardware controls. They also can manage known contacts/numbers, incoming calls, call logs, voicemail, and conferencing features using the touchscreen and thumbwheel. Rather than attempting to load too much functionality into the system, the design focuses on making the most frequent and important phone features much easier to use. Note the finger-sized regions devoted to touchable areas on the screen and the use of text hints to reinforce the interactions.

Let context drive the design

Another distinct difference between embedded systems and desktop applications is the importance of environmental context. Although desktop applications sometimes can introduce contextual concerns, designers generally can assume that most software running on the desktop will be used on a computer that is stationary and located in a relatively quiet and private location. Although this is becoming less true as laptops gain both wireless capabilities and the power of desktop systems, it remains true that users will, by necessity of the form factor, be stationary and out of the hubbub even when using laptops.

Exactly the opposite is true for many embedded systems, which either are designed for on-the-go use (handhelds), in an active workspace (like an operating theater), or are stationary but in a location at the center of public activity (kiosks). Even embedded systems that are mostly stationary and secluded (like household appliances) have a strong contextual element. For example, a host juggling plates of hot food at a dinner party is distracted, not in a focused state of mind to navigate a cumbersome set of controls for a smart oven. Navigation systems built into a car's dashboard cannot safely use "soft keys" that change their meaning in different contexts, because the driver would be forced to take her eyes off the road to read each function label. Similarly, a technician on a manufacturing floor should not be required to focus on difficult-to-decipher equipment controls, because that kind of distraction could be life-threatening in some circumstances.

Thus, the design of embedded systems must match very closely the context of use. For handhelds, this context concerns how and where the device is physically handled. How is it held? Is it a one-handed or two-handed device? Where is it kept when not in immediate use? What other activities are users engaged in while using the device? In what environments is it used? Is it loud, bright, or dark there? How does the user feel about being seen and heard using the device if he is in public? We'll discuss some of these issues in detail a bit later.

For kiosks, the contextual concerns focus more on the environment in which the kiosk is being placed and also on social concerns. What role does the kiosk play in the environment? Is the kiosk in the main flow of public traffic? Does it provide ancillary information, or is it the main attraction itself? Does the architecture of the environment guide people to the kiosks when appropriate? How many people are likely to use the kiosk at a time? Are there sufficient numbers of kiosks to satisfy demand without a long wait? Is there sufficient room for the kiosk and kiosk traffic without impeding other user traffic? We touch on these and other questions shortly.

Use modes judiciously, if at all

Desktop computer applications are often rich in modes: The software can be in many different states in which input and other controls are mapped to different behaviors.

Tool palettes (such as those in Photoshop) are a good example. When you choose a tool, mouse and keyboard actions are mapped to a set of functions defined by that particular tool. When you choose a new tool, the behavior resulting from similar input changes.

Unfortunately, users are easily confounded by modal behavior. Because devices typically have smaller displays and limited input mechanisms, it is very difficult to convey what mode the product is in. Significant navigational work is often required to change modes. Take, for example, pre-smartphone mobile telephones. They often required navigation of seemingly countless modes organized into hierarchical menus. Most of these old-school mobile phone users only mastered the dialing and address book functionality but quickly got lost when they tried to access other functions. Even an important function such as silencing the ringer was often a daunting task.

When designing for embedded systems, it's important to limit the number of modes. Ideally, mode switches should result naturally from situational changes in context. For example, it makes sense for a smartphone to shift into telephone mode when an incoming call is received and to shift back to its previous mode when the call is terminated. If modes are truly necessary, they should be clearly accessible in the interface, and the exit path should also be immediately clear.

Limit the scope

Most embedded systems are used in specific contexts and for specific purposes. Avoid the temptation to turn these systems into general-purpose computers. Users will be better served by devices that enable them to do a limited set of tasks more effectively, than by devices that attempt to address too many disparate tasks in one place.

Many devices like smart phones and slate computers share information with desktop systems. It makes sense to approach the design of such systems as *satellites* of the desktop: The device is an extension, or satellite, of the desktop, providing key information and functions in contexts where the desktop system is unavailable. Scenarios can help you determine what functions are truly useful for such satellite systems.

Balance navigation with display density

Many devices are constrained by limited display real estate. Whether the underlying reasons are hardware cost, form factor, portability, or power requirements, designers must make the best use of the display technology available while meeting users' information needs. Every pixel, segment, and square millimeter of display is significant in the design of display-constrained embedded systems. Such limitations in display real estate almost always result in a trade-off between clarity of information displayed and complexity of navigation. By appropriately limiting the scope of functions, you can ameliorate this

situation somewhat, but the tension between display and navigation almost always exists to some degree.

Certainly it's best to completely flatten the information hierarchy if possible. But if not, you must carefully map out embedded systems' displays, and develop a simple and understandable hierarchy of information. Determine the most important information to get across, and make it the most prominent. Then, look to see what ancillary information can still fit on the screen. Try to avoid flashing between different sets of information by blinking the screen. For example, an oven with a digital control might display both the temperature you set it to reach and how close it is to reaching that temperature by flashing between the two numeric values. However, this solution easily leads to confusion about which number is which. A better solution is to display the temperature that the oven has been set to reach and, next to that, show a small bar graph that registers how close to the desired temperature the oven currently is. You must also leave room in the display to show the state of associated hardware controls or, better yet, use controls that can display their own state, such as hardware buttons with lamps or hardware that maintains a physical state (such as toggles, switches, sliders, and knobs).

Minimize input complexity

Almost all embedded systems have simplified inputs, not general-purpose keyboards or a desktop-style pointing device. This means that any input to the system—especially text input—is awkward, slow, and difficult for users. Even the most sophisticated of these input systems, such as touchscreens, voice recognition, handwriting recognition, and thumb-boards, are cumbersome in comparison to full-sized keyboards and mice. So it's important that input be limited and simplified as much as possible.

Kiosks, even though their screens are usually larger, should nonetheless avoid complex text input whenever possible. Public kiosks run an unfortunate risk of being a disease vector, so your first pass should try for noncontact inputs like voice, proximity switches, or non-contact gestural inputs. If more is needed, touchscreens can display soft keyboards if they are large enough; each virtual key should be large enough to make it difficult for the user to accidentally mistype. Touchscreens should also avoid idioms that involve dragging; single-tap idioms are easier to control and more obvious (when given proper affordance) to novice users on such a transient application.

Designing for single-purpose handheld devices

For single-purpose handheld devices (ones for which you are designing both the hardware and the software form factors and interfaces), you should keep in mind some additional considerations:

- Think about how the device will be held and carried. Physical models are essential to understanding how a device will be manipulated. The models should at least reflect the device's size, shape, and articulation (flip covers and so on). They are more effective when weight is also taken into account. These models should be employed by designers in context and key path scenarios to validate proposed form factors. Labels on buttons have very different contextual needs, depending on where and when they will be used. For example, the labels on a package-delivery tracking tool don't need to be backlit like those on a cell phone or TV remote control.

- Determine early on whether the device or application will support one-handed or two-handed (or even no-handed) operations. Again, scenarios should make it clear which modes are acceptable to users in various contexts. It's okay for a device that is intended primarily for one-handed use to support some advanced functions that require two-handed use, as long as they are needed infrequently.

- Avoid use of pluralized and pop-up windows. Floating windows typically have no place on small, low-resolution screens. In this regard, interfaces should resemble sovereign-posture applications, using the full screen real estate. Modeless dialogs should always be avoided. Modal dialogs and errors should, whenever possible, be replaced using the techniques discussed in Chapter 15.

Designing for kiosks

On the surface, kiosks may appear to have much in common with desktop interfaces: large, colorful screens and reasonably beefy processors behind them. But as far as user interactions are concerned, the similarity ends there. Kiosk users, in comparison with sovereign desktop application users, are at best infrequent users of kiosks and, most typically, use any given kiosk only once. Furthermore, kiosk users will have either a specific goal in mind when approaching a kiosk or no readily definable goal at all. Kiosk users typically don't have access to keyboards or pointing devices, and often they would be unable to use either effectively even if they did. Finally, kiosk users typically are in a public environment, full of noise and distractions, and may be accompanied by others who will be using the kiosk in tandem with them. Each of these environmental issues has a bearing on kiosk design (see Figure 19-41).

Transaction versus exploration

Kiosks generally fall into two categories: transactional and explorational. Transactional kiosks provide a tightly scoped transaction or service. These include bank machines (ATMs) and ticketing machines such as those used in airports, train and bus depots, and some movie theaters. Even gasoline pumps and vending machines can be considered a simple type of transactional kiosk. Users of transactional kiosks have specific goals in mind: to get cash, a ticket, a Tootsie Roll, or a specific piece of information. These users have no interest in anything but accomplishing their goals as quickly and painlessly as possible.
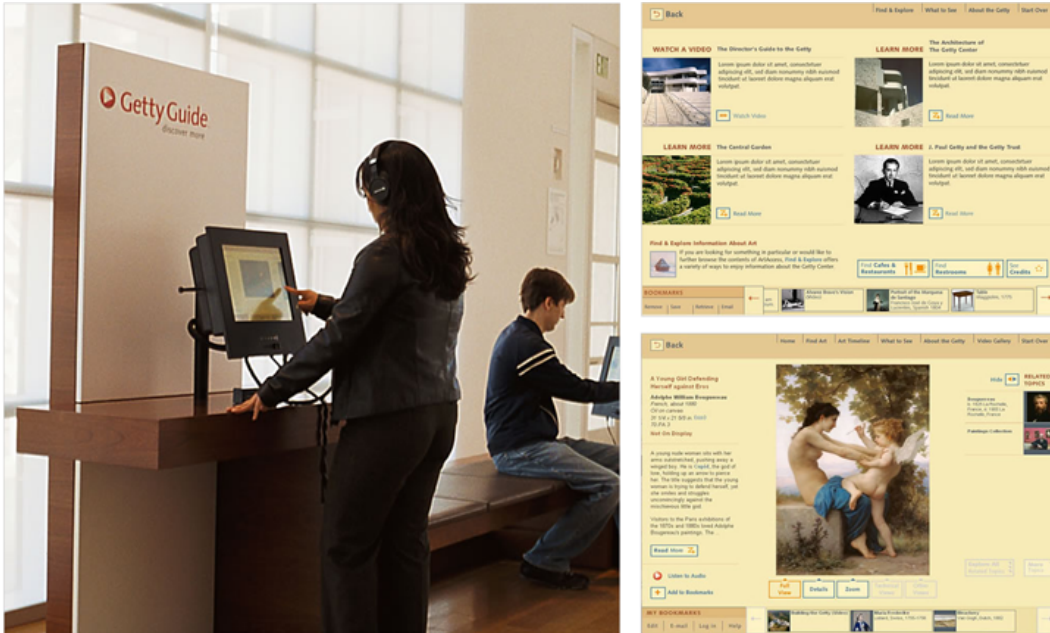
Figure 19-41: The GettyGuide, a system of informational kiosks at the J. Getty Center and Villa in Los Angeles, designed by Cooper in collaboration with the Getty and Triplecode

Explorational kiosks are most often found in museums or as an information display in a mall. Educational and entertainment-oriented kiosks typically are not a main attraction, but provide additional information and a richer experience for users who have come to see the main exhibits. (A few museums, such as science museums and the Experience Music Project in Seattle, have interactive kiosks that can be exhibits.) Explorational kiosks are somewhat different from transactional kiosks in that users typically have open-ended expectations when approaching them. They may be curious, or want to be entertained or enlightened, but they may not have any specific end goals in mind. (On the other hand, they may also be interested in finding the café or the nearest restroom, which are goals that can be supported alongside the more open-ended experience goals.) For explorational kiosks, the act of exploring must engage the user. Therefore, not only must the kiosk's interface be clear and easy to navigate, but it also must be aesthetically pleasing and visually (and possibly audibly) exciting to users. Each screen must be interesting in itself and also should encourage users to further explore other content in the system.

Interaction in a public environment

Transactional kiosks, as a rule, require no special enticements to attract users. However, they do need to be placed in an optimal location to be obviously visible and to handle the flow of user traffic they will generate. Use wayfinding and sign systems in conjunction

with these kiosks for maximum effectiveness. Some transactional kiosks, especially ATMs, need to take into account security issues: If their location seems insecure, users will avoid them or consider them risky. Architectural planning for transactional kiosks should occur at the same time as the interaction and industrial design planning.

As with transactional kiosks, place explorational kiosks carefully, and use wayfinding systems in conjunction with them. They must not obstruct any main attractions and yet must be close enough to the attractions to be perceived as connected to them. There must be adequate room for people to gather, because exploration kiosks are more likely to be used by groups (such as families). A particular challenge lies in choosing the right number of kiosks to install at a location. Companies employing transactional kiosks often engage in user flow research at a site to determine optimum numbers. People don't linger long at transactional kiosks, and they are usually more willing to wait in line because they have a definite end goal in mind. Explorational kiosks, on the other hand, encourage lingering, which makes them unattractive to onlookers. Because potential users have few expectations of the contents of an explorational kiosk, it becomes difficult for them to justify waiting in line to use one. It is safe to assume that most people will approach an explorational kiosk only when it is vacant.

When designing kiosk interfaces, carefully consider the use of sound. Explorational kiosks seem like a natural for the use of rich audible feedback and content, but volume levels should not encroach on the experience of the main attraction such kiosks often support. Audible feedback should be used sparingly in transactional kiosks, but it can be useful, for example, to help remind users to take back their bank card or the change from their purchases.

Also, because kiosks are in public spaces, designing to account for the needs of differently abled users is especially important. For more about designing for accessibility, see Chapter 16.

Lastly, as mentioned above, public objects can get dirty and become vectors for disease. Efforts should be taken to see if the kiosk can be made non-contact and still assist users with their goals.

Managing input

Most kiosks use either touchscreens or hardware buttons and keypads that are mapped to objects and functions on the screen. In the case of touchscreens, the same principles apply as for other touchscreen interfaces:

- Make sure that your click targets are large enough. Touchable objects should be large enough to be manipulated with a finger, high contrast, colorful, and well separated on the screen to avoid accidental selection. A 20mm click target is a typical minimum if

users will be close to the screen and not in a hurry. This size should be increased for use at arm's length or when in a rush. A good low-tech trick to make sure that your click targets are large enough is to print the screens at actual size, ink your fingertip with an ink pad, and run through your scenarios at a realistic speed. If your fingerprints overlap the edges of your click targets, you probably should increase their size. It's messy, but incontrovertible.

- Use soft-keyboard input sparingly. It may be tempting to use an onscreen keyboard to enter data on touchscreen kiosks. However, this input mechanism should be used only to enter very small amounts of text. Not only is it awkward for the user, but it also typically results in a thick coating of fingerprints on the display.

- Avoid drag and drop. This gesture can be difficult for users to master on a touchscreen, making it inappropriate for kiosk users who will never spend the necessary time to master demanding interaction idioms. Scrolling of any kind should also be avoided on kiosks except when absolutely necessary.

Some kiosks use hardware buttons mapped to onscreen functions in lieu of touchscreens. As in handheld systems, the key concern is that these mappings remain consistent, with similar functions mapped to the same buttons from screen to screen. These buttons also should not be placed so far from the screen or arranged spatially so that the mapping becomes unclear. (See Chapter 12 for a more detailed discussion of mapping issues.) In general, if a touchscreen is feasible, you should strongly consider it in favor of mapped hardware buttons.

Designing for 10-foot interfaces

Television-based interfaces (also called 10-foot interfaces) such as TiVo and most cable and satellite set-top boxes rely on user interaction through a remote control. Users typically operate it when they are sitting across the room from the television. Most remote controls use inexpensive infrared light to communicate, so unless the remote control uses more expensive radio-frequency technology, this also means that the user needs to point the remote at the TV and set-top boxes. All of this makes for challenges and limitations in designing effective information display controls for system navigation and operation.

It turns out that lists, grids, carousels, and swimlanes all map reasonably well to 10-foot interfaces, because D-pad navigation consists of up-down and left-right movements. This is good news for designers of services like Netflix, which live on both mobile devices and set-top boxes. Such interfaces can often be adapted across these platforms with only minor adjustments. As discussed in Chapter 17, Microsoft's Metro design language successfully spans desktop, mobile, and set-top platforms.

Here are some additional considerations to keep in mind for 10-foot user interfaces:

- Use a screen layout and visual design that can be easily read from across the room. Even if you think you can rely on high-definition television (HDTV) screen resolutions, your users will not be as close to the TV screen as they would be to, say, a computer monitor. This means that text and other navigable content needs to be displayed in a larger size, which in turn dictates how screens of information are organized.

- Keep onscreen navigation simple. People don't think about their TV like they do a computer, and the navigation mechanisms provided by remotes are limited. Therefore, the best approach is one that can be mapped easily to a five-way (up, down, left, right, and select) controller. There may be room to innovate with scroll wheels and other input mechanisms for content navigation. But these will likely need to be compatible with other set-top devices in addition to yours (see the next point), so take care in your design choices. In addition, visual wayfinding techniques are important for ensuring ease of use. These include color-coding screens by functional area and providing visual or textual hints about what navigational and command options are available on each screen. TiVo does a particularly good job of this.

- Keep control integration in mind. Most people hate the fact that they need multiple remotes to control all the home entertainment devices connected to their TV. By enabling control of commonly used functions on other home entertainment devices besides the one you are designing for (ideally with minimal configuration), you will be meeting a significant user need. This means that your product's remote control or console needs to broadcast commands for other equipment and may need to keep track of some of the operational state of that equipment as well. Logitech's line of Harmony universal remote controls does both of these things, and the remotes are configured via a web application when connected to a computer via USB.

- Keep remote controls as simple as possible. Many users find complex remote controls daunting, and most functions available from typical home entertainment remotes remain little used. Especially when remote controls take on universal functionality, the tendency is to cram them with buttons. It's not unusual to see 40, 50, or even 60 buttons on a universal remote.

  One way to mitigate this is to add a display to the remote. This can allow controls to appear in context, provide additional information to the user, so fewer buttons are needed at any one time. These controls can be accessed via a touchscreen or via soft-labeled physical buttons that lie adjacent to the screen. Each of these approaches has drawbacks. The vast majority of touchscreens do not provide tactile feedback, so the user is forced to look away from his or her TV to actuate a touchscreen control. Soft-labeled buttons address this problem but add back more buttons to the remote's surface. The addition of a display on your remote may also tempt you to allow navigation to multiple "pages" of content or controls on the display. This may be warranted sometimes, but any design choice that divides the user's attention between two displays (the TV and the remote) runs the risk of creating user confusion and annoyance.

- Focus on user goals and activities, not on product functions. Most home entertainment systems require users to understand the topology and system states to use it effectively. For example, to watch a movie, the user may need to know how to turn on the TV, how to turn on the DVD player, how to switch input on the TV to the one that the DVD player is connected to, how to turn on surround sound, and how to set the TV to widescreen mode. Doing this may require three separate remote controls, or half a dozen button presses on a function-oriented universal remote. Remote controls like Logitech's Harmony take a different approach: organizing the control around user activities (such as "watch a movie") and building a detailed understanding of the user's entertainment setup through a very usable setup wizard. While this is much more complex to develop, it is a clear win for the user if implemented well.

Designing for automotive interfaces

Automotive interfaces, especially those that offer sophisticated navigation and entertainment functionality, (called "telematics" in the field itself) have the particular challenge of driver safety. Even mildly complex, confusing, or distracting interactions can risk lives. Such systems require significant design effort and usability validation to avoid such issues. Add to the risk factor the spatial limitations of the automobile dashboard, center console, and steering wheel, and you have a very particular challenge.

- Minimize the amount of time hands are off the wheel. Commonly used navigation controls such as play/pause, mute, and skip/scan should be available on the steering wheel (driver use) as well as on the center console (passenger use).

- Enforce consistent layout from screen to screen. If you maintain a consistent layout, the driver will be able to keep his bearings between context shifts.

- Use direct control mappings when possible. Controls with labels on them are better than soft-labeled controls. Touchscreen buttons with tactile feedback are also preferable to soft labels with adjacent hard buttons, because again this requires fewer cognitive cycles from the driver operating the system to make the mapping.

- Choose input mechanisms carefully. It's much easier for drivers to select content from knobs than from a set of buttons. There are fewer controls to clutter the interface, knobs protrude and therefore are easier to reach, and (when properly designed) they afford both rough and fine controls in an elegant and intuitive way.

- Differentiate the physical design of different controls clearly so that they can be managed as much as possible by touch.

- Use very strong contrasts in the visual design of displays and a very shallow visual hierarchy to enable quick glances for information.

- Keep mode/context switching simple and predictable. With its reviled iDrive system, BMW mapped most of the car's entertainment, climate control, and navigation into

a single control that was a combination knob/joystick. The idea was to make things simple. But by overloading the control so extensively, BMW created a danger for users by requiring them to navigate an interface in order to switch contexts and modes. Modes (such as switching from CD to FM or from climate control to navigation) should be directly accessible with a single touch or button press. And the location of these mode buttons should be fixed and consistent across the interface.

- Provide audible feedback. Audible confirmations of commands help reduce the need for the driver to take his eyes off the road. However, you should ensure that this feedback is itself not too loud or distracting. For in-car navigation systems, verbal feedback highlighting driving directions can be helpful. This is true as long as the verbal help (such as turning instructions and street names) is delivered early enough for the driver to react to it in time. Speech input is another possibility, using spoken commands to operate the interface. However, the automobile environment is noisy. It is unclear whether verbalizing a command, especially if it needs to be repeated or corrected for, is any less cognitively demanding than pressing a button. While this kind of feature makes for great marketing, we think the jury is still out on whether it makes for a better or safer user experience in the automobile.

Designing for audible interfaces

Audible interfaces, such as those found in voice message systems and automated call centers, involve some special challenges. Navigation is the most critical issue, because it is easy to get lost in a tree of functionality with no means of visualizing where you are in the hierarchy. Also, bad phone tree interactions are a common way to erode an otherwise strong brand identity. (Almost all voice interfaces are based on a tree, even if the options are hidden behind voice recognition, which introduces a whole other set of problems.)

The following are some simple principles for designing usable audible interfaces:

- Organize and name functions according to user mental models. This is important in any design but is doubly important when functions are described only verbally and only in the context of the current function. Be sure to examine context scenarios to determine what the most important functions are, and make them the most easily reachable. This means listing the most common options first.

- Always signpost the currently available functions. After every user action, the system should restate the currently available activities and how to invoke them.

- Always provide a way to go back one step and to return to the top level. After every action, the system should tell the user how to go back one step in the function structure (usually up one node in the tree) and how to get to the top level of the function tree.

- Always provide a means to speak with a human. If appropriate, the interface should give the user instructions on how to switch to a human assistant after every action, especially if the user seems to be having trouble. Some more sophisticated systems are trained to hear user stress and anger (parsing for curse words is common) and automatically responding by connecting to a human.

- Give the user enough time to respond. Systems usually require verbal or telephone keypad entry of information. Testing should be done to determine an appropriate amount of time to wait. Keep in mind that phone keypads can be awkward and very slow for entering textual information.

DESIGNING FOR THE WEB

The advent of the World Wide Web was initially both a boon and a curse for interaction designers. For perhaps the first time since the invention of graphical user interfaces, corporate decision makers began to understand and adopt the language of user-centered design, and the term "user experience" came into vogue among business executives far and wide. On the other hand, the limitations and challenges of web interactivity, which are a result of its historical evolution, set back interaction design by nearly a decade.

When the first edition of this book was published in August 1995, the web was just beginning to emerge from its roots in academic and scientific computing. At that time, the web was really only good for publishing and reading text documents that had a few links and inline images (form elements were introduced in HTML 2.0 a few months later). When the second edition of this book was published in 2003, the consumer and enterprise web, including corporate intranets, had come into being (and had survived a major industry implosion a few years earlier) but was still highly limited in terms of interactivity. There were strong conventions for navigation and basic data entry, but doing anything more sophisticated was a miracle.

Even following the dotcom bust, the promise of the web was apparent to everyone. The industry was flooded with fresh design school graduates, traditional graphic designers, and young enthusiasts who saw the web as an exciting and lucrative opportunity to create compelling communication (and commerce) through new forms of interactive visual expression. The biggest challenges involved working around the severe limitations of the medium. Creating a user experience with even a rudimentary level of interactivity, as well as visual and logical organization, was a real challenge for early web designers.

But by the time the third edition of *About Face* came out in 2007, more powerful web technologies had come into common use. Things such as HTML5, CSS3, and AJAX had enabled the rise of rich Internet applications (RIAs). These featured much more sophisticated UI capabilities, including drag and drop, the ability to stream data into UI elements, and much more robust screen structuring capabilities. Browser-based user interfaces were starting to approach parity with many native desktop capabilities. But in many areas of the industry, Microsoft .NET native Windows applications were still the dominant paradigm for software creation and delivery.

As we publish this fourth edition of *About Face* in 2014, the landscape has largely changed. With the rise of GitHub, the open source movement has created an impressive body of highly sophisticated HTML5 user-interface components that are highly capable and largely interoperable (such as the Bootstrap and jQuery ecosystems). Thanks to investment by companies such as Google and Apple, web browsers' ability to quickly render and process HTML and JavaScript has become very effective, and the deeper plumbing of the web stack has also become highly capable.

Deploying software-based experiences over the web has many advantages. It allows for continuous deployment, and therefore continuous improvement. Network-based applications can be much better suited to the social and collaborative way we live and work. And we can accommodate much more transient, grazing usage. Installing software (and keeping it up to date) is a commitment we don't always want people to have to make in order to interact with our product or service.

All of this adds up to a situation where there are very few experiences that a designer can dream up that can't be built to perform in a modern-day web browser. It is increasingly becoming the case that we are building native applications only to support sophisticated authoring tools (for example, graphic design, 3D modeling, video editing, presentation, and code editors). Furthermore, the web has become the most important and popular channel that people use to communicate and that companies use to interact with their customers.

This means that the quality of web experiences is incredibly important, and the increased ability to deliver complex behavior in a browser demands application-quality interaction design. The visual designer's focus on look and feel and the information architect's focus on content structure are insufficient to create effective and engaging user experiences with this new generation of the web.

It is now easy to browse GitHub for ready-coded UI components that include many interaction-design-friendly features (such as rich visual modeless feedback). But even with all these rich capabilities, we are still left with the important questions of exactly what will suit the needs and desires of people interacting with a product or service, and how to create a coherent, useful experience from these building blocks.

In many ways, it's difficult to generalize about designing for the web, because it has become a huge place. In different tabs in the same browser window, you might be looking at mass media, enterprise software, e-commerce, and social networking sites.

Even though people clearly have different expectations for different kinds of web experiences, they must rely on convention to orient themselves to each website or application they arrive at, especially those they may be seeing for the first time or may visit only occasionally. While these conventions are constantly evolving, they are also largely tied to the nature of the medium. They are important for the interaction designer to consider when creating a browser-based experience.

This chapter looks at the most important of these considerations. We should also say that because web design is an area of rich thought, there is a sizeable body of work worth engaging with. In particular, Steve Krug's *Don't Make Me Think, Revisited* (New Riders, 2014) and Louis Rosenfeld and Peter Morville's *Information Architecture for the World Wide Web* (O'Reilly, 2006) cover the essential elements of web design in a clear and straightforward manner. The website called A List Apart is also a great resource, even if it often has a more technical focus.

Page-Based Interactions

The fundamental character of the medium that is the web is shaped in a huge way by the concept of the *page*. From its inception, the whole technology stack has been formed around pages. Developments like AJAX and MVC frameworks for the web allow us to get pretty fancy with how a page is structured and how pages are related to each other. Many of the most important conventions and considerations for designing web experiences are tied to the concept of the page.

It's important for designers who work in both native application software (either desktop or mobile) and browser-based software to be aware of and intentional about the medium they're designing for. Native applications are usually constructed in terms of *screens* or *views*. Although they are analogous to the page, there are some meaningful and important differences between the two ways of structuring an experience, which we'll discuss in this chapter.

Navigation and wayfinding

First and foremost, while some navigation between views may occur in a native application, this is usually nothing compared to the amount of navigation in a typical web application. One way of thinking about it is that a native application usually has a limited number of spaces or modes that the user can be in, and different pieces of content

can populate each of those spaces or modes. On the other hand, on the web, each piece of content typically has its own place (or, rather, URL), and the trick is figuring out how to help people get to the content they want.

This leads us to the field of information architecture. In the early days of the commercial web, people designing and building websites recognized that a new design issue resulted from supporting numerous hyperlinked pages: the challenge of organizing and structuring content across pages in a meaningful way. A new breed of designer, the information architect, built a discipline and practice to address the nonvisual design problems of logical structure and flow of content.

It's beyond the scope of this book to deeply address the topic of information architecture. But this phenomenon—that web experiences typically are constructed of numerous diverse pages with some sort of logical organization—also has given interaction designers the challenge of creating meaningful interactions related to navigation.

Primary navigation

Since the early days of the commercial web, the term *primary navigation* has signified how the user gets to the major areas or sections of a website or application. For some time, almost every website and application has included persistent links along the top or left side.

Top navigation is a superior approach in most cases (see Figure 20-1). Side navigation makes the page crowded and occupies the page's visual entry point, forcing the user to scan past it to read content. The biggest limitation of top navigation—that it can accommodate only a few items of limited length—may actually be one of its greatest benefits. Forcing designers to reduce the number of major areas of a website or application—and to keep the titles short and punchy—usually has a better chance of resulting in something that is comprehensible and useful to users.
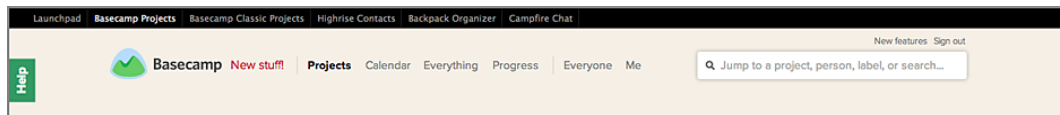
Figure 20-1: Basecamp illustrates the common practice of placing primary navigation on the top of the page. The topmost black bar allows the user to switch between different applications, the navigation adjacent to the Basecamp logo provides access to the major areas of Basecamp itself.

As with most rules of thumb, there are exceptions. If you have a large heterogeneous content space, reducing to items that can fit on a horizontal bar can result in navigation terms that are meaninglessly abstract to your users. The biggest advantages of left-side navigation are that items can be longer, there can be more of them, and they are easier

for users to scan because they are left-aligned. Amazon, which is well known for using analytics to optimize its page designs, and which sells almost everything known to man, currently uses left-hand navigation for product categorization on some pages. But on every page except the home page, this navigation is hidden until the user mouses over Show by Department to reveal it (see Figure 20-2).
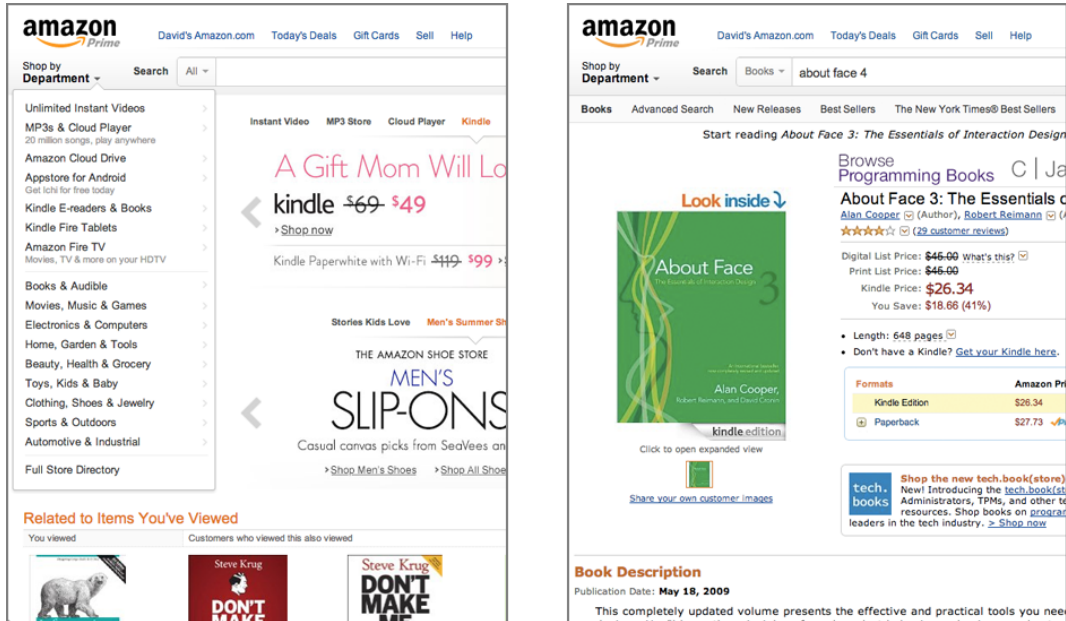
Figures 20-2: Amazon uses an approach to side-navigation where it is displayed to the user on the home page, but requires a mouseover to access on all other pages.

This brings us to another important topic in web design: dynamically hiding and showing navigation controls that depend on the user's location in the system, and even where he or she is on a page. An increasingly popular and successful pattern is to keep this top navigation bar locked to the top of the browser window when the user scrolls. Branding and other elements are minimized so that the bar takes up less screen real estate and visual attention as the user engages with content lower down the page (see Figure 20-3).

When considering the best approach for primary navigation, it's important to consider people using mobile web browsers. If this is a vital platform for you (and, in this day and age, in most cases, it should be), make sure you think through how well your navigation works on smaller screens. One common and utilitarian approach is not to show the navigation persistently and to reveal it only when the user clicks a menu or "hamburger icon" control (three stacked horizontal lines).
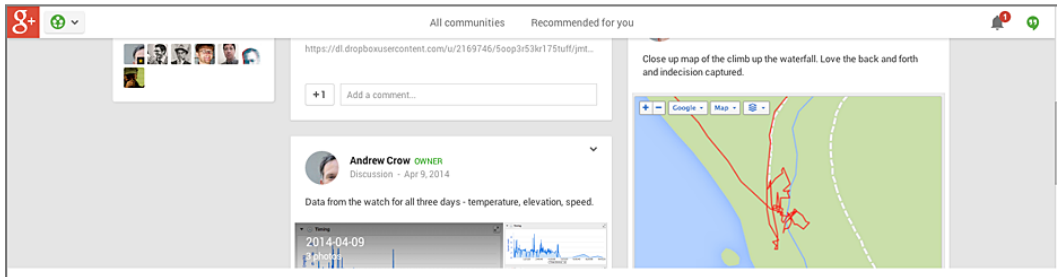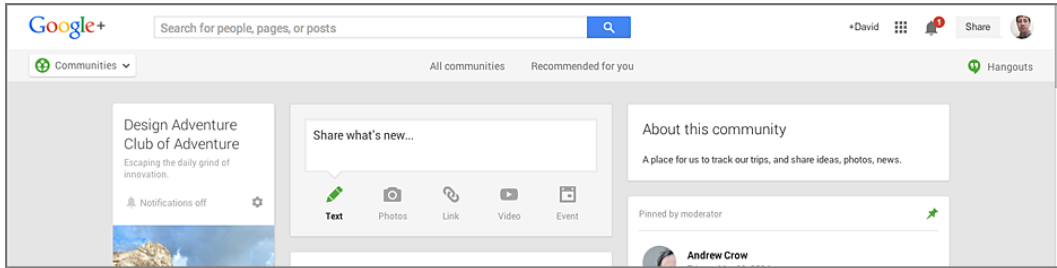
Figure 20-3: The header of Google+ is persistent, but makes itself smaller when the user scrolls down the page

DESIGN PRINCIPLE

Use persistent headers to maintain context.

A healthy debate is currently under way about whether most users understand the hamburger icon. At least one statistically significant study has shown that, for at least some users, the word "menu" performs better than the hamburger icon.[1] Figure 20-13, in the later section on the mobile web, shows how the Boston Globe employs a responsive approach to a top navigation, reducing the number of navigation items for smaller browser windows, ultimately shrinking to a single "sections" menu for smartphone-sized screens.

Secondary navigation and beyond

Often the entire information space of an application (or suite of applications) cannot be meaningfully navigated to from a handful of top-level links. Even though users will almost certainly search past this point, the content may bear secondary levels of persistent navigation, and perhaps additional levels beyond that. Expert users of sovereign applications may be able to memorize navigation paths three or more layers deep. But in our experience, most intermediate or novice users struggle to find information if it's buried in a three-level hierarchy. While a good search mechanism may mitigate this

problem, it's best to try to keep your navigation space as flat and compact as possible to make it easier for users to create a useful mental model of how your application is organized.

There are several basic yet effective mechanisms for secondary and additional levels of navigation. You can add a left-hand menu or a second stripe of horizontal navigation links (if you're using this approach for your primary navigation; see Figure 20-4).
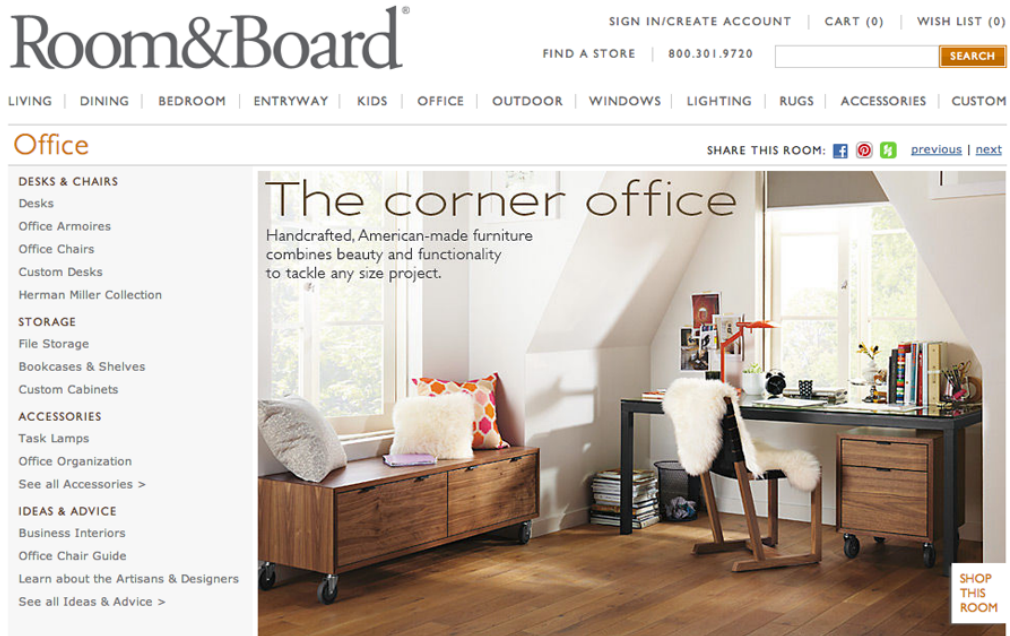
Figure 20-4: Room & Board's website uses a classic left-hand secondary navigation to access sub-pages in a major section of the site.

A useful approach to secondary navigation is *fat navigation*. When the user clicks the primary navigation, it expands to reveal a much larger set of content areas choices (see Figure 20-5). This type of navigation can be effective because it flows naturally from interactions the user may have with the primary navigation. And because it is modal and temporary, it can be much more generous with the amount of space it uses.

To better reinforce the user's mental model of site organization, it's important to provide persistent feedback about where he or she currently is, regardless of the depth of your navigation structures. Two common ways to do this are through visual feedback in the navigation elements themselves, and through *breadcrumbs* —a sequence of links that represent the user's path through the site hierarchy (see Figure 20-6).
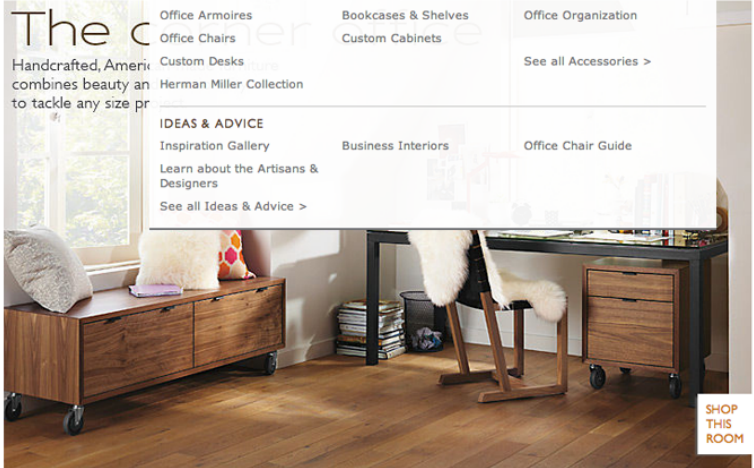
Figure 20-5: Hovering over Room & Board's primary navigation provides easily accessible links to sub-pages, without requiring the user to navigate to the office section page first.

Figure 20-6: When you're looking at the page for a desk on the Room & Board website, you can see where you are in the site and navigate back up using the breadcrumbs.

On some sites, clicking each breadcrumb "step" opens a pop-up menu of lateral links, enabling users to navigate more easily to different parts of the site hierarchy without as many clicks—a feature borrowed from recent Windows OS file browser interfaces.

Content navigation

Another important type of navigation is the navigation of content such as photos and articles. These items often are numerous and subject to change, and the relationships between them often are associative, rather than strictly linear or hierarchical. These facts create a number of navigation challenges and patterns.

Most commonly, items are presented in listings of some sort—sequences of headlines and blurbs for articles, and galleries for photographs. Contemporary designs for these listings are often inspired by the "feed" format, popularized by blogs and social media like Twitter and Facebook.

Because some items may be newer, more important, or more likely to be interesting to the audience, it's also useful to highlight featured content. This can be done by using more prominent typography, by changing the size and position on the page, or by using a carousel that cycles through features in a more visual format.

It is common for content to be organized in multiple ways (such as by topic, author, or date published) and for users to want to use one or more of those organization schemes to find content they're looking for. In these cases, it can be desirable to expose multiple navigation schemes to browse content, or make use of the faceted search techniques discussed in the next section.

Searching

One of the most important navigation methods on the web is searching. From our observations and a number of studies, it's clear that although search algorithms continue to improve, most people are not very good at forming queries to find what they're looking for. The idea that Google has somehow trained people to search instead of using navigation is largely untrue.[2]

What this means is that an effective search pattern for your website or application should help users go from their initial search term to a page that contains what they're

looking for. There are a number of good strategies for doing this. Sometimes using several of them in succession helps your audience find what they're looking for. Chapter 19 discusses these strategies and their variants in the context of mobile apps, but these concepts apply equally well on the web, as we'll see here.

One of the most successful innovations in searching has been *auto-complete*, also known as *type ahead*. When the user types in his or her search terms, a number of choices for complete search terms are presented. These can be based on previous searches (as Google does) or actual results (the Spotlight search function in Apple OS X). Auto-complete greatly increases the chances that the user will enter a search term that is likely to have a meaningful result set (see Figure 20-7).
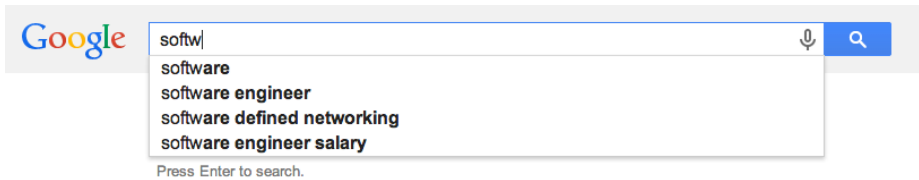
Figure 20-7: Google Search's auto-complete provides a list of expanded search terms based on what the user has already typed into the search field.

Disambiguation, or *auto-suggest* is another tool Google has normalized as part of searching. As you can see in Figure 20-8, if the searcher types a word that is spelled similarly to a more commonly searched word (or, more often, mistypes or misspells the word they really meant to search for), Google displays a list of suggestions along with the results. It also provides a link to the top suggestion as part of the results.
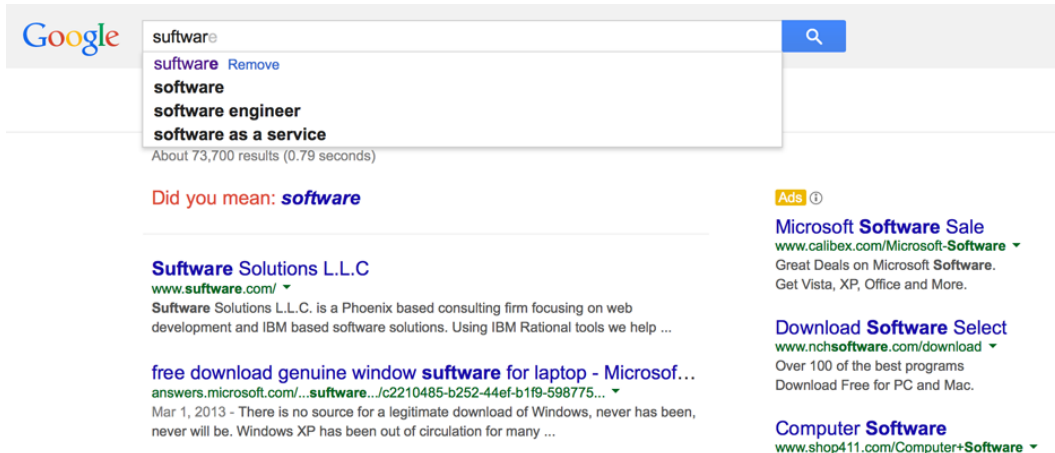
Figure 20-8: Google Search also supports auto-suggest, which provides a list of search terms based on fuzzy matching based on what the user has typed, in essence allowing the search box to auto-correct spelling errors.

That said, even if the user forms his search terms in a productive way, he still may have a large number of items to look through. This is where *faceted search* — which allows users to specify the attributes of the item they are looking for—can be really useful (see Figure 20-9).

Allowing users to narrow their search in a structured way helps them form a query that specifies precisely what they're looking for. An effective faceted search mechanism should provide users some visibility into the characteristics of the set of items they're searching, as well as give them ideas about how to make the result set small enough to efficiently find the desired item. Chapter 14 discusses some related approaches for attribute based sorting and filtering.
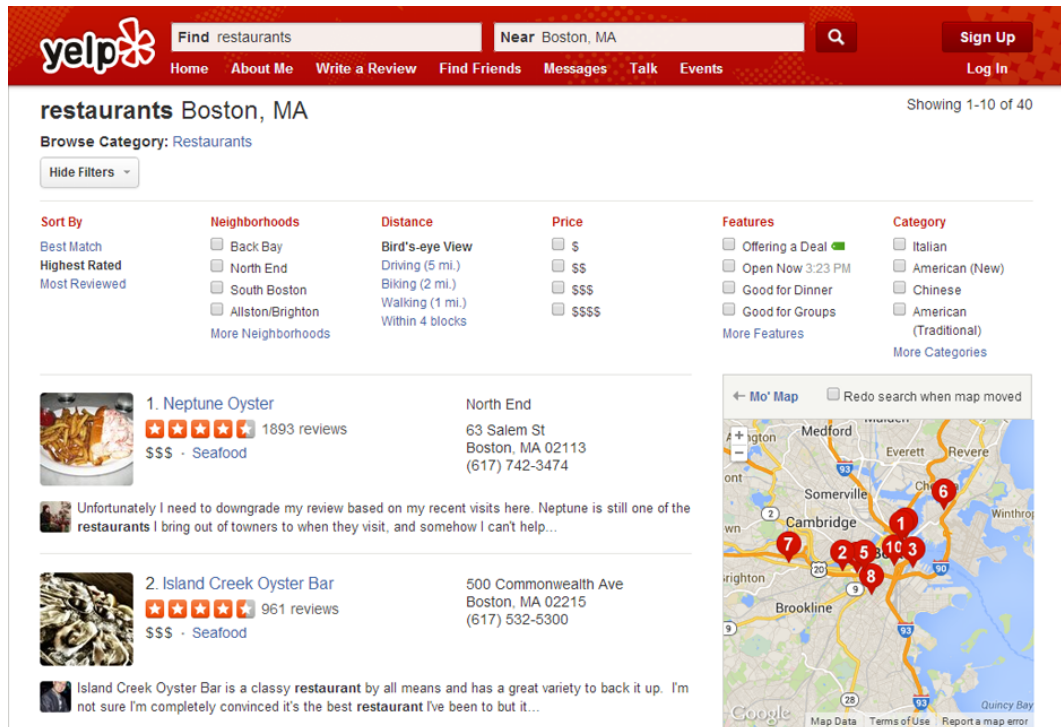
Figure 20-9: Yelp provides effective faceted search mechanisms, allowing users to quickly fine-tune a search

Categorized suggestions is yet another method of speeding the user to relevant results when a search term is applicable across many different categories or domains. This is achieved by the system offering a list of suggestions, each of which scopes the search to a particular category. Amazon, with its dozens of retail departments, makes good use of categorized suggestions (see Figure 20-10).
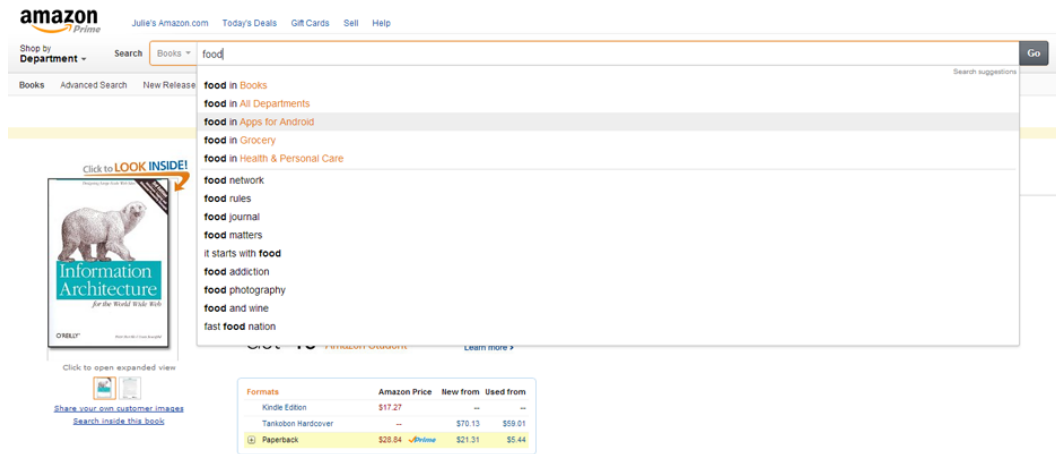
Figure 20-10: Amazon makes good use of categorized suggestions in its main search box, which allows both explicit scoping via a dropdown to the left of the search field, and categorized suggestions once you start typing.

Scrolling

One of the most obvious yet significant features of the page-based web experience is the prevalence of scrolling. Screen/view-based native experiences often have a fixed screen layout with multiple panes. Although each of them may have scrolling content, it is almost always good design in a native application to have key functions persistently available. Even though people are used to seeing a document they're editing scroll, they'd most likely be surprised and dismayed if the toolbar scrolled along with it.

With web experiences, on the other hand, critical information and functions often must be accessed through scrolling. Web designers have long been concerned about the "fold"—the vertical position on the page below which content isn't visible upon page load. However, the rapid rise in the number of touchscreen interactions and things like Apple's Magic Trackpad have made scrolling much more natural and expected than it used to be with fiddly scrollbars.

Furthermore, the growth in using mobile devices to access web content has given rise to the importance of *responsive design*, in which a web page is designed to format itself appropriately for the size of the user's screen (more on this in the next section). Because this responsiveness narrows content areas on smaller screens, it's much more challenging to try to control what fits on the screen or goes above the fold.

The result of all this is that one of the most important aspects of successful web design is to engage the user to progress through content or functionality as he or she scrolls down a page. This doesn't apply only to editorial-type content, but also to more highly interactive capabilities. One example is the popular "parallax" effect, in which different onscreen elements respond to the user's scrolling at different speeds.

Much of enterprise software is catching up with the era of touch interaction. Nevertheless, there's a real opportunity to create more seamless engaging interactions by bringing together content and interactive elements that traditionally have been broken into numerous pages and put them into a longer, scrollable page.

DESIGN PRINCIPLE   *Make scrolling an engaging experience.*

One method is to create an effective visual rhythm through the use of white space and a strong typography system. You also should be generous with font and control sizes to improve usability for touch users and to improve *scanability* when the user scrolls. It's also important to help users stay oriented as to where they are on the long page. The Nest website (Figure 20-11) is structured as a number of long scrolling pages. For example, the "Life with Nest" page is a timeline of how Nest programs itself over time, and as the user scrolls down, the primary navigation docks to the top of the page and there are visual cues as to what day the user is looking at.

Even though it makes sense to let a single "unit" of content scroll on a single long page, some sites still divide it across several pages. The reasons for this most often seem not to be about minimizing either vertical scrolling or page load size, but rather maximizing ad revenue from the multiple loads. If the content is finite, paginating it like this makes finding, saving, and using the content a more convoluted task even with print functions. Pagination makes sense only for very long lists of similar elements, such as search results or news articles.
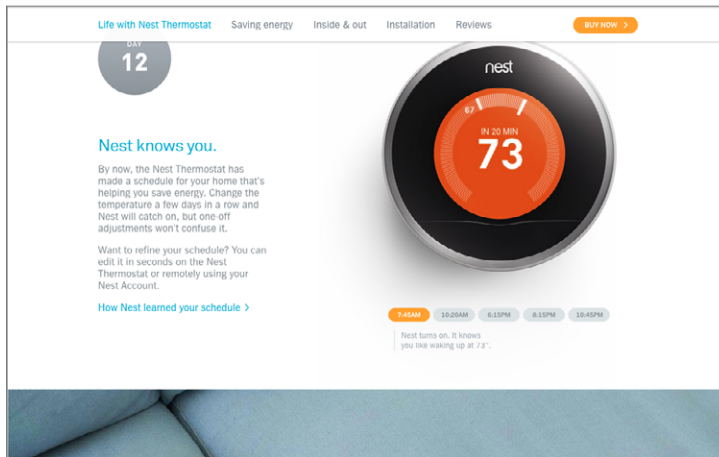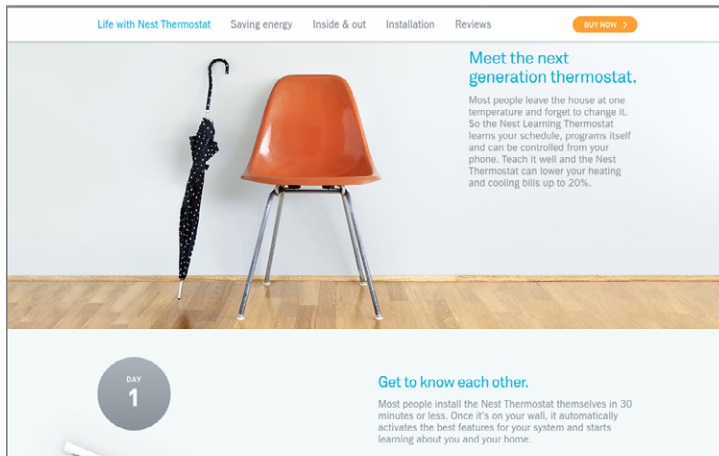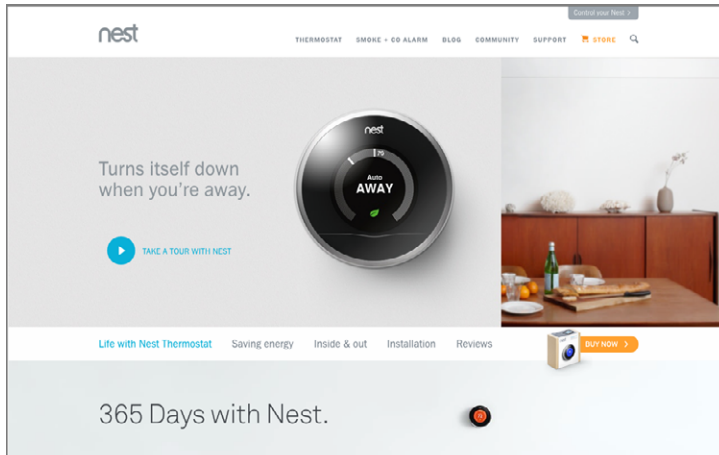
Figures 20-11: The Nest website consists of a number of long scrolling pages.

The header and footer

An obvious and hugely important characteristic of the scrolling page is that the top and bottom of the page are special places with unique opportunities to improve user flow. The top, often called the header, can be the first thing the user sees when he arrives on a page. But in many cases, we intentionally bring the most important content below the header to the forefront, allowing the header to recede into the background a bit. The header almost always includes a brand element like a logotype, and some persistent navigation items, including the primary navigation (discussed earlier). The header is also commonly the place where a website or application tells the user whether he or she is signed in. Finally, the search function often resides in the header.

The bottom of the page, or footer, is where, if you're lucky and wise, the user ends up, because he's viewed all the content that came before it on the page. This makes it a great place to suggest where the user should go next—often to related content. You can see this pattern used to good effect on many media websites. Another effective use of the footer is for persistent access to more rarely visited areas of your site or application, like legal notices, or for a complete fat navigation that includes all top-level and second-level pages (see Figure 20-12). These can certainly be effective approaches. But it's important to consider the circumstances under which your users might need access to these links and whether you imagine they'll be web-savvy enough to scroll to the bottom of the page to look for them.

Figures 20-12: The fat footer on Zappos.com contains a condensed sitemap as well as other social and promotional content and links.

Paging versus infinite scrolling

One important scrolling-related pattern in things like social media streams and search results is commonly called *infinite scrolling*. As the user scrolls down the page, the page populates more results into the bottom. Assuming you can keep latency down, this can be a useful and natural-feeling interaction.

This is in contrast to the *paging* of results, in which a predefined number of results are shown on a page, and navigational links are supplied so that the user can advance to the next or previous page, as well as (typically) the beginning or end of the results, or to an arbitrary page of results.

> **DESIGN PRINCIPLE**
>
> *Infinite scroll and site footers are mutually exclusive idioms.*

It's important to remember that if you implement infinite scrolling, your users will never get to the bottom of the page and therefore will never see the page footer. Infinite scrolling and page footers are mutually exclusive navigation idioms.

Furthermore, infinite scrolling can introduce other potential usability challenges, so it should be used juduciously:

- Keyboard and screen-reader navigation typically does not work well (if at all) with infinite scrolling, leading to accessibility issues.
- Unless carefully designed and implemented, infinite scrolling may not retain its place in the list after use of the browser back button (and subsequent use of the forward button to return). This can take users off guard, and can lead to a laborious and frustrating experience of re-scrolling to find a lost item.

The inability to page directly and predictably to items far down in the list makes infinite scrolling most appropriate for contexts such as news feeds, where information far down in the list quickly loses its relevancy, and browsing recent items is the primary activity.

Infinite scrolling should *never* be employed for interfaces in which users need to get to the end of the list quickly, or need to return to a particular list item after navigating elsewhere.

The Mobile Web

Since the web's early days, design has had to contend with users who have different-sized screens, using different browsers, on different operating systems. The huge rise of people interacting with websites and applications on tablets and phones has made it critical that designs render gracefully and properly on a wide variety of screen sizes.

The contemporary approach for handling these different screen sizes is commonly called *responsive design*. It's a deep topic, handled well by several books and articles that have more time to discuss it than we do here. We recommend Ethan Marcotte's *Responsive Web Design* (A Book Apart, 2011).

This method involves creating a modular layout grid in which content areas flexibly resize up to a point. At key screen widths, called *breakpoints*, this grid may make more substantial changes. For example, for screens greater than 1024 pixels wide, we might choose to show three data visualizations side by side on a page. But when the screen is less than 1024 pixels wide, the data visualizations stack on top of each other. The Boston Globe's website is a good example of a site that makes use of responsive techniques (see Figure 20-13).

DESIGN PRINCIPLE	*If you have only one version of your site, make it responsive.*

The basic idea with responsive design is not to have multiple versions of a website or application for different screen sizes, but a single version that dynamically adapts to the screen on which it's being viewed. This approach has both pros and cons. The advantage is that a team works within a single conceptual framework. The disadvantage is that this single UI can be complex for developers to build, and every breakpoint means another layout to design.

An alternative (and, sometimes, more effective) approach is to create a separate mobile version of the site or application. One of the biggest reasons for this is that screen size is only one consideration on the mobile web. It's also critical to think about how your designs accommodate touch interaction and other sensors, as well as how they perform in sunlight and other challenging lighting conditions. Because of these usage considerations, it is sometimes a better choice to create a separate version of your web application or site for mobile users.

Figure 20-13: The Boston Globe's website has several different breakpoint screen widths that change how the content is flowed and accessed on a desktop, tablet, or phone.

The Future

As web technologies become more robust, and browsers continue on their trajectory toward providing richer interaction patterns, the browser will continue to be one of the most important UI platforms. More-sophisticated visualizations and interactions will be possible in HTML5. Browsers are likely to improve their local data caches, breaking down one of the last remaining differences between locally installed native applications and applications that run in a browser window.

As the web and "traditional" media, like TV and print, continue to converge, we see many possibilities for new content models, new ways of telling stories, and new ways for people to interact with media. Looking at stellar examples like Medium, a website that allows collaborative content creation, and the beautiful "Snow Fall" multimedia journalistic piece in the *New York Times*, it's clear that the web is one of the most important places to watch for software and media to truly converge.

Notes

1. http://exisweb.net/menu-eats-hamburger

2. http://www.nngroup.com/articles/incompetent-search-skills/

DESIGN DETAILS: CONTROLS AND DIALOGS

Even though they may have somewhat different visual design trappings from platform to platform, controls and dialogs constitute a common language for users interacting with most of the digital products available today. These standard objects are available as part of most GUI development libraries, but this doesn't prevent their misuse in many software applications.

This chapter outlines the majority of common interactive GUI controls and also discusses their appropriate (and inappropriate) contexts of use.

Controls

Controls are self-contained screen objects through which people interact with digital products. Controls (also known as widgets, gadgets, and gizmos) are the primary building blocks for creating a typical graphical user interface.

Examined in light of users' goals, controls come in four basic types:

- Imperative controls, used to initiate a function
- Selection controls, used to select options or data

- Entry controls, used to enter data
- Display controls, used to manipulate the how and where the application displays itself and its data

Some controls span more than one of these categories.

Imperative controls

The interaction between humans and computers involves a language of nouns (sometimes called objects), verbs, adjectives, and adverbs. When we issue a command, we are specifying the verb—the statement's action. When we describe what or whom the action will affect, we are specifying the sentence's noun. Sometimes we choose a noun from an existing list, and sometimes we enter a new one. We can modify both the noun and the verb with adjectives and adverbs, respectively.

A control that corresponds to a verb is called an *imperative control* because it commands action, most often immediately. Menu bar items (discussed in Chapter 18) are also imperative idioms. In the world of controls, the quintessential imperative idiom is the button. Click the button, and the associated action—the verb—executes immediately.

Buttons

Buttons were once identified by their simulated-3D raised aspect. However, the trend of flattening affordances, started in the mobile world, removes the 3D cues and threatens to degrade these controls' learnability, as shown in Figure 21-1. Generally, if the control is rectangular (with rounded edges on some platforms), it has the visual affordance of an imperative. It executes as soon as the user either taps it or clicks and releases it using the mouse. In dialogs (discussed later in this chapter), a default button is often highlighted to indicate the most common action for the user to take.

Figure 21-1: Standard buttons from Microsoft Windows (top left) Apple OS X (top right), Android (bottom left), and iOS (bottom right). Although pushbuttons were once given a 3D raised affordance indicating pressability, a flatter look appears to be the growing trend.

The button is arguably the most easily discoverable idiom in the designer's toolkit. It isn't surprising that it has evolved with such diversity across the user interface. The manipulation affordances of faux three-dimensional buttons prompted their widespread use.

Part of a button's affordance is its visual pliancy, which indicates its pressability. When the user points to it and clicks the mouse button—or taps the button with a finger or stylus—the button control visually changes from raised to indented, indicating that it has been activated. This is an example of dynamic visual hinting, discussed in Chapter 13. Poorly designed applications and many websites contain buttons that don't animate when clicked or tapped. This is disconcerting for users, because it generates a mental question: "Did that actually do something?" Users expect to see the button change—the pliant response—and you must satisfy that expectation.

Though flat design does away with this pliancy, it can only afford to do so because of the decades of training and experience users have with the rounded-rectangle shape.

Icon buttons

The toolbar (discussed at length in Chapter 18) has grown into a de facto standard as familiar as the menu bar. To populate the toolbar, buttons were adapted from their traditional home on the dialog.

When buttons moved to the toolbar, they changed from rectangular to square, and their text labels were replaced with iconic ones. Thus was born the icon button: half button, half icon (see Figure 21-2).

Figure 21-2: Icon buttons from Microsoft Office. On the left are examples in Office for Windows, and on the right are the same examples in Office for OS X. Icon buttons aren't rendered with a button affordance until the mouse cursor passes over them.

In Windows 98, the *icon button* (aka the toolbar button) continued to evolve, losing its raised affordance except when used. This move reduced visual clutter in response to the overcrowding of toolbars. Unfortunately, this made it more difficult for newcomers to understand the idiom. Starting with Windows 2000, desktop icon buttons revealed their affordance only when pointed at by the mouse cursor.

Icon buttons are, in theory, easy to use: They are always visible and don't demand as much time or dexterity as a drop-down menu. Because they are constantly visible, they are easy to memorize, particularly in sovereign applications. The advantages of the icon button are hard to separate from the advantages of the toolbar; the two are inextricably linked.

The downside of the icon button derives not from its button part but from its icon part. Most users have no problem understanding the visual affordance. The problem is that the image on the face of the icon button is seldom that clear for a first-time user.

Most icons are difficult to decipher with certainty at first glance, but ToolTips can help with this. A good icon will be learned and remembered when users return to that function frequently. This is the type of behavior we typically see from intermediate and advanced users.

However, even the best icon designers will be hard pressed to devise an icon system that will be usable by novice users without resorting to text labels. ToolTips will help them, but it is awkward to move the mouse cursor and then wait for the ToolTip for every icon button in your interface. In these cases, menus with verbose textual labels are a better approach. Microsoft's ribbon control (also discussed in Chapter 18) takes a hybrid approach that combines text and icons, trading some screen real estate for greater clarity and ease for novice users or infrequently-used commands.

Hyperlinks

Hyperlinks, or links, are a web convention that has found their way into all sorts of different applications. Typically taking the form of blue underlined text (CSS styling can play havoc with this standard in all sorts of ways, such as changing their default and traversed colors, or providing a focus highlight color on mouseover), a link is an imperative control used for navigation. This direct and useful interaction idiom has grown beyond its simple beginnings in taking users to a web page that provides more details about a hyperlinked word or phrase—the original concept behind hypertext. Links now form the navigational infrastructure of complex transactional websites such as Amazon.com (see Figure 21-3), and, somewhat amazingly, they remain up to the task.

Figure 21-3: Complex transactional websites such as Amazon.com rely on the simple hyperlink for much of their navigational infrastructure, which for the most part works remarkably well.

Images can also be used as links. However, the lack of affordance, especially on mobile browsers where there is no possibility of highlight on mouseover to indicate pliancy, can be problematic.

Unfortunately, the idiom's success and utility have given many designers the wrong idea: They believe that replacing more common imperative controls such as buttons or icon buttons with hyperlinks will automatically result in a more usable and successful user interface. Usually this is not the case. Because most users have learned that links are a navigational idiom, they will be confused and disoriented if clicking a link results in the execution of an action (such as the launching of a dialog). In general, you should use links for navigation through content, and buttons or icon buttons for other actions and functions.

A common web tactic for dialog boxes is to present options using a combination of a button for the "default" choice and an adjacent hyperlink for the other option. This is effective because a button has a greater visual weight and real estate for easier selection. Unfortunately, it has been used too often to trick users into not noticing the link, and, thinking they only have one option, making an expensive choice. Because of this, the tactic can be seen as manipulative, potentially reducing the user's trust in the site and the brand.

> **DESIGN PRINCIPLE**   *Use links for navigation and buttons for action.*

Selection controls

Because imperative controls represent commands— verbs—they need objects—nouns— on which to operate. Selection and entry controls are the two controls used to define nouns (in addition to various custom direct manipulation idioms). A *selection control* allows the user to choose this noun from a group of valid choices. Selection controls are also used to configure actions. The nouns may be defined by direct-manipulation idioms, with selection controls then used to define an adjective or adverb modifying the object or action on it. Common examples of selection controls include check boxes, list boxes, and drop-down or pop-up lists.

Traditionally, selection controls did not directly result in actions—they required an imperative control to activate. This is no longer always the case. In some situations, such as the use of a drop-down or pop-up list as a navigation control on a web page, this can be disorienting to users. In other cases, such as using a drop-down list to adjust type size in a word processor, this can seem quite natural.

As with many things in interaction design, both approaches have advantages and disadvantages. In cases where it is desirable to allow the user to make a series of selections before committing to the action, there should be an explicit imperative control (a button). In cases where users would benefit from seeing the immediate impact of their actions, and those actions are easy to undo, it is completely reasonable for the selection control to double as an imperative control.

Check boxes

The *check box* was one of the earliest visual control idioms. It has been a favorite for presenting a single, binary choice or for selecting from among several choices in a short list (see Figure 21-4). The check box has a strong visual affordance for clicking; it appears as a pliant area because of a mouseover highlight or a 3D "recessed" visual treatment. (The flattening trend in mobile visual design threatens learnability here as well.) After the user selects it and sees the checkmark appear, he has learned all he needs to know to make it work at will: Click to check; click again to uncheck. The check box is simple, visual, and elegant.
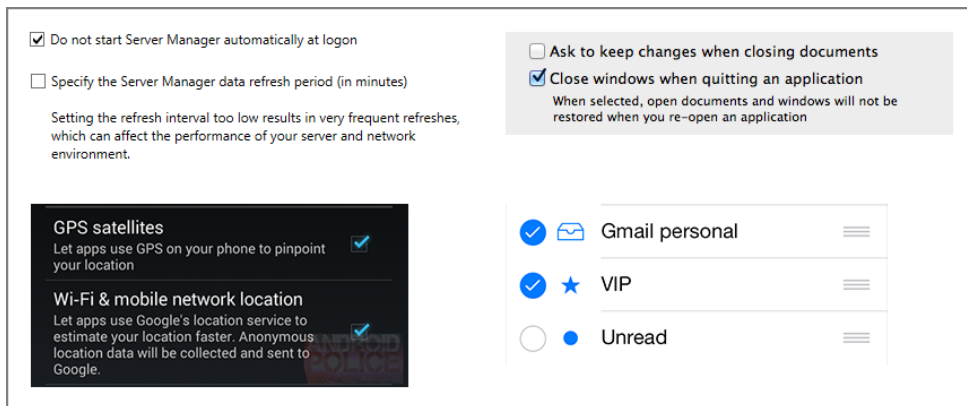
Figure 21-4: Standard check boxes from Microsoft Windows (top left), Apple OS X (top right), Android (bottom left), and iOS (bottom right). Again, the trend towards flatness decreases learnability. iOS also breaks the standard check box idiom by using a circular control rather than a square.

The check box is, however, primarily a text-based control. The check box is a familiar, effective idiom, but it has the same strengths and weaknesses as menus. Well-written text can make check boxes unambiguous. However, this exacting text forces users to slow down to read it, and it takes up a considerable amount of real estate.

Traditionally, check boxes are square. Users recognize visual objects by their shape, and the square check box is an important standard. Nothing is inherently good or bad about squareness; it just happens to have been the shape originally chosen, and many users

have learned to recognize this shape. There is no good reason to deviate from this pattern. Don't make check boxes diamond-shaped, and especially not round (which confuses them with the visual idiom employed by radio buttons), regardless of what your marketing department or visual designers say.

Toggle buttons

Check boxes work well to allow binary state changes, but this idiom isn't well-suited for toolbars. It is, however, possible to implement a more graphical approach to the unitary check box by modifying the icon button idiom. By allowing an icon button to stay in the pushed-in state when clicked, and then returning to the nondepressed state when it is clicked again, you create a *toggle button* (see Figure 21-5). The pushed-in state is no longer momentary, but rather locks in place until clicked again. The button's toggle behavior has changed its character sufficiently to move it into an entirely different category of control: from imperative to selection.

Figure 21-5: These images depict toggle buttons in their flat, mouseover, clicked, and selected states.

The toggle button is superseding the check box as a single-selection idiom. It is especially appropriate in modeless interactions that do not require interrupting the user's flow to make a decision. Toggle buttons are more space-efficient than check boxes. They are smaller because they can rely on visual recognition instead of text labels to indicate their purpose. Of course, this means that they exhibit the same problem as imperative icon buttons: the icon's inscrutability. We are saved once again by ToolTips. Those tiny pop-up windows give us just enough text to disambiguate the icon button without permanently consuming too many pixels.

State-switching buttons: an idiom to avoid

State-switching buttons are an all-too-common control variant used to save interface real estate. Unfortunately, this savings comes at the cost of considerable user confusion. A classic example is collapsing play and pause onto the same button on an audio player. In the paused state, the button contains the universal play triangle icon. Then, when you click it, it switches to the play state and displays the universal pause icon—two vertical bars—without indenting the button as a regular toggle button would.

The control suggests that you can click it, so when it displays the play icon, it intends to mean that when you click it, music will start. The button then changes to display the pause icon to indicate that clicking it again will pause playback. The problem with this

approach is that the control could be interpreted as indicating the player's state—paused or playing. This means that there are two very reasonable and contradictory interpretations of the icons on the button. The control can serve as either a state indicator or a state-switching selection control, but not both (see Figure 21-6). Of course there's music playing to confirm your selection in a music player, but there are plenty of interfaces where such explicit confirmation is not available.
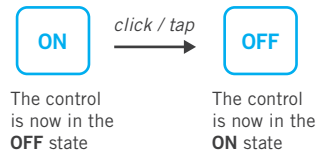
Figure 21-6: State-switching button controls are very efficient. They save space by controlling two mutually exclusive options with a single control. The problem with these controls is that they fail to fulfill the second duty of every control—to inform users of their current state. If the button says ON when the state is off, it is unclear what the setting is. If it says OFF when the state is off, however, where is the ON button? Using switch controls makes much more sense in this instance.

The solution is to either spell out the verb—Play or Pause—in text on the button or, better yet, to use some other technique altogether. For example, you could replace the button with two buttons or, as some audio players do, with icons for both states. Doing so would make the toggle's state-switching nature more explicit. If this last approach also included highlighting the icon representing the active state, it would work almost perfectly. Unfortunately, however, almost every audio player app now uses the broken state-switching idiom, toggling between the play and pause icon, and most users have now become accustomed to it, for better or worse.

Radio buttons

Similar in appearance to the check box is the *radio button*, shown in Figure 21-7. When radios were first put in automobiles, people discovered that tuning the radio by rotating a knob while driving was unsafe. So, automotive radios were offered with a panel consisting of a half-dozen chrome-plated buttons. When pressed, each one would mechanically dial the tuner to a preset station. Now you could tune to your favorite station—without taking your eyes off the road—just by pushing a button.
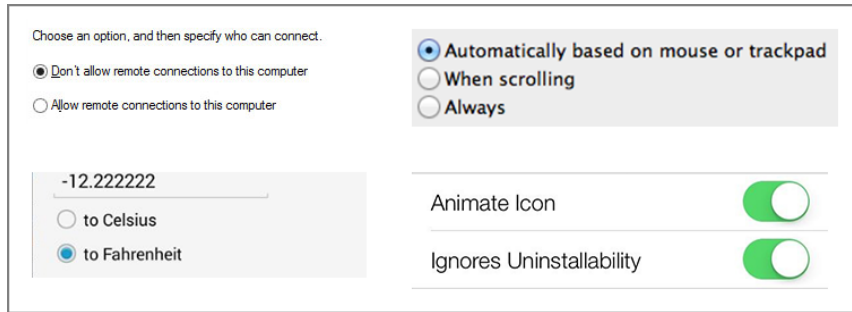
Figure 21-7: Standard radio buttons from Microsoft Windows (top left), Apple OS X (top right), Android (bottom left). iOS (bottom right) doesn't have a radio button idiom, but makes use of the switch control for some instances where a radio button might be used on other platforms.

The behavior of GUI radio buttons, like their mechanical forebears, is mutually exclusive: When one option is selected, the previously selected option automatically deselects. Only one button can be selected at a time. Consequently, radio buttons always come in groups of two or more, and one radio button in each group must always be selected. Presenting a single radio button doesn't make sense for the user—you should use a check box or similar selection control in that instance.

Radio buttons use the same amount of space on-screen as check boxes—more, even, since radio buttons are meaningful only in groups. Radio buttons are well suited to a pedagogical role, which means that they can be justified in infrequently used dialogs. But drop-down lists are often a better choice on the surface of a sovereign application that must cater to daily users.

For the same reason that check boxes are traditionally square, i.e., that's how we've always done it, radio buttons are almost always round.

Icon buttons have reimagined the radio button in the same way they have the check box: If two or more latching (toggle) icon buttons are grouped and are programmed so that only one of them at a time can be activated, they form a bank of radio icon buttons. This more modern construct ends up looking and behaving more like its mechanical ancestors than the traditional circular radio buttons.

The alignment controls on Word's toolbar are an excellent example of radio icon buttons, as shown in Figure 21-8.

Just as with all icon button idioms, these are efficient consumers of space, letting experienced users rely on spatial memory and pattern recognition to identify them and letting infrequent users rely on ToolTips to remind users of their purpose. First-time users

either will be clever enough to learn from the ToolTips or will learn more slowly, but just as reliably, from other, parallel, pedagogic command vectors.

Figure 21-8: Word's alignment controls are a bank of radio icon buttons, which act like traditional radio buttons. One is always selected, and when another is clicked, the first one returns to its unselected state. This variant is a space-conservative idiom that is well suited for frequently used options.

Switches

The *switch control* is a more compact version of two radio buttons used together. (It is also a more understandable version of a single check box, since both states are labeled explicitly.) It has two states, typically on and off, which are labeled on either side of the switch, as shown in Figure 21-9. Clicking either side of the switch or, on mobile, swiping in the appropriate direction slides the switch's 3D affordance to the on or off position. These are handy in Settings screens on mobile apps, where many app functions often can be selectively turned on and off. They are less common, and somewhat more awkward to use, in desktop and web apps.
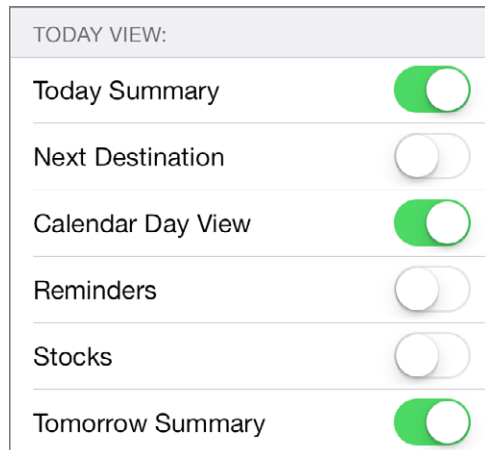
Figure 21-9: Switch controls are prevalent in mobile apps (such as in iOS, shown here), especially on Settings screens, which contain product features that can be turned on and off.

Combo icon buttons

A variant of the radio icon button replaces the bank of icon buttons with a drop-down list of icons. Because of its similarity to the combo box control (see the later section "Combo Boxes"), we call this a *combo icon button* (see Figure 21-10). Normally in Windows it looks like a single icon button with a small down arrow to its right. If you click the arrow, it drops down a menu of several icon buttons from which users may choose. The selected icon button then appears on the toolbar next to the arrow. Clicking the icon button itself actuates the imperative indicated by the selected state. Like menus, icon buttons also should activate if the user clicks and holds the arrow, drags, and then releases over the desired selection.
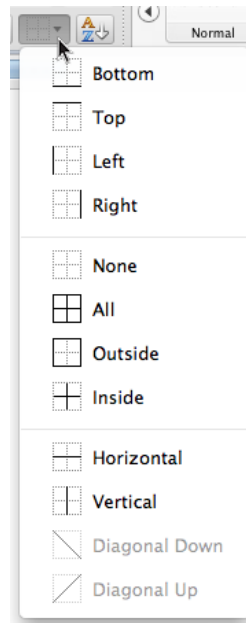
Figure 21-10: This combo icon button from Microsoft Office is a group of icon buttons that behave like a combo box.

Variations on combo icon buttons include drawing a small downward- or right-pointing triangle in the lower-right corner of the combo icon button icon in place of the separate down arrow that is seen in Microsoft toolbars. Adobe products use this variant in their palette controls. The user must click and hold the icon button itself to bring up the menu (which, in Adobe palette controls, unfolds to the right rather than down, as shown in Figure 21-11). You can vary this idiom quite a bit. Creative software designers are doing just that in the never-ending bid to cram more functions onto screens that are always too small.
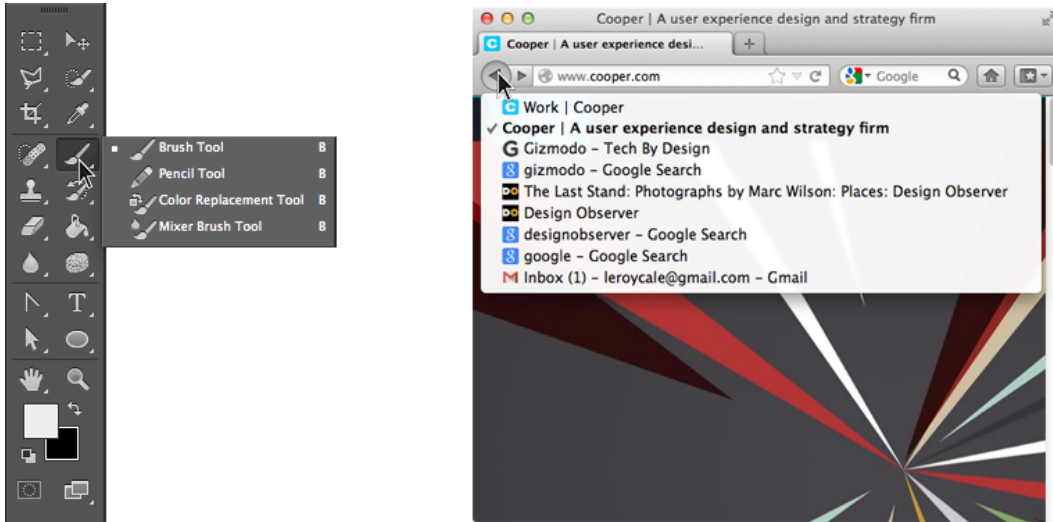
Figure 21-11: These combo icon buttons taken from Adobe Photoshop (left) and Mozilla Firefox (right) show the diversity of idiom applications. In Photoshop, the combo icon button is used to switch between various modal cursor tools, whereas in Firefox it is used to select a previously visited web page to return to. In the first example, it is used to configure the user interface, whereas in the second it is used to perform an action.

You can see a Microsoft variant in Word, where the icon button for specifying the colors of highlights and text show combo icon button menus that look more like little palettes than stacks of icon buttons. As you can see from Figure 21-11, these menus can pack a lot of power and information into a compact control. This facility is definitely for frequent users, particularly mouse-heavy users, and less for first-timers. However, for users who have at least basic familiarity with the available tools, the idiom is instantly clear after it is discovered or demonstrated. This is an excellent control idiom for sovereign-posture applications with which users interact for long hours. Working a menu with relatively small targets demands sufficient manual dexterity. But this is much faster than going to the menu bar, pulling down a menu, selecting an item, watching the dialog deploy, selecting a color on the dialog, and then clicking the OK button.

With the introduction of the ribbon control idiom, Microsoft moved away from traditional combo icon buttons and instead offered yet another variant of this idea: an icon button attached to a more standard menu. Clicking the icon button fires off an imperative command. Clicking the arrow to the right of or below the button launches a menu with related but less frequently accessed functions (see Figure 21-12).

Figure 21-12: Microsoft Office's ribbon control contains a new variant of combo button icons. Clicking the button launches an imperative, and clicking the arrow opens a more traditional menu of related functions.

List controls

List controls allow users to select from a finite set of text strings, each representing a command, object, or attribute. These controls are also known as list boxes or listviews, depending on which platform and control variant you are talking about. Like radio buttons, list controls are powerful tools for simplifying interaction, because they eliminate the possibility of making an invalid selection.

List controls are small text areas with a vertical scrollbar on the right edge, as shown in Figure 21-13. The application displays objects as discrete lines of text in the box, and the scrollbar moves them up or down. The user can select a single line of text at a time by clicking it. A list control variant allows multiple selection, in which the user can select more than one item at a time, usually by Shift-clicking for continuous or Ctrl-clicking for discontinuous items.
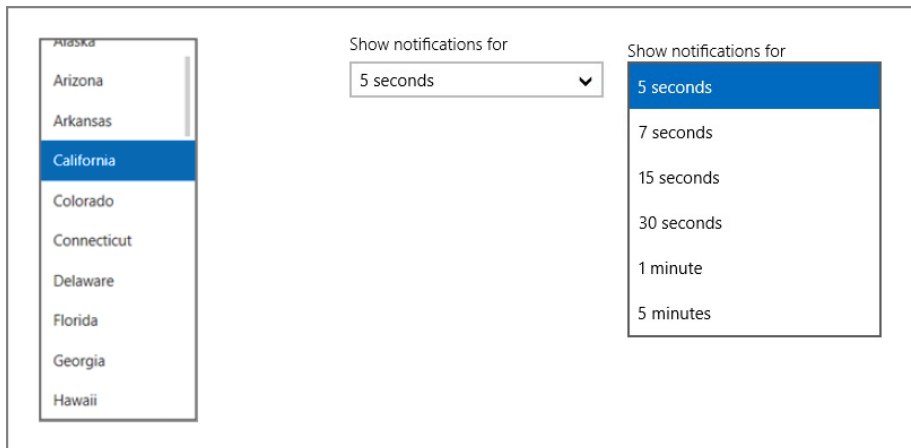
Figure 21-13: On the left is a list control from Windows. The images on the right show a Windows drop-down menu in its closed and open states.

The drop-down menu, discussed previously, can also be considered a variant of the list control. These ubiquitous controls show only the selected item in a single row, until the control is clicked or tapped. Doing so reveals other available choices (also shown in Figure 21-13).

Apple's iOS operating system has introduced a gestured-based variant of the list control sometimes called the "barrel control." In it, the list of text items are rendered to appear as though they are wrapped around a cylinder that you rotate by swiping until the desired item is centered in the control. The interesting twist on this control, besides its barrel-shaped rendering, is that it can contain independently scrolling columns. This makes it ideal for purposes such as selecting dates and times. This clever approach merges several related list controls into a single widget, as shown in Figure 21-14.
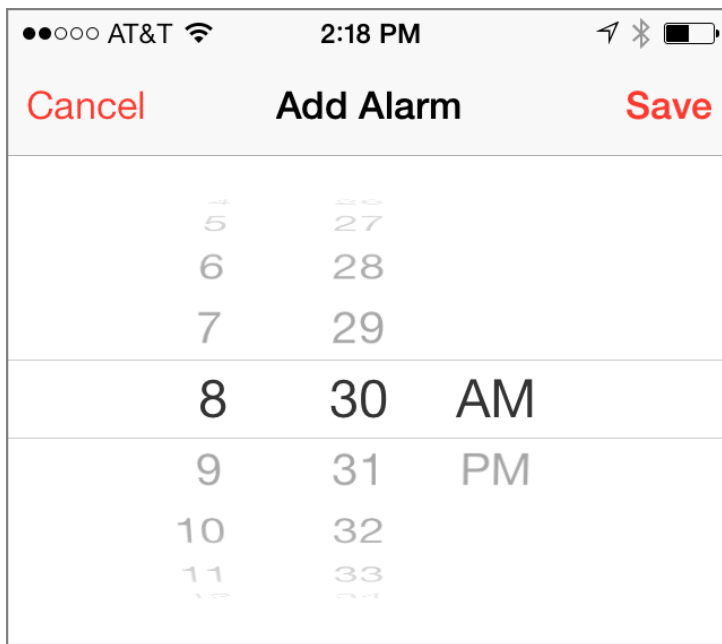
Figure 21-14: iOS supports a gesture-based "barrel control" listview variant that also supports multiple independently swipable scrolling columns. The barrel control thus can merge several related list controls into a single widget.

Early list controls handled only text. Unfortunately, that decision often affects their behavior to this day. A list control filled with line after line of text unrelieved by visual symbols is a dry desert indeed. However, starting with Windows 95, Microsoft has allowed each line of text in a listview control to be preceded with an icon (without the need for custom coding). This can be useful. In many situations, users benefit from seeing a graphical identifier next to important text entries (see Figure 21-15). A newer convention is to use the list items in a drop-down or other listview control as a preview

facility. This is commonly used in cases where the control functions as both a selection control and an imperative control, such as when you select a style in Microsoft Word.
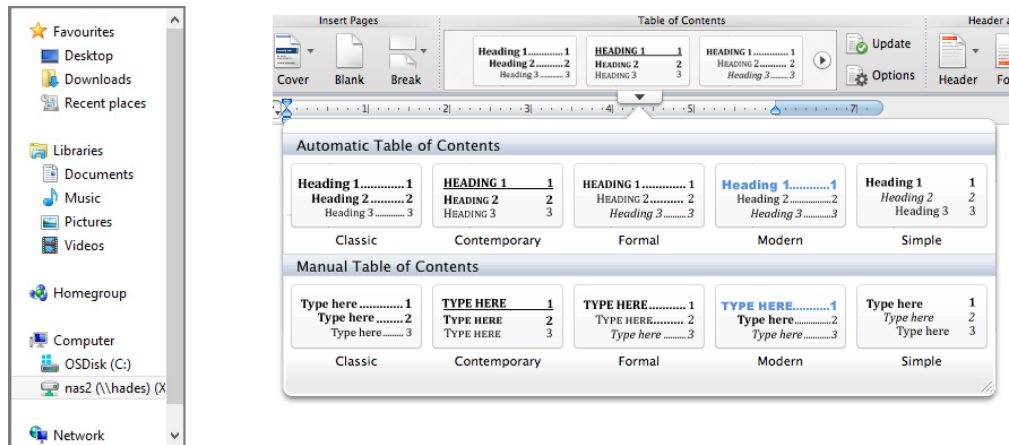
Figure 21-15: On the left is a list control with icons from Windows that allows users to visually identify the application they are looking for. On the right is the style drop-down list from Office 2010. Here, the items in the list provide a preview of the effects of their selection.

> **DESIGN PRINCIPLE**
>
> *Distinguish important text items in lists with graphic icons.*

Listviews are, true to their name, good for displaying lists of items and allowing users to select one or more of them. They are also a good idiom for providing a source of draggable items (though not with the drop-down variant). If the items are draggable within the listview itself, this makes a fine tool for enabling the user to put items in a specific order. (See the "Ordering Lists" section later in this chapter.)

Earmarking

Generally speaking, users select items in a list control as input to some function, such as selecting the name of a desired font from a list of several available fonts. Selection in a list control is conventional, with a selected item shown using the highlight color.

Occasionally, however, list controls are used to select multiple items, and this can introduce complications. The selection idiom in list controls is well suited for single selection but is much weaker for multiple selection. In general, the selection of multiple discrete objects works adequately if the entire playing field is visible at once, like the icons on a desktop. If two or more icons are selected at the same time, you can clearly see this, because all the icons are visible at the same time.

But if the pool of available discrete items is too large to fit in a single view, and some of it must be scrolled offscreen, the selection idiom immediately becomes unwieldy. This is the normal state of affairs for list controls. Their standard mode of selection is mutual exclusion, so when you select one thing, the previous selected thing is deselected. Thus, it is far too easy, in the case of multiple selection, for users to select an item, scroll it into invisibility, and then select a second item, forgetting that they have now deselected the first item because they can no longer see it.

The alternative is equally unpalatable: The list control is programmed to disable the mutual-exclusion behavior of a standard list control in its selection algorithm. This allows users to click as many items as they like, with all remaining selected. Things now work perfectly (sort of): The user selects one item after another, and each one stays selected. The fly in the ointment is that there is no visual indication that selection is behaving differently from the norm. It is just as likely that the user will select an item, scroll it into invisibility, and then spot a more desirable second item and select it, expecting the first—unseen—item to automatically be deselected because the control is mutually exclusive. You get to choose between offending the first half of your users or the second half.

When objects can scroll off the screen, multiple selection requires a better, more distinct idiom. A possible approach is to use an idiom different from simple selection—one that is visually distinct. But what is it?

It just so happens that we already have another well-established idiom to indicate that something is selected—the check box. Check boxes communicate their purposes and settings quite clearly, and, like all good idioms, they are easy to learn. Check boxes are also clearly disassociated from any hint of mutual exclusion. Suppose we added a check box to every item in our problematic list control. Not only would the user clearly see which items were selected and which were not, but he also would clearly see that the items were not mutually exclusive. This would solve both of our problems in one stroke. This check box alternative to multiple selection is called *earmarking*, an example of which is shown in Figure 21-16.

Dragging and dropping from lists

List controls can be treated as palettes of goodies to use in a direct-manipulation idiom. If the list were part of a report-writing application, for example, you could click an entry and drag it to the surface of the report to add a column representing that field. It's not selection in the usual sense, because it is a completely captive operation. Without a doubt, many applications would benefit if they used list controls that supported dragging and dropping.
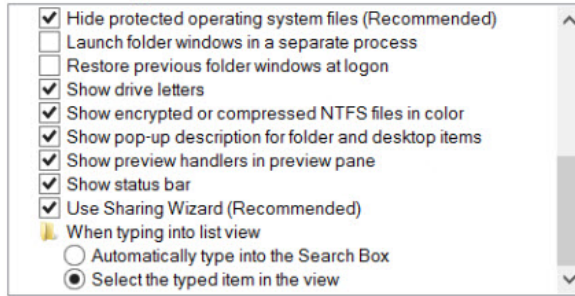
Figure 21-16: Selection normally is a mutually exclusive operation. When the need arises to discard mutual exclusivity to provide multiple selection, things can become confusing if some of the items can be scrolled out of sight. Earmarking is a solution. Put check boxes next to each text item, and use them instead of selection to indicate the user's choices. Check boxes are a clearly non-mutually exclusive idiom and a familiar GUI idiom. Users grasp the workings of this idiom right away.

Such draggable items can help users gather items into a set. Providing two adjacent list controls, one showing available items and the other showing chosen items, is a common GUI idiom. One or sometimes a bidirectional pair of arrow buttons placed between them allows items to be selected and transferred from one box to the other (see Figure 21-17). It is so much more pleasant when the idiom is buttressed with the ability to just click and drag the desired item from one box to another, without having to go through the intermediate steps of selection and function invocation.

Ordering lists

Sometimes the need arises to drag an item from a list control to another position in the same list control. This need arises more often than most interaction designers seem to think.

Many applications offer automatic sort facilities for important lists. Windows Explorer, for example, lets you sort files by name, modification date, type, and size. That's nice, but wouldn't it be even better if users could order them by importance? Algorithmically, the application could order them by frequency of user access, but that wouldn't always get the right results. Also adding a factor of how recently a file was accessed would get closer but still wouldn't be exactly right.

Why not let users move what's important to them to a region at the top, and sort those things separately (in alphabetical or whatever order), in addition to sorting the full directory below? For example, you might want to rearrange a list of the people in your department in descending order by where they sit. No automatic function will do this; you just have to drag them until it's right. This is the kind of customizing that an experienced

user wants to do after long hours of familiarization with an application. It takes a lot of effort to fine-tune a directory like this, and the application must remember the exact settings from session to session. Otherwise, the ability to reorder things is worthless.
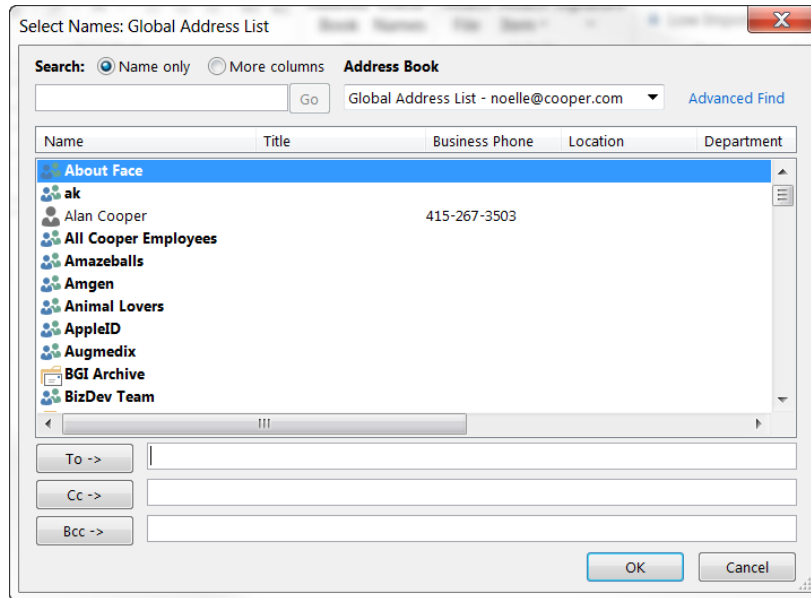
Figure 21-17: This dialog from Microsoft Outlook Express would benefit from the ability to drag a contact from the list on the left into the To, Cc, and Bcc lists. The arrow button functionality is a bit less clear since the lists that the contacts are copied to are below the list being copied from, rather than adjacent to it. Note also the unfortunate use of a horizontal scrollbar—but luckily, the dialog can be expanded by dragging the lower left corner.

Being able to drag items from one place to another in a list control is powerful, but it demands that autoscrolling be implemented (see Chapter 18). If you pick up an item in the list but the place you need to drop it is currently scrolled out of view, you must be able to scroll the listview without putting down the dragged object.

Horizontal scrolling in lists

List controls normally have a vertical scrollbar for moving up and down through the list. List controls can also be made to scroll horizontally. This feature allows the developer to put extra-long text into the list controls with a minimum of effort. However, it is a huge pain for users.

Text should only be scrolled horizontally in large tables such as spreadsheets, where locked row and column headers can provide context for each column. When a text list is scrolled horizontally, it hides from view one or more of the first letters of every single

line of text showing. This makes none of the lines readable, and the text's continuity is destroyed.

If you find a situation that seems to call for horizontal scrolling of text, search for alternative solutions. Begin by asking yourself why the text in your list is so long. Can you shorten the entries? Can you wrap the text to the next line to avoid that horizontal length? Can you allow the user to enter aliases for the longer entries? Can you use graphical entries instead? Can you use ToolTips? Ideally, you also should consider whether you can widen the control. Can you rearrange things on the window or dialog to expand horizontally?

Absent the ability to widen the control, there are two possible approaches: Wrap the text to the next line, indenting it so that it is visually different from other entries, or truncate with an ellipsis, and provide the full text with a ToolTip. The former means that you now have a list control with items of variable height. The latter might be problematic if the list entries start with similar text. But either is still better than horizontal scrolling.

Remember, we're just talking about text. For graphics or large tables, there is nothing wrong with horizontal scrollbars or horizontally scrollable windows in general. But providing a text-based list with a required horizontal scrollbar is like providing a computer with a required hand-crank electrical generator—bad news.

Entering data into a list

Modern list and tree controls (discussed later in this chapter) offer an edit-in-place facility. Windows Explorer uses both of these controls; you can see how they work by renaming a file or directory. To rename a file in OS X or Windows, you click the desired name twice—but not too quickly, lest this action be interpreted as a double click and open the object in question. You then enter whatever changes you want. Items that can be edited in other circumstances should, when displayed in list controls, be editable there as well.

The edge case that makes edit-in-place a real problem is adding a new entry to the list. Most designers use other idioms to add list items. Click a button or select a menu item, and a new, blank entry is added to the list. The user can then edit its name in place. It would be more sensible if you could, say, double-click in the space between existing entries to create a new, blank entry right there, or at least have a perpetual open space at the beginning or end of the list with a Click to Add Entry label on it to make it

discoverable. Another solution to this problem is the combo box, which we'll talk about in a moment.

Drop-down and pop-up lists

Drop-down lists (also called pop-up lists) take the place of a stack of radio buttons. This gives users a compact way to make a single selection from a list (see Figure 21-13). The current selection is shown when the pop-up list closes. Generally speaking, pop-up lists, unlike their command-oriented cousins in the menu bars of desktop applications, focus on selecting objects rather than executing commands. However, they can sometimes immediately affect the display of information if, for example, they are used as part of a live search filter, or as the mechanism for navigating to a page on a website.

Combo boxes

The *combo box* is—as its name suggests—a combination of a list box and an edit field (see Figure 21-18). It provides an unambiguous method of data entry into a list control. As with normal list boxes, a drop-down variant has a reduced impact on screen real estate.
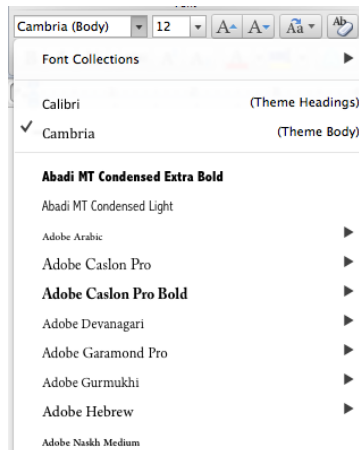
Figure 21-18: The Word font selection drop-down combo box allows users to select a font from the drop-down list or simply type the name of the desired font into the text field.

Combo boxes clearly differentiate between the text-entry part and the list-selection part, minimizing user confusion. For single selection, the combo box is a superb control. You can use the edit field to enter new items, and it also shows the current selection in the list. When the current selection is showing in the edit field, the user can edit it there—sort of a poor man's edit-in-place.

Because the edit field of the combo box shows the current selection, the combo box is by nature a single-selection control. There is no such thing as a multiple-selection combo box. Single-selection implies mutual exclusion, which is one of the reasons why the combo box has replaced groups of radio buttons for selection from among mutually exclusive options. Other reasons include space efficiency and the ability to add items dynamically, something that radio buttons cannot do.

When drop-down variants of the combo box are used, the control shows the current selection without consuming space to show the list of choices. Essentially, it becomes a list-on-demand, much like a menu provides a list of immediate commands on demand. A combo box is a pop-up list control.

The screen efficiency of the drop-down combo box allows it to do something remarkable for a control of such complexity: It can reasonably reside permanently on an application's main screen. It can even fit comfortably on a toolbar. It is an effective control for deployment on a sovereign-posture application. Using combo boxes on the toolbar is more effective than putting the equivalent functions on menus. Combo boxes display their current selection without requiring any action on the user's part, such as pulling down a menu to see the current status.

If drag-and-drop is implemented in list controls, it should also be implemented in combo boxes. For example, being able to open a combo box, scroll to a choice, and then drag the choice onto a document under construction is a powerful idiom (the appearance of a drag handle on mouseover could provide pliancy). Drag-and-drop functionality should be a standard part of combo boxes.

Tree controls

Tree controls (sometimes called "treeviews") are listviews that can present hierarchical data. They display a sideways "tree" of visually-connected branches of items, often with icons for each entry. The entries can be expanded or collapsed the way that many outlining applications work. Developers tend to like this control, because it often matches the way they think about complex functions and data, and because it is easy to build. It is often used as a file system navigator and is a highly effective way to present hierarchical information.

Unfortunately, hierarchical trees are one of the most inappropriately used controls in the toolbox. They can be problematic for users because many people have difficulty thinking in terms of hierarchical data structures. We have seen countless interfaces where developers have forced nonhierarchical data into a tree control with the rationale that trees are "intuitive." While they may be intuitive to developers, they neither allow users to capitalize on other, more interesting relationships between objects, nor respect the often messy real-world relationships between things.

It only makes sense to use a tree control (no matter how tempting it may be) in the case where what is being represented is "naturally" thought of as a hierarchy (such as a family tree). Using a tree control to represent objects that are arbitrarily related is asking for big trouble when it comes to usability.

Entry controls

Entry controls enable users to supply information to or set a value in an application.

The most basic entry control is a text edit field. Like selection controls, entry controls represent nouns to the application. Because a combo box contains an edit field, some combo box variants qualify as entry controls too. Also, any control that lets users enter a numeric value is an entry control. Because they allow users to set numeric values, controls such as spinners, gauges, sliders, and knobs fit in this category.

Bounded and unbounded entry controls

Any control that restricts the available set of values that the user can enter is a *bounded entry control*. A slider that moves from 1 to 100, for example, is bounded. Regardless of the user's actions, no number outside those specified by the application can be entered with a bounded control. This prevents users from entering an invalid value.

Conversely, a simple text field can accept any keyboard character the user types into it. This open-ended entry idiom is an example of an *unbounded entry control*. With an unbounded entry control, it can be easy for users to enter values that are invalid for the application. The application may subsequently reject the value, of course, but users can still enter it.

Simply put, bounded controls should be used wherever bounded values are needed. If the application needs a number between 7 and 35, presenting users with a control that accepts any numeric value from –1,000,000 to +1,000,000 doesn't do anyone any favors. People would much rather be presented with a control that embodies 7 as its bottom limit and 35 as its upper limit. (Clearly indicating these limits is also useful.) Users are smart, and they will immediately comprehend and work within the limits of their sandbox.

It is important to understand that we are talking about the quality of the entry control, not the quality of the data. To be a bounded control, it needs to clearly communicate, preferably visually, the acceptable data boundaries to the user. A text field that rejects the user's input after he enters it is not a bounded control. It is a rude control.

> **DESIGN PRINCIPLE**
>
> *Use bounded controls for bounded input.*

Most quantitative values needed by software are bounded, yet many applications still permit unbounded entry within numeric fields. The poor user types 17 into a field, and this innocent entry is rewarded with an error dialog saying, "You can enter only values between 4 and 8." This is poor user-interface design. A much better scheme is to use a bounded control that automatically limits the input to 4, 5, 6, 7, or 8. If the bounded set of choices is composed of text rather than numbers, you can still use a bounded slider (sometimes called a *trackbar*), combo box, or list box.

Figure 21-19 shows a vertical trackbar used by Microsoft in the Windows Display Settings dialog. It works like a slider or scrollbar, but has several discrete positions that represent distinct resolution settings. Microsoft could easily have used a simple drop-down list in its place. In many cases, a slider is a nice choice because it displays the range of valid entries. A drop-down menu isn't much smaller, but it keeps its options hidden until clicked—a less friendly stance. It's unclear why Microsoft chose to put a slider *inside* a drop-down menu.
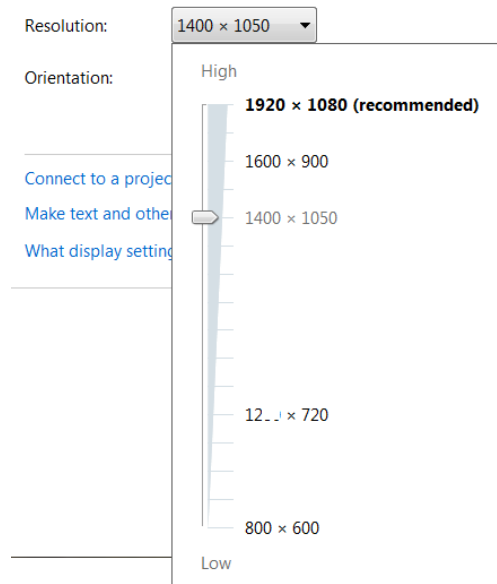
Figure 21-19: A bounded control lets users enter only valid values. It does not let them enter invalid values, only to reject them when they try to move on. This figure shows a bounded slider control from the Display Settings dialog in Windows. The slider (which, oddly, is deployed inside a drop-down menu) has several discrete positions. As you drag the slider, the legend beside it reflects different allowable screen resolutions, with recommended resolutions shown even when the trackbar thumb is not on the detent.

Spinners

Spinner controls are a common form of numeric entry control that permit data entry using the mouse, keyboard, or finger. Spinners on the desktop contain a small edit field with two half-height buttons attached, as shown in Figure 21-20. On iOS they're called *steppers* and have plus or minus buttons side-by-side, making them much easier to actuate with fingers.
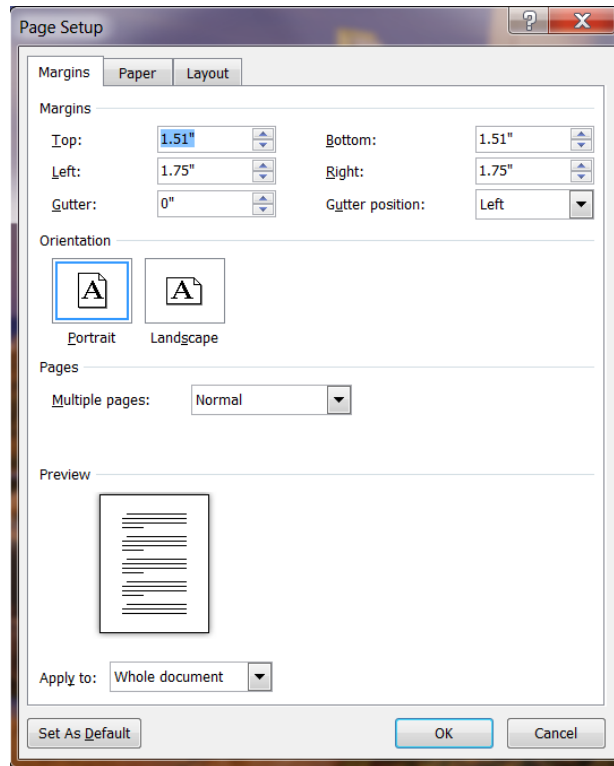
Figure 21-20: The Page Setup dialog from Microsoft Word makes heavy use of the spinner control. By clicking either of the small, arrowed buttons, the user may increase or decrease the specific numeric value in small, discrete steps. If the user wants to make a large change in one action or enter a precise setting, he can use the edit field portion for direct text entry. The arrow button portion of the control embodies bounding, whereas the edit field portion does not.

Spinners blur the difference between bounded and unbounded controls. Using either of the two small arrow buttons enables the user to change the value in the edit field in small, discrete steps. These steps are bounded, meaning that the value doesn't go above the upper limit set by the application or below the lower limit. If the user wants to make a large change in one action or enter a specific number, he can do so by clicking in the edit field portion and typing in it, just like entering text into any other edit field. Unfortunately, the edit field portion of this control is unbounded, leaving users free to enter

values that are out of bounds or even unintelligible. In the Page Setup dialog shown in Figure 21-20, if the user enters an invalid value, the application behaves like most other rude applications: It issues an error dialog explaining the upper and lower boundaries (sometimes) and requiring the user to click the OK button to continue.

Overall, the spinner is an excellent idiom and can be used in place of plain edit fields for most bounded numeric entries.

Dials and sliders

Dials and sliders are idioms borrowed directly from Mechanical-Age metaphors of rotating knobs and sliding levers. Dials are very space-efficient. Both can do a nice job of providing visual feedback about settings, as shown in Figure 21-21.

Figure 21-21: Korg's iPolysix app, a software synthesizer, makes heavy use of dials and sliders. These are effective interface elements because musicians and producers are familiar with them from hardware. More importantly, they provide users with more visual and easy-to-comprehend feedback about parameter settings than a long list of numbers, which aren't that exciting to look at while making music. iPolysix dials make users move their finger in an arc, rather than up-down or left-right swipes, which would be easier to control.

Improperly implemented, dials can be extremely difficult to manipulate. Sliders are often a better option where space isn't at a premium, because they visually suggest the fact that movement is along just one axis.

Sometimes developers force users to trace an arc with their mouse or finger, with mouse or finger distance from the control therefore controlling the granularity of rotation. Proper implementation of a dial should allow linear input in two dimensions: Clicking (or tapping) the dial and moving up or right should increase the dial's value, and moving down or left should decrease the value. Velocity can control granularity of adjustment. Of course, users must learn this idiom, or they try to move in an arc anyway.

Dials are best suited for specialized, sovereign applications where users become accustomed to the idiom. Because of their compact size and visual qualities (not to mention heritage), they are popular in audio software.

Although sliders and dials are both used primarily as bounded entry controls, they are sometimes used (and misused) as controls for changing the display of data. For most purposes, scrollbars do a better job of moving data in a display, because they can easily indicate the magnitude of the scrolling data, which sliders can't do as well. However, sliders are an excellent choice for zooming interactions on the desktop, such as adjusting the scale of a map or the size of photo thumbnails. Direct manipulation interfaces are better off sticking to the pinch in/pinch out conventions of touch technology.

Thumbwheels

The thumbwheel is a variant of the dial, but it is much easier to use. Onscreen thumbwheels look rather like the scroll wheel on a mouse, and they behave in much the same way. They are popular with some 3D applications because they are a compact unbounded control, which is perfect for certain kinds of panning and zooming. Unlike a scrollbar, they need not provide any proportional feedback, because the control's range is infinite. It makes sense to map a control like this to unbounded movement in some direction (like zoom) or movement within data that loops back on itself.

Other bounded entry controls

Breaking free from the heritage of traditional GUI controls and the baggage of mechanical analogs, a new generation of more experimental user interfaces is establishing new visual and gestural idioms. These range from a simple two-dimensional box where a click at any point defines the values for two input mechanisms (the vertical and horizontal coordinates each drive a parameter's value) to more complex direct manipulation interfaces (see Figure 21-22). These controls typically are bounded, because their implementation requires careful thought about the relationship between gesture and

function. Such control surfaces often provide a mechanism for visual feedback. These controls are also most appropriate for situations where users attempt to express themselves in regards to a number of variables and are willing to spend some effort developing proficiency with a challenging idiom.

Figure 21-22: Camel Audio's Alchemy Pro app employs a variety of two-dimensional bounded input controls. These provide good visual feedback, allow users to adjust multiple parameters from a single control, and support more expressive gestural user interactions. Their bounded nature also provides users with context about how the current settings fit within the allowable ranges and eliminates the chance that the user will make an invalid entry. No musician wants to be stopped by an error dialog!

Unbounded entry: text edit controls

The primary unbounded entry control is the text edit control. This simple control allows users to key in any alphanumeric text value. Edit fields often are small areas where the user can enter a word or two of data, but they can also be fairly sophisticated text editors. Users can edit text within them using the standard tools of contiguous selection (as discussed in Chapter 18) with either the mouse or keyboard.

Text edit controls are often used either as data-entry fields in database applications (including websites connected to databases), as option entry fields in dialogs, or as the entry field in a combo box. In all these roles, they are frequently called on to do the work of a bounded entry control. However, if the desired values are finite, the text edit control should not be used. If the acceptable values are numeric, use a bounded numeric entry control such as a slider instead. If the list of acceptable values is composed of text strings, a list control should be used so that users are not forced to type.

Sometimes the set of acceptable values is finite but too big to be practical for a list control. For example, an application may require a string of any 30 alphabetic characters excluding spaces, tabs, and punctuation marks. In this case, a text edit control is probably unavoidable even though its use is bounded. If these are the only restrictions, however, the text edit control can be designed to reject nonalphabetic characters and similarly disallow more than 30 characters to be entered into the field. However, this brings up interaction issues surrounding validation.

Validated entry controls

In cases where an unbounded text-entry field is provided, but the field accepts only entries of a certain form, it may be necessary to help users construct a "valid" entry. Typically you do this by evaluating the user's entry after she finishes entering it and displaying an error message if it is invalid. Obviously, this can be irritating for users and ultimately can undermine their effectiveness. Although bounded controls can often eliminate the need for validated entry, when the number of valid entries is large— credit card numbers, for example—validated entry becomes necessary.

Validation controls are a type of unbounded text-entry control with built-in validation and feedback for the user. These controls can validate many formats, such as dates, phone numbers, postal codes, and Social Security numbers.

Although the *validated entry control* is a widespread idiom, most such controls can be improved. The key to successfully designing a validated entry control is to give users generous feedback, as close to real-time as possible, so they can catch an error immediately, understand why the input was an error, and know how to remedy it.

Another improvement is based on the design principle of visually distinguishing elements that behave differently (see Chapter 17). Make validated entry controls visually distinct from nonvalidated controls, whether through the typeface used in the text edit field, the border color, or the background color for the field itself.

Note that password and other security inputs can't adhere strictly to usability concerns (lest they make it usable for hackers and scammers). These kinds of inputs have their own considerations.

Active and passive validation

Some controls reject users' keystrokes as they are entered. When a control actively rejects keystrokes during the entry process, this is an example of *active validation*. A text-only entry control, for example, may accept only alphabetic characters and refuse to allow numbers to be entered. Some controls reject any keystrokes other than numeric. Other controls reject spaces, tabs, hyphens, and other punctuation in real time. Some variants can get pretty intelligent and reject certain numbers based on live calculations. For example, numbers might need to pass a checksum algorithm.

When an active validated entry control rejects a keystroke, it must tell the user it has done so. It also should tell the user why the rejection occurred. If an explanation is offered, users will be less inclined to assume that the rejection is arbitrary (or the product of a defective keyboard). They also will be in a better position to give the application what it wants.

Sometimes the range of possible data is such that the application cannot validate it until the user has completed his entry (rather than at each individual keystroke). The validation then takes place only when the data reaches some threshold—like a set number of characters—or the control loses focus—that is, when the user is done with the field and moves on to the next one. The validation step also must take place if the user closes the dialog—or invokes another function if the control is not in a dialog (such as clicking Place Order on a web page). If the control waits until the user finishes entering data before it edits the value, this is passive validation.

The control may wait until an address is fully entered, such as interrogating a database to check if an address is valid. In such cases each character may be valid by itself, yet the whole may not pass muster. Besides, while the application would know at any given instant whether the address was valid, the user could still legitimately turn to some other task in the form while the name was in an invalid state, intending to return to it later.

A way to address this is by maintaining a countdown timer in parallel with the input and reset it with each keystroke. If the countdown timer ever hits 0, do your validation processing. The timer should be set to approximately half a second. The effect is that as long as the user enters a keystroke faster than once every half-second, the system is extremely responsive. If the user pauses for more than half a second, the application reasonably assumes that he has paused to think, so it goes ahead and analyzes the input so far.

To provide rich visual feedback, the entry field could change colors or reveal an icon to reflect its estimate of the validity of the entered data. For example, the field could show in shades of pink until the application judged the data valid, when it would change to white or green.

Hints

Another good solution to the validation control problem is the *hint*. This little pop-up text looks and behaves much like a ToolTip: It explains the range of acceptable data for a validation control. Whereas a ToolTip appears when the cursor sits for a moment on a control, a hint appears as soon as the control detects an invalid character. (It also can appear, just like a ToolTip, if the cursor sits unmoving on the field for a second or so.) For example, if the user enters a nonnumeric character in a numeric-only field, the application would show the hint near the point of the offending entry, yet without obscuring it. It would say, for example, **ZIP codes can only contain numeric characters, 0–9**. Yes, the user is rejected, but he is not ignored. The hint also works for passive validation, as shown in Figure 21-23.
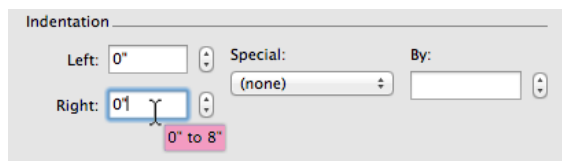
Figure 21-23: The ToolTip idiom is so effective that it could easily be extended to other uses. Instead of yellow ToolTips offering flyover labels for icon buttons, we could have pink ones offering flyover hints for unbounded edit fields. These hints can help eliminate traditional error messages. In this example, if the user enters a value lower than is allowed, the application would replace the entered value with the lowest allowable value and modelessly display a hint that explains the reason for the substitution. The user can enter a new value or accept the minimum without being stopped by an error dialog.

Handling out-of-bounds data

Typically, an edit field is used to enter a numeric value the application needs, such as a font's point size. The user can enter anything he wants, from 5.5 to 500, and the field will accept it and return the value to the owning application. If the user enters garbage, the control must make a decision. In Microsoft Word, for example, if you enter **asdf** as a font point size, the application issues an error dialog informing you "This is not a valid number." It then reverts the size to its previous value. The error dialog is rather silly, but the summary rejection of your meaningless input is perfectly appropriate. But what if you type the value **nine**? The application rejects it with the same curt error message. If instead the control were programmed to think of itself as a numeric entry control, it could perhaps behave better. It doesn't bother us if the application refuses to accept nonnumeric characters (especially if pop-up hints are also employed), but it is incorrect when it says that **nine** is an invalid number.

Units and measurements

It's nice when a text edit control is smart enough to recognize appropriate units. For example, if an application requests a measurement, and the user enters **5"**, **5i**, **5in**, **5 inches**, not only should the control report the result as five, but it also should report inches. If the user enters **5mm**, the control should report it as 5 millimeters. SketchUp, an elegant architectural sketching application, supports this type of feedback. Similarly, well-designed financial analytics applications should know that "5mm" means 5 million.

Say that the field is requesting a column width. The user can enter either a number or a number and an indicator of the measurement system, as just described. Users also could be allowed to enter the word **default**, and the application would set the column width to its default value. The user could alternatively enter **best fit**, and the application would measure all the entries in the column and choose the most appropriate width for the circumstances. This scenario has a problem, however, because the words "default" and "best fit" must be in the user's head rather than in the application somewhere. This is easy to solve, though. All we need to do is provide the same functionality through a combo box. The user can drop down the box and find a few standard widths and the words "default" and "best fit." Microsoft uses this idea in Word, as shown in Figure 21-24.
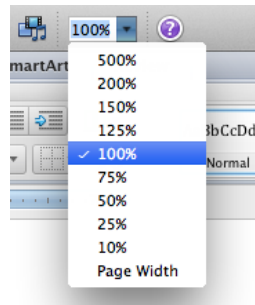
Figure 21-24: The drop-down combo box makes an excellent tool for bounded entry fields because it can accommodate entry values other than numbers. The user doesn't have to remember or type words like "page width" or "whole page," because they are there to be chosen from the drop-down list. The application interprets the words as the appropriate number, and everyone is satisfied.

The user can pull down the combo box, see items like Page Width and Whole Page, and choose the appropriate one. With this idiom, the information has migrated from the user's head into the application, where it is visible and choosable.

Avoid text edit controls for output

The text edit control, with its familiar system font and visually articulated white box, encourages data entry. Yet software developers frequently use the text edit control for read-only output fields. The edit control certainly works as an output field, but using this control for output only is like pulling a bait and switch on your user, and he will not be amused. If you have text data to output, use a text display control, not a text edit control. If you want to show the amount of free space on disk, for example, don't use a text edit field, because novice users are likely to think that they can get more free space by entering a bigger number. At least, that is what the control is telling them with its equivalent of body language.

If you want to output editable information, go ahead and output it in a fully editable text control, and wire it up internally so that it works exactly as it will appear. If not, stick to display controls, described in the next section.

> **DESIGN PRINCIPLE**   *Use noneditable (display) controls for output-only text.*

Display controls

Display controls are used to display and manage the visual presentation of information onscreen. Typical examples include scrollbars and screen splitters. Controls that manage how objects are displayed visually onscreen fall into this category, as do those that display static, read-only information. These include paginators, rulers, guidelines, grids, group boxes, and those 3D lines called dips and bumps. Rather than discuss all these at length, we will focus on a few of the more problematic controls.

Text controls

Probably the simplest display control is the text control, which displays a written message at some location onscreen. The management job it performs is pretty prosaic, serving only to label other controls and to output data that users cannot or should not change.

The only significant problem with text controls is that they are often used where edit controls should be (and vice versa). Users can change most information stored in a computer. Why not allow them to change it at the same point the software displays it? Why

should the mechanism to input a value be different from the mechanism to output that value? In many cases, it makes no sense for the application to separate these related functions. In almost all cases where the application displays a value that could be changed, it should do so in an editable field so that a user can click it and change it directly. Special edit modes are almost always examples of excise.

For years, Adobe Photoshop insisted on opening a dialog to create formatted text in an image. Thus, users could not see exactly how the text would look in the image, forcing them to repeat the procedure several times to get things right. Finally Adobe fixed the problem, letting users edit formatted text directly into an image layer, in full WYSIWYG fashion—as it should be.

Scrollbars

Scrollbars serve a critical need in the modern GUI: They enable smallish rectangles (windows or panes) to meaningfully contain large amounts of information. Unfortunately, they are also typically quite frustrating, difficult to manipulate, and wasteful of pixels. The scrollbar is, without a doubt, both overused and underexamined. In its role as a window content and document navigator—a display control—its application is appropriate.

The singular advantage of the scrollbar—aside from its near-universal availability—is that it provides useful context about where you are in the window. The scrollbar's thumb is the small, dragable box that indicates the current position and, often, the scale of the "territory" that can be scrolled.

Many scrollbars are quite parsimonious in doling out information to users. The best scrollbars use thumbs that are proportionally sized to show the percentage of the document that is currently visible.

While scrollbars are useful for nearly all types of content, scrollbars for pages of text should also show the following:

- The total pages
- The page number (record number, graphic) as we scroll
- A thumbnail of the page as we scroll

Additionally, many scrollbar implementations are stingy with functions. To better help us manage navigation within documents, they should give us powerful tools for going where we want to go quickly and easily:

- Buttons for skipping ahead by pages/chapters/sections/keywords
- Buttons for jumping to the beginning and end of the document

- Tools for setting bookmarks that we can quickly return to
- Annotated scrollbars that visually show the position of searched-for items on the background of the toolbar itself (The thumb of the scrollbar must be partly transparent for this to work well.)

Recent versions of Microsoft Word use scrollbars that exhibit many of these features.

Shortcomings in contextual information aside, one of the biggest problems with scrollbars in a WIMP OS is that they demand a high degree of precision with the mouse. You must position the mouse cursor with great care, taking your attention away from the data you are scrolling. Some scrollbars put both up and down nudge arrows at each end of the scrollbar. For windows that will likely stretch across most of the screen, this can be helpful. For smaller windows, such replication of controls is probably overkill and simply adds to screen clutter. (See Chapter 18 for more discussion of this idiom.)

The ubiquity of scrollbars has resulted in some unfortunate misuse. Most significant here is their shortcomings in navigating time. Without getting too philosophical or theological, we can all hopefully agree that time has no meaningful beginning or end (at least within the perception of the human mind). What, then, is the meaning of dragging the thumb to one end of a calendar scrollbar? (See Figure 21-25.)
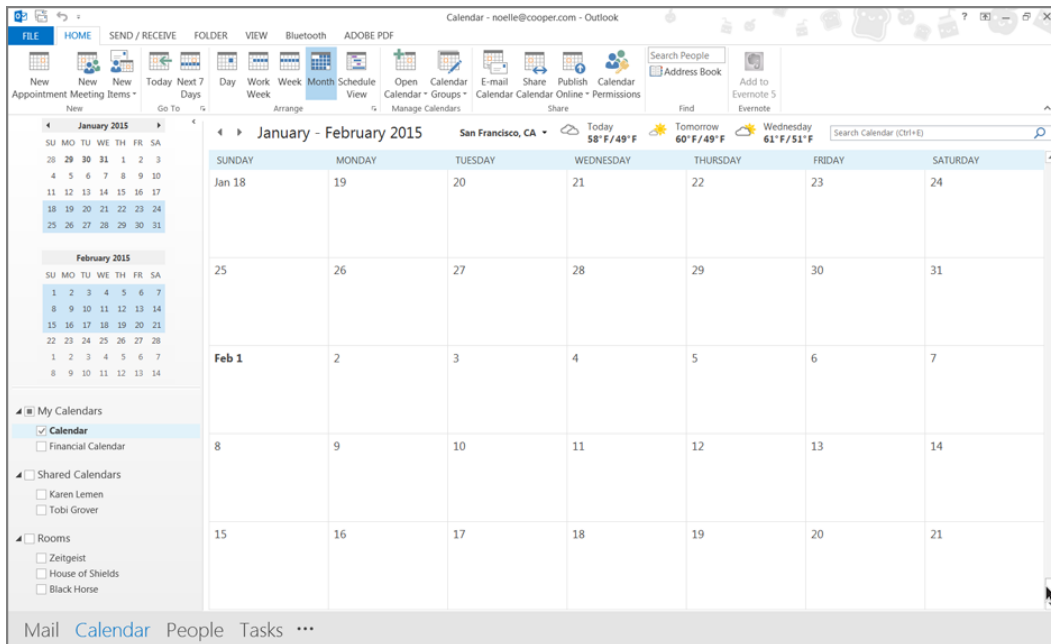
Figure 21-25: This image shows a limitation of using a scrollbar for navigating the endless stream of time. Dragging the thumb all the way to the end of the scrollbar takes the user one year into the future. This seems a bit arbitrary and limiting.

On mobile platforms, and now even in some desktop apps, scrollbars appear only when scrolling takes place. This makes more sense on mobile, where scrolling is performed via gesture— although it also means that a user has to scroll when they don't really want to, in order to discover where in a document they are.

On the desktop, trackpad gestures or mouse wheels (and their capacitive equivalents) also allow scrollbars to be hidden on some platforms, like OS X. They are used primarily to indicate the content's position in its viewport pane rather than for actual scrolling. However, hiding scrollbars on the desktop has some usability gotchas:

- It may not be clear to users that panes are scrollable. This can be fixed by ensuring that some items are partially obscured at the edges of the pane, which is a strong visual cue that scrolling is possible.

- Fine control of scrolling becomes much more difficult. If scrollbars disappear when not in use, it becomes difficult to tweak positioning, because movement is required to activate the fine controls. It's therefore not a wise idea to use hideable scrollbars for any application where fine-tuning scrolling is a necessity.

- With large screens, it's entirely possible that toolbars become hidden by the time the user is able to move her mouse to them, requiring her to scroll just to summon them again.

There are some viable alternatives to scrollbars. One of the best is the document navigator, which uses a thumbnail of the entire document space to provide direct navigation to portions of the document, as shown in Figure 21-26. Many image-editing applications (such as Photoshop) utilize these for navigating around a document when zoomed in. These also can be useful when you navigate time-based documents, such as video and audio. Critical to the success of such idioms is that it is possible to meaningfully represent the big picture of the document in visual form. For this reason, they aren't necessarily appropriate for long text documents. In these cases, the document's structure (in outline form) can provide a useful alternative to scrollbars. A basic example can be seen in Microsoft Word's Document Map.

Splitters

Splitters are useful tools for dividing a sovereign application into multiple related panes in which information can be viewed, manipulated, or transferred. Movable splitters should always advertise their pliancy with cursor hinting. Although it is easy and tempting to make all splitters movable, you should exercise care in choosing which ones to make movable. In general, a splitter should be unable to be moved in such a way that makes a pane's contents unusable. In cases where panes need to collapse, a drawer may be a better idiom.
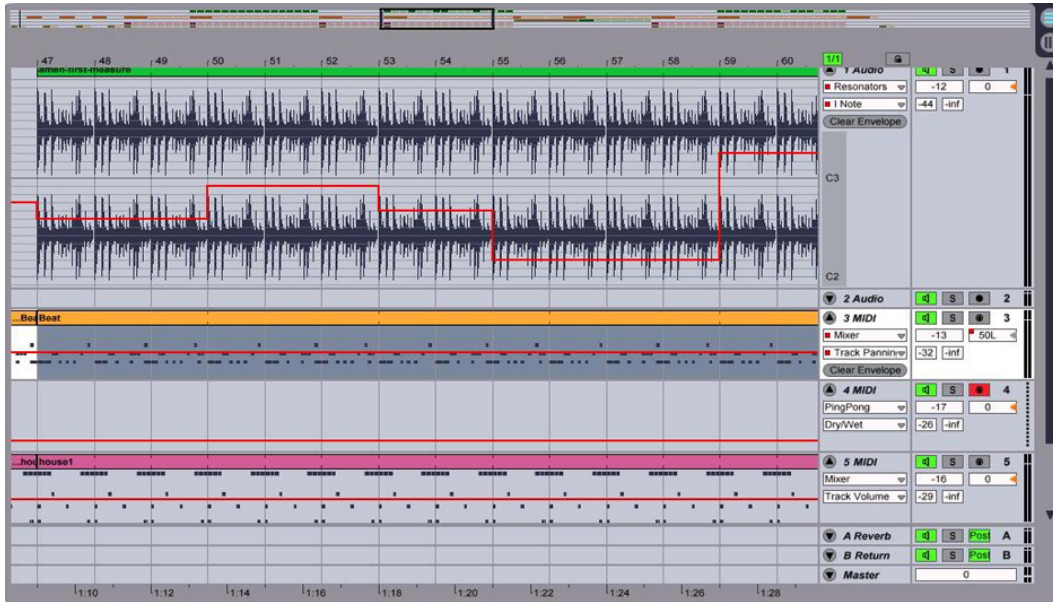
Figure 21-26: Ableton Live features a document navigator on the top of the arrangement screen that provides an overview of the entire song. The black rectangle denotes which part of the song the work area below is zoomed in on. The navigator provides context in a potentially confusing situation and simultaneously provides a direct navigation idiom where the user may move the rectangle to focus on a different part of the song.

Drawers and levers

Drawers are panes in a sovereign application that can be opened and closed with a single action. They can be used in conjunction with splitters if the user can configure the amount by which the drawer opens. You usually open a drawer by clicking a control in the vicinity of the drawer. This control needs to be visible at all times and should be either a latching button/icon button or a lever. A lever behaves similarly but typically swivels to indicate an open or closed state.

Drawers are a great place to put controls and functions that are less frequently used but are most useful in the context of the application's main work area. Drawers have the benefit of not covering the main work area the way a dialog does. Property details, searchable lists of objects or components, and histories are good candidates for putting in drawers.

On mobile devices, horizontally sliding drawers with levers have been employed ubiquitously and successfully to stow primary navigation panes. They take users to different functional screens (using the "hamburger" icon and drawer idiom first popularized and then largely abandoned by Facebook), to content picked from an ordered list as is typical

of many mobile mail applications, or to a UI that provides interactions with the selected drawer item, such as the right-hand chat drawer in the iOS Facebook app.

Dialogs

Dialogs are pop-up windows superimposed over the application's main window. A dialog engages users in a conversation by offering information and requesting some input. When the user has finished viewing the information or selection from options presented, he can dismiss or accept the dialog. The dialog then disappears, returning the user to the main application window.

> **DESIGN PRINCIPLE**
>
> *Put primary interactions in the primary window.*

In the modern era of modeless toolbar and ribbon controls, a hallmark of poor interaction design is a user interface that consists primarily of control-laden modal dialogs. It is very difficult to create fluid interactions if you force users through a maze of pop-up dialogs. If a user is the chef, and the application is the kitchen, a dialog is the pantry. The pantry plays a secondary role, as should dialogs. They are supporting actors rather than lead players, and although they may move the action forward, they should not be the engines of motion. Primary actions and controls for an application belong in its main screen or window.

Appropriate use of dialogs

It's sometimes useful to take users out of their flow to force them to focus on particular questions. Dialogs are appropriate for functions or features that are out of the normal course of things: Anything that is confusing, dangerous, or rarely used can be usefully placed in a dialog. This is particularly true for actions that make immediate and major changes to the application state. Such changes can be jarring, and should be cordoned off from users who are unfamiliar with them. For example, a function that allows wholesale reformatting of a document should be considered a dislocating action. The dialog helps prevent this feature from being invoked accidentally by ensuring that a big, friendly Cancel button is always present, and also by providing the space to show more protective and explanatory information along with the risky controls. The dialog can graphically show users the function's potential effects with a thumbnail of what the changes will look like. And of course, a robust Undo function (see Chapter 15) should be provided for such actions.

Dialogs are appropriate for functions that are out of the main interaction flow.

Dialogs are also well suited for concentrating information related to a single subject, such as the properties of a domain object—an invoice or customer, for example. They also can gather all information relevant to a function performed by an application, such as printing reports. This has obvious benefits to users: With all the information and controls related to a given subject in a single place, users don't have to search around the interface as much for a given function, and navigation excise is reduced.

Dialogs are appropriate for organizing controls and information about a single domain object or application function.

Similar to menus, dialogs can be effective for users who are still learning an application. Because dialogs can be more verbose and structured, they can provide an alternative, more pedagogic interface for functions that are also accessible through direct manipulation in the main application window. However, this sort of interface also can be more effectively placed in expandable, modeless control panes or contextual toolbars in modern desktop apps.

Dialogs serve two masters: the frequent user who is familiar with the application and uses dialogs to control its more advanced or dangerous facilities, and the infrequent user who is unfamiliar with the scope and use of the application and who uses dialogs to learn the basics. This dual nature means that dialogs must be compact and powerful, speedy and smooth, yet clear and self-explanatory. These two goals may seem contradictory, but they can actually be useful complements. A dialog's speedy and powerful nature can contribute directly to its power of self-explanation.

Basic dialog interactions

Most dialogs contain a combination of informative text, interactive controls, and associated text labels. Although some rudimentary conventions apply, the idiom's diverse applications mean that there are few hard and fast rules. It is important to create dialogs in accordance with good visual interface design practices and ensure that they use GUI controls appropriately. In particular, a dialog should exhibit a strong visual hierarchy, visual groupings based on similarities in subject, and a layout based on the conventional reading order (left to right and top to bottom for Western writing systems). For more details about these visual interface design practices, see Chapter 17.

When instantiated, a dialog should appear on the topmost visual layer so that it is obvious to the user. Subsequent user interactions may obscure the dialog with another dialog or application, but it should always be obvious how to restore the dialog to prominence.

A dialog should have a title that clearly identifies its purpose. If the dialog is a function dialog, the title bar should contain the function's *action*—the verb, if you will.

If the dialog is used to define an object's properties, the title bar should contain that object's name or description. The properties dialogs in Windows work this way. When you request the Properties dialog for a directory named Backup, the title bar says Backup Properties. Similarly, if a dialog is operating on a selection, it can be useful to reflect a truncated version of the selection in the title to keep users oriented.

Most conventional dialogs have at least one *terminating command*—a control that, when activated, causes the dialog to shut down and go away (and most often some other function that was the point of the dialog). Most modal dialogs offer at least two pushbuttons as terminating commands, OK and Cancel, although the Close box in the upper-right corner is also a terminating command idiom.

It is technically possible for dialogs not to have terminating commands. Some dialogs are unilaterally launched and dismissed by the application—for reporting on the progress of a time-consuming function, for example—so their designers may have omitted terminating commands. This is poor design for a variety of reasons, as we will see.

Modal and modeless dialogs

Dialogs come in two flavors: modal and modeless. *Modal dialogs* are by far the more common variety. After a modal dialog opens, the owner application cannot continue until the dialog is closed. It stops all proceedings in their tracks. Clicking any other window belonging to the application only gets the user a rude beep for his trouble. All the controls and objects on the surface of the owner application are deactivated for the duration of the modal dialog. Of course, the user can activate *other* applications while a

modal dialog is up, but the dialog stays there indefinitely. When the user goes back to the application, the modal dialog is still there waiting.

In general, modal dialogs are easier for users (and designers) to understand. The operation of a modal dialog is quite clear, saying to users, "Stop what you're doing and deal with me now. When you're done, you can return to what you were doing." The rigidly defined behavior of the modal dialog means that, although it may be abused, it is rarely misunderstood. There may be too many modal dialogs, and they may be weak or stupid, but their purpose and scope usually are clear to users.

Some modal dialogs operate on the entire application or the entire active document. Others operate on the current selection, in which case the user can't change the selection after summoning the dialog. This is the most important difference between modal and modeless dialogs.

Because modal dialogs stop only their owning applications, they are more precisely described as *application-modal*. It is also possible to create a *system-modal* dialog that brings every application in the system to a halt. In most cases, applications should never have one of these. Their only purpose is to forestall or report catastrophic occurrences (such as the hard disk melting) that affect either the entire system or a real-world process.

Modeless dialogs are less common than their modal siblings. When a modeless dialog opens, the parent application continues without interruption. It does not stop the proceedings, and the application does not freeze. The various facilities and controls, menus, and toolbars of the main window remain active and functional. Modeless dialogs have terminating commands, too, although their conventions are far weaker and more confusing than for modal dialogs.

A modeless dialog is a much more difficult beast to use and understand, mostly because the scope of its operation is unclear. It appears when you summon it, but you can go back to operating the main window while it stays around. This means that you can change the selection while the modeless dialog is still visible. If the dialog acts on the current selection, you can select, change, select, change, select, and change all you want. For example, Microsoft Word's Find and Replace dialog allows you to find a word in text (which is automatically selected), make edits to that word, and then pop back to the dialog, which has remained open during the edit.

In some cases, you can also drag objects between the main window and a modeless dialog. This characteristic makes them effective as tool or object palettes.

Differentiating modal and modeless dialogs

If you have limited time and resources to deal with interaction design issues, we recommend leaving modeless dialogs pretty much the way they are, while adopting the following guiding principles and applying them consistently.

> **DESIGN PRINCIPLE**
>
> *Differentiate modeless dialogs from modal dialogs.*

First, modal dialogs *must* include one or more terminating commands, usually in the form of large pushbuttons at the bottom of the dialog.

Second, modeless dialogs should *not* use terminating command buttons. Instead, they should use the Close control in its window title bar.

> **DESIGN PRINCIPLE**
>
> *Do not use terminating button commands for modeless dialogs.*

Third, modal dialogs should *not* use Close controls in their title bars, not only to help differentiate them from their modeless brethren, but because the function executed by the close control may be unclear to users. (Does clicking it cancel, or confirm what's been entered in the dialog?)

Issues with modal dialogs

One particular modal dialog variation to avoid is the use of terminating buttons that change from Cancel to Apply, or from Cancel to Close, depending on whether the user has taken an action. This dynamic change is disconcerting, hard to interpret and, at worst, frightening and inscrutable. These labels should *never* change. If the user hasn't selected a valid option but clicks OK anyway, the dialog should assume the user means "Dismiss the box without taking any action," for the simple reason that this is what the user actually did.

> **DESIGN PRINCIPLE**
>
> *Don't dynamically change the labels of terminating buttons.*

The cognitive strength of modal dialogs is their rigidly consistent OK and Cancel buttons. In modal dialogs, the OK button means "Accept my input and close the dialog." In modal dialogs, the Cancel button means "Abandon my input and close the dialog."

Issues with modeless dialogs

Many modeless dialogs are implemented awkwardly. Their behavior is inconsistent and confusing. They are visually very similar to modal dialogs, but they are functionally very different. They have few established behavioral conventions, particularly with respect to terminating commands.

Much of the confusion arises because users are so familiar with the behavior of modal dialogs. A modal dialog can adjust itself for the current selection at the instant it was summoned. It can do this with assurance that the current selection won't change during its lifetime. Conversely, the current selection can easily change during the lifetime of a modeless dialog. What should the dialog do then? For example, if a modeless dialog modifies text, what should it do if we now select a nontext object in the main window? Should the controls in the dialog become disabled, change, or disappear? Questions such as this require careful analysis, as well as close examination of persona needs, goals, and mental models. Consequently, modeless dialogs can be much more challenging to design and implement than modal dialogs, which avoid these issues by freezing the application state.

Modeless dialogs frequently have several buttons that immediately invoke various functions. The dialog should not close when one of these function buttons is clicked. It is modeless because it stays around for repetitive use and should close only when the window close control is clicked.

Modeless dialogs must also be incredibly conservative of pixels. They will be staying around on the screen, occupying the front and center location, so they must be extra careful not to waste pixels on anything unnecessary.

Modeless dialogs and Undo

Because the controls on a modeless dialog are always live, the equivalent concept is clouded in confusion. The user doesn't conditionally configure changes in anticipation of a terminal Execute command as he does with a modal dialog.

The changes made from a modeless dialog are immediate, occurring as soon as any entry or control in the dialog is changed. There is no concept of "Cancel all my actions." Dozens of separate actions may have been performed on a number of selections. The proper

idiom for this is the Undo function, which is active application-wide for all modeless dialogs. This all fits together logically, because the Undo function is unavailable if a modal dialog is up, but it is still usable with modeless ones.

The only consistent terminating action for modeless dialogs is Close, which should be accessed from the window close control on the title bar. Furthermore, if the Close button actuates a function in addition to shutting the dialog, you have created a modal dialog that should follow the conventions for the modal idiom instead.

Modeless dialogs and sidebars

Any modeless dialog that is intended to provide persistent support to activities in the main window is a good candidate for recasting as a sidebar control pane (see Chapter 18). These have all the advantages of modeless dialogs without forcing users to manage the set of controls within a separate floating window that must be moved out of the way of the job at hand. As screen resolutions have increased, there is less and less reason for banks of controls intended for frequent use in the construction of a document to be placed anywhere but in toolbars or sidebar panes.

Five purposes of dialogs

The concepts of modal and modeless dialogs are derived from developer terms. They affect our design, but we should also examine dialogs from a more goal-directed point of view. In that light, five fundamental types of information are useful to convey with a dialog: property, function, process, notification, and bulletin.

Property dialogs

Property dialogs allow users to view and change settings or attributes. A properties dialog generally modifies the current selection, but it also can be used to set application global properties. (Figure 21-31, shown later in this chapter, is a somewhat over-the-top example.) You can think of property dialogs as control panels containing configuration controls for the selected object.

Property dialogs typically are modeless. When you modify the properties of a selection, these dialogs often are more useful when they are implemented as a task pane or sidebar (see Chapter 18), rather than as a standard dialog—especially a modal one. This is true unless they are being used for set-once-and-forget-it or other infrequently accessed properties.

Function dialogs

Function dialogs are usually launched from a menu. They are most frequently modal dialogs, and they control a single function such as printing, modifying large numbers of database records, inserting objects, or spell checking.

Not only do function dialogs allow users to initiate an action, but they also often allow users to configure the details of the action's behavior. In many applications, for example, when the user wants to print, she uses the Print dialog to specify which pages to print, the number of copies to print, which printer to output to, and other settings directly relating to the print function. The terminating OK button on the dialog not only confirms the settings and closes the dialog but also executes the print operation.

This common technique combines two functions: Configuring the function and invoking it. Just because a function *can* be configured, however, doesn't necessarily mean that the user will *want* to configure it before every invocation. It's often better to make these two functions separately accessible (although they should also be seamlessly linked).

Many functions available from modern software are quite flexible and have a number of options. If you don't segregate configuration and actuation, users can be forced to confront considerable complexity, even if they want to perform a routine task in a simple manner.

Process dialogs

Process dialogs are launched at an application's discretion rather than at the user's request. They indicate that the application is busy with some internal function and that performance in other areas is likely to degrade.

When an application begins a process that will take perceptible quantities of time (anything over a second), it must make clear that it is busy, but that everything is otherwise normal. If the application does not indicate this, the user interprets this as rudeness at best; at worst, he assumes that the application has crashed and that he must take drastic action.

> **DESIGN PRINCIPLE**
>
> *Inform the user when the application is unresponsive.*

Many applications currently rely on active *wait-cursor hinting*, turning the cursor into something like a spinning beach ball or hourglass, and disabling further clicks until the process is complete. A better, more informative solution is a process dialog.

Each process dialog should make clear to users:

- That a time-consuming process is happening
- That things are completely normal
- How much more time the process will take
- How many more objects or items need to be operated on (when applicable)
- How they can cancel the operation and regain control of the application

The mere presence of the process dialog satisfies the first requirement, alerting users to the fact that a process is occurring. Satisfying the third requirement can be accomplished with a *progress meter* of some sort, showing the relative percentage of work performed and how much is yet to go. Satisfying the second requirement is the tough one. The application can crash (or lose connection with a server) and leave the dialog up, lying mutely to the user about the operation's status. The process dialog must continually show, via time-related movement, that things are progressing normally. The meter should show the progress relative to the total *time* the process will consume rather than the total size of the process. Fifty percent of one process may be radically different in time from 50 percent of the next process.

The user's mental model of the computer executing a time-consuming process will quite reasonably be that of a machine cranking along. A static dialog that merely announces that the computer is Reading Disk may *tell* users that a time-consuming process is happening, but it doesn't *show* that this is true. The best way to show the process is by using animation in the dialog. Users get the sense that the computer is really *doing* something: The sensation that things are working normally is visceral rather than cerebral, and users—even expert users—are reassured.

The user may have second thoughts about how long the operation will take and decide to postpone it. However, if the user realizes he issued the wrong command and wants to cancel the operation, not only will he want the operation to stop, but he also will want already executed portions of the operation to be undone.

A good approach would be to have two buttons on the dialog—one labeled Cancel and the other labeled Pause. Users could then choose the one they really want.

You should think about something else when considering the need for process reporting. Because a dialog is a separate room, designers should question whether a process reported by a dialog is really a function separate from what is happening in the main window. If the function is an integral part of what is shown in the main window, the status of that function should be shown in the main window. For example, Windows uses a Copy dialog, but isn't copying a file fundamental to what the Explorer does? A more subtle animation could have been built right into the main Explorer window.

Process dialogs are, of course, much easier to develop than building animation right into the main window of an application. They also provide a convenient place for the Cancel button, so it is a reasonable compromise to fling up a process dialog for the duration of a time-consuming task. But don't lose sight of the fact that, by doing this, we are still going to another room for a this-room function. It is an easy solution, but not the correct one. Web browsers such as Google Chrome and Microsoft Internet Explorer provide a much more elegant solution. Because loading web pages is so intrinsic to their operation, the progress indicator (an animated circle) is displayed on the currently loading browser tab, as shown in Figure 21-27.
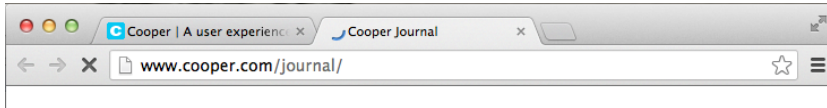
Figure 21-27: Web browsers such as Google Chrome don't launch a process dialog every time they load a page. Rather, a progress indicator is displayed in the tab for the currently loading page. Other browsers place this indicator in the URL field or in a status bar at the bottom of the window. This allows users to easily understand what's going on without obscuring their view of the partially loaded web page in front of them.

Notification dialogs

Notification dialogs report important messages that are either the result of triggered events or the result of communications from other users. Alarms, appointments, and e-mail or IM notifications are good examples. These are in contrast to system-generated alerts (discussed in the previous section) that are launched unbidden by the app purely to communicate its own internal problems or successes.

Mobile products support heavy use of notifications, which include both communications and other on-the-go information based on changes in time and location. Some of these idioms have become more prevalent on the desktop and in web apps, since communication apps span these platforms as well.

Mobile platforms have done a good job of collecting notifications into *notification centers*, which permit viewing of notifications after the fact: Users might be unable to address a message or triggered alarm until they have stopped driving, taken a seat on the bus they were catching, or finished a phone call they were in.

Notifications frequently appear as small pop-up windows or drawers in the periphery of the screen, with a subtle animation to call attention to themselves. They can linger modelessly or can close after a short delay, leaving behind a marker or badge in the notification center to alert the user that something still needs his or her attention. Notifications work

well in this manner as long as they are also collected for review in a top-level notification center *and* the arrival of new, unread notifications is clearly, noticeably, and persistently marked in the interface.

Bulletin dialogs

Bulletin dialogs, like process dialogs, are launched, unrequested, by the application. The three types of bulletin dialogs are errors, alerts, and confirmations. Each of these reports on, or requires a user decision about, the application's internal state. Each also suffers from frequent misuse. Together they represent some of the worst product interactions, if only by virtue of their vapidity and omnipresence.

The ubiquitous error dialog best characterizes the bulletin dialog. Normally, the application's name is shown in the caption bar, and a brief text description of the problem is displayed in the body. A graphic icon that indicates the problem's class or severity, along with an OK button, usually completes the ensemble. Sometimes a button to summon online help is added. An example from Word is shown in Figure 21-28.
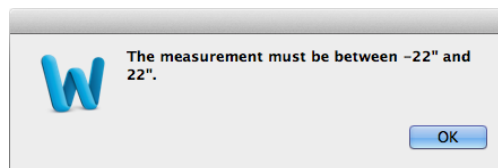
Figure 21-28: Here's a typical bulletin dialog. It is never requested by the user but is always issued unilaterally by the application when the application fails to do its job or when it just wants to brag about having survived the procedure. This dialog in effect blames the user, rather than helping solve the problem. Users interpret this as saying "The measurement must be between –22 inches and 22 inches, and you are a buffoon for not knowing that fundamental fact. You are so stupid that I won't even try to correct it for you!"

Bulletin dialogs normally are *application-modal*: They stop all further progress of the application until the user issues a terminating command—like tapping the OK button. This type of bulletin is *blocking* because the application cannot continue until the user responds.

It is also possible for an application to launch a bulletin dialog and then unilaterally dismiss it after a short delay. This type of bulletin is *transitory* because the dialog disappears and the application continues without user intervention.

Transitory bulletins are sometimes used for error reporting. An application that launches an error dialog to report a problem may correct the problem itself or may detect that the problem has disappeared via some other agency. Some developers issue an error or alert merely as a warning—Your disk is getting full—and dismiss it after, say, 10 seconds. This type of behavior is fraught with usability issues.

Errors, alerts, and confirmations should pause the application or, at the very least, maintain their presence until the user takes notice. If they don't, the user may be unable to read the bulletin fully, or, if he is looking away, he may not see it—or, worse yet, see only a fleeting glimpse. He will be justifiably suspicious that he has missed something important, something that will come back to haunt him, and not know how to get the message back. He will begin to worry about what he missed. Was it an important bit of intelligence that he will regret not knowing? Is something terribly wrong? This is true even if the problem goes away by itself.

If something is worth saying with a dialog, it's worth ensuring that the user definitely gets the message. Because a transitory notification can't make that guarantee, it should never be used in the role of error reporting or confirmation gathering. Error, alert, and confirmation bulletins should almost always be blocking.

> **DESIGN PRINCIPLE**   *Never use transitory dialogs as error messages, alerts, or confirmations.*

Property, function, and even notification dialogs are intentionally requested by users—they serve users. The application, however, issues bulletin dialogs—they serve the application, most often at the user's expense. As we shall see, most of these annoying and often useless dialogs *should simply be eliminated* in favor of more helpful and supportive interaction patterns. We'll discuss this in detail at the end of the chapter.

Managing property and function dialogs

Even if you are conscientious about the use and organization of property and function dialogs, they can easily become quite crowded with controls, options, and the like. There are several common strategies for managing this crowding so that these dialogs maintain their usefulness.

Tabbed dialogs

In the 1990s, *tabbed dialogs* became an established standard in the world of commercial software. The idiom, while useful, became an unfortunately convenient way for developers to cram piles of only vaguely related functions into a single dialog.

On a more positive note, this idiom also allows application objects with numerous properties to have correspondingly rich property dialogs without making those boxes excessively large and crowded with controls (see Figure 21-29). Many function dialogs that were previously jam-packed with controls now make better use of their space. Before tabbed dialogs, this problem was more clumsily solved with expanding and cascading dialogs, which we'll discuss shortly.
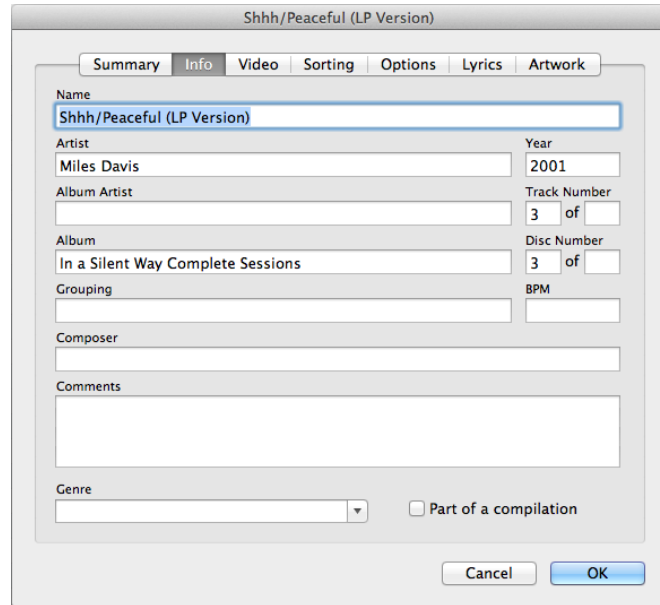
Figure 21-29: This is a tabbed dialog from iTunes. Combining the different properties of a song in one dialog is effective for users because they have a single place to go to find such things. Note that the terminating controls are correctly placed outside the tabbed pane, in the lower right.

More controls won't necessarily mean that users will find the interface easier to use or more powerful. The contents of the various tabs must have a meaningful rationale for being together. Otherwise, this ability is just another way to build a product according to what is easy for developers, rather than what is good for users.

The tabs in a dialog should be organized to provide either increased depth or increased breadth on a well-defined topic. To organize for breadth, each tab should cover parallel, alternative aspects of the primary topic, the way song properties from iTunes, shown in Figure 21-29, address a variety of properties and settings for the song that would be unwieldy in a single pane. In the case of organizing for more depth, each tab should probe the same aspect of one topic in greater depth. The commonly employed Advanced tab is an example of this strategy.

Tabs are successful because the idiom follows many users' mental model of how things are normally stored. The various controls are grouped in several parallel panes, one level deep. But this idiom can also be abused.

Because it's easy to cram so many controls into a tabbed dialog, the temptation is great to add more and more tabs to a dialog. The now-defunct Options dialog from Microsoft Word, shown in Figure 21-30, illustrates this problem. The 10 tabs are far too numerous to show in a single line, so they are stacked two deep. The problem with this idiom, called *stacked tabs*, is that the user has to do a significant amount of work to find the single option she wants to change. While the labels of the tabs may give her some help, she is still forced to scan the contents of several tabs while switching between them. And as if that isn't enough, when she clicks a tab in the back row, the entire row of tabs moves forward, pushing the other two rows to the back. Few users are happy with this, because it's disconcerting to click a tab and then have it move out from under the mouse. It's no wonder that Microsoft has largely abandoned this idiom.
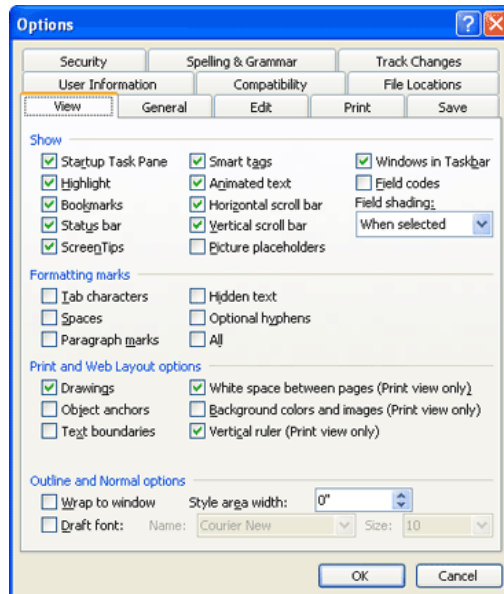
Figure 21-30: The now-defunct Options properties dialog from Word was an abuse of the tabbed dialog idiom. The problem was that users had to do a lot of work to find the option they were looking for.

Stacked tabs illustrate the following axiom of user-interface design: All idioms, regardless of their merits, have practical limits. A group of five radio buttons may be excellent, but a group of 50 is ridiculous. Five or six tabs in a row is fine, but adding enough tabs to require stacking greatly reduces the idiom's usefulness.

A better alternative would be to use several separate dialogs with fewer tabs on each. In Figure 21-30, Options is just too broad a category, and lumping all this functionality in one place doesn't do users any favors. There is little connection among the 12 panes, so there is little need to move among them. This solution may lack a certain programming elegance, but it is much better for users.

Expanding dialogs

Expanding dialogs unfold to expose more controls. The dialog shows a button marked More, or uses a down-pointing arrow icon button that toggles to point up when the dialog has been expanded. When the user clicks it, the dialog grows to occupy more screen space. The newly added portion of the dialog contains added functionality, usually for advanced users or more-complex, but related, operations. The Find and Replace dialog in Microsoft Word, shown in Figure 21-31, is a familiar example of this idiom.
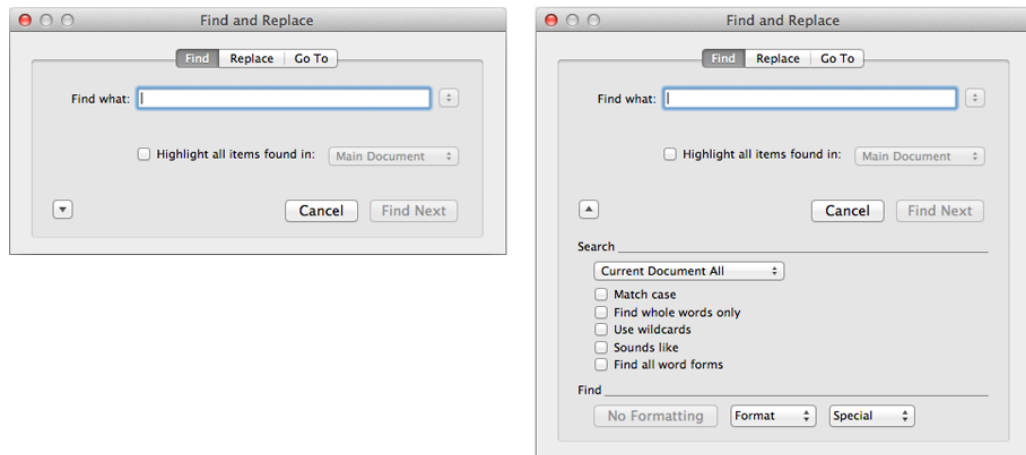
Figure 21-31: The Microsoft Word Find and Replace dialog is an example of an expanding dialog. The image on the left shows it in its original state; the one on the right is what happens after the arrow toggle button is clicked.

Expanding dialogs give infrequent or first-time users the luxury of not having to confront the complex facilities that more frequent users don't find confusing or overwhelming. Think of the dialog as being in either beginner or advanced mode. However, these types of dialogs must be designed with care. When an application has one dialog for beginners and another for experts, it all too often simultaneously insults the beginners and hassles the experts. It's usually a good idea for the dialog to remember what mode it was used in the last time it was invoked. Of course, this means you should always remember to include a Less command to return the dialog to simple beginner mode.

Cascading dialogs

Cascading dialogs are a diabolical idiom whereby controls, usually pushbuttons, in one dialog summon another dialog in a hierarchical pile. The second dialog usually covers the first one either partially or completely. Sometimes the second dialog can summon yet a third one. What a mess! Thankfully, cascading dialogs have fallen from grace and are hard to find anymore. Figure 21-32 shows an example taken from Windows Vista.
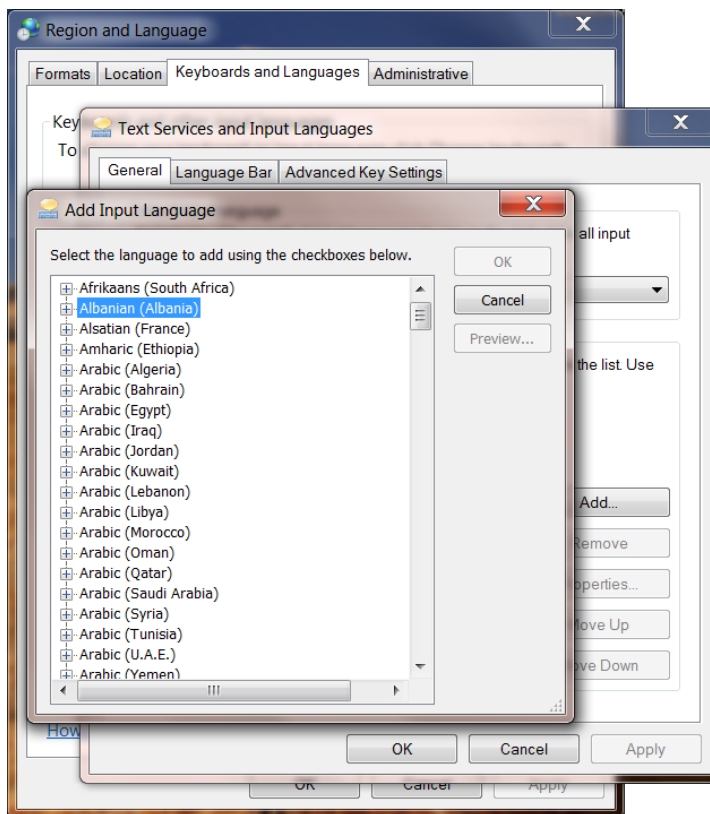
Figure 21-32: You can still find a few (terrible) cascading dialogs in Windows. Each dialog offers a set of terminating buttons. The resulting excise and ambiguity are not helpful.

It is, simply put, hard to understand what is going on with cascading dialogs. Part of the problem is that the second dialog covers at least part of the first. That isn't the big issue. After all, combo boxes and pop-up menus do that, and some dialogs can be moved. The real confusion comes from the presence of a second set of terminating buttons. What is the scope of each Cancel? What are we OKing?

If you find your application requiring cascading dialogs for anything other than really obscure stuff that your users generally won't need, you should take another look at your interaction framework. It may have structural problems that could be remedied using tabbed dialogs, sidebars, or even toolbars (from which a dialog could be launched).

Dialogs can become useful assistants that help your users accomplish their goals, instead of dreaded roadblocks that confound them at every step. By keeping your dialogs manageable, and invoking them only when their functions are truly those that belong in another room, you will go far toward maintaining your users' flow and ensuring their success and gratitude.

Eliminating Errors, Alerts, and Confirmations

As we've already discussed, bulletin dialogs—errors, alerts, and confirmations—represent some of the most problematic digital product interactions, enough so that websites and blogs chronicle the worst examples of these idioms. In most cases, bulletin dialogs can be replaced with interactions that better serve user goals and needs. We'll discuss why and how in this section.

Error dialogs

Probably no user interface idiom is more annoying—or more historically misused—than the error dialog. They are often poorly written, unhelpful, rude, and, worst of all, don't even help prevent the error. Although they are on the wane, it's important to be vigilant to root them out of your application whenever and wherever possible.

What's wrong with error dialogs?

Users don't need to be told they've made an error. Rather, they need help in avoiding errors and their consequences. We believe that applications have a responsibility to try to make things right for users, rather than summarily rejecting their input.

Since the early days of computing, developers have largely left unexamined the notion that the proper way for software to interact with humans was to demand input and to complain when the human failed to meet the application's expectations. Examples of this unfortunate tradition exist wherever software insists that users do things its way rather than adapting machine behavior to the needs of humans. Nowhere has this been more prevalent than in the use of error messages.

Humans have emotions and feelings; applications don't. When one module of code rejects the input of another, the rejected module doesn't care; it doesn't scowl, get hurt, or seek counseling. Humans, on the other hand, get angry when they are flatly told they did something stupid. Make no mistake: When the user sees an error message, it is as if someone has told her she is stupid (see Figure 21-33). Unsurprisingly, users hate this. Despite this inevitable reaction, some developers use error messages anyway. They don't know how else to create reliable software.
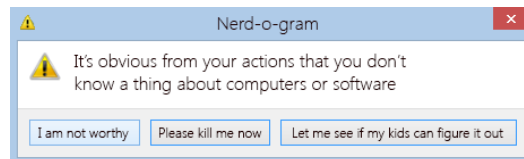
Figure 21-33: No matter how nicely your error messages are phrased, this is how they will be interpreted.

The assumption that users need to be told when they are wrong is false in most circumstances. How important is it for you to know that you requested an invalid type size? Most of the time, applications can and should make reasonable substitutions rather than scolding users.

We consider it impolite to tell people when they have committed a social faux pas. Telling someone he has a bit of lettuce stuck to his tooth or that his fly is unzipped is equally embarrassing for both parties. Sensitive people look for ways to bring the problem to the victim's attention without letting others notice. Yet the default tool for broaching such a topic with the user is a big, bold box in the middle of the screen that stops all the action and emits a scornful beep. Does that really seem appropriate?

Many designers and developers imagine that their error messages alert users to serious problems. This is a widespread misconception. Most error messages inform users of the application's inability to work flexibly and are an admission of stupidity on the application's part. In other words, to most users, error messages are seen not just as the application stopping the proceedings, but as stopping the proceedings with *idiocy*. We can significantly improve the quality of our interfaces by eliminating error dialogs.

Whose mistake is it, anyway?

Conventional wisdom says that error messages tell users when they have made a mistake. Actually, most error messages simply report when the application gets confused. Users make far fewer substantive mistakes than imagined. Typical "errors" consist of issues such as the user's inadvertently entering an out-of-range number, or entering an alphabetic character where a number was expected.

When the user enters something unintelligible by the application's standards, whose fault is it? Is it the user's fault for not knowing how to use the application properly, or is it the fault of the application for not making choices and their effects more clear to users?

Information that is entered in an unfamiliar sequence is often considered an error by software, but people don't have this difficulty with unfamiliar sequences. Humans know how to wait, to bide their time until the story is complete. Software usually jumps to the erroneous conclusion that out-of-sequence input means wrong input, so it issues an error message.

For example, when a user creates an invoice for a customer without an ID number, most applications reject the entry. They stop the proceedings with the idiocy that the user must enter a valid customer number right now. Alternatively, the application could accept the transaction with the expectation that a customer number will eventually be entered, or that the user may even be trying to create a new customer. The application could provide rich modeless visual feedback (as discussed at length in Chapter 15) showing that the customer ID hasn't been entered yet. Then it could watch to make sure that the user enters the necessary information to make that ID valid before the end of the session, or even at the end of the month's book closing.

If a person forgets to fully explain things to the application, it can, after some reasonable delay, provide more insistent signals to the user. At the end of a session, the application can make sure that any irreconcilable transactions are apparent. The application doesn't have to bring the proceedings to a halt with an error message. After all, the application will remember the transactions, so they can be tracked down and fixed. As long as users remain well-informed throughout, there shouldn't be a problem. The trick is to inform without stopping the proceedings. We'll discuss this idea a bit later in this chapter.

Error messages don't work

Error messages have a final irony: They don't actually prevent users from making errors. We imagine that users are staying out of trouble because our trusty error messages keep them straight, but this is a delusion. What error messages really do is prevent the application from getting into trouble. In most software, the error messages stand like sentries where the application is most sensitive, not where users are most vulnerable, setting in concrete the idea that the application is more important than users. Users get into plenty of trouble with our software regardless of the quantity or quality of its error messages. All an error dialog can do is keep me from entering letters in a numeric field. It does nothing to protect me from entering the wrong numbers, which is a much more difficult design task.

How to eliminate error messages

We can't eliminate error messages by simply discarding the code that shows the actual error bulletin dialog and letting the application crash if a problem arises. Instead, we need to redesign applications so that they are no longer susceptible to the problem. We must replace the error dialog with more robust software that prevents error conditions from arising, rather than having the application merely complain when things don't go precisely the way it wants. Like vaccinating it against a disease, we make the application immune to the problem, and then we can toss the message that reports it. To eliminate the error message, we must first reduce the possibility of users making errors. Instead of assuming error messages are normal, we need to think of them as abnormal solutions to rare problems—as surgery instead of aspirin. We need to treat them as an idiom of last resort.

The software designer must reevaluate the entire concept of invalid data. When it comes from a human, the software should assume that the input is correct, simply because the human is more important than the code. Instead of software rejecting input, it must work harder to understand and reconcile confusing input. An application may understand the state of things inside the computer, but only the user understands the state of things in the real world. Remember, the real world is more relevant and important than what the application thinks.

Making errors impossible

Making it impossible for users to make errors is the best way to eliminate error messages. By using bounded widgets (such as spinners and drop-down list boxes) for data entry, we can prevent users from entering invalid values. Instead of forcing the user to key in his selection, present him with a list of possible selections from which to choose. Instead of making the user type in a zip code, for example, look it up from the entered address. In other words, make it impossible for the user to enter an erroneous state.

Another excellent way to eliminate error messages is to make the application smart enough that it no longer needs to make unnecessary demands. Many error messages say things like "Invalid input. User must type xyz." Why can't the application, if it knows what the user must type, just enter xyz by itself and save the user the tongue-lashing? Instead of demanding that the user find a file on a disk, introducing the chance that the user will select the wrong file, the application should remember which files it has accessed in the past and allow the user to select from that list. ("Recent file" lists under the "File" menu accomplish this nicely.) Another example is designing a system that gets the date from the internal or an Internet clock instead of asking for input from users.

Undoubtedly, these solutions cause more work for developers. However, it is the developer's job to satisfy users, not vice versa. If the developer thinks of the user as just another input device, it is easy to forget the pecking order in the world of software design.

Users are unsympathetic to the difficulties developers face. They don't see the technical rationale behind an error message. All they see is the application's unwillingness to deal with things in a human way. They see all error messages as some variant of the one shown in Figure 21-34.
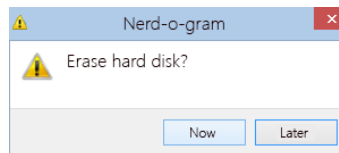
Figure 21-34: This is how most users perceive error bulletin dialogs. They see them as Kafkaesque interrogations, with each successive choice leading to a blacker pit of retribution and regret.

One of the problems with error messages is that they are usually *ex post facto* reports of failure. They say, "Bad things just happened, and all you can do is acknowledge the catastrophe." Such reports are not helpful. And these dialogs almost always come with an OK button, requiring the user to be an accessory to the crime. These error messages are reminiscent of the scene in old war movies where an ill-fated soldier steps on a landmine while advancing across the battlefield. He and his buddies clearly hear the click of the mine's triggering mechanism. The soldier realizes that although he's safe now, as soon as he removes his foot from the mine, it will explode, taking some large and useful part of his body with it. This is likely the feeling users will get when they see your app's ill-considered error messages.

Positive feedback

One of the reasons why software is hard to learn is that it so rarely gives positive feedback. People learn better from positive feedback than from negative feedback. People want to use their software correctly and effectively, and they are motivated to learn how to make the software work for them. They don't need to be slapped on the wrist when they fail. They do need to be rewarded, or at least acknowledged, when they succeed. They will feel better about themselves if they get approval, and that good feeling will be reflected to the product.

Advocates of negative feedback can cite numerous examples of its effectiveness in guiding people's behavior. This evidence is true, but almost universally, the context of effective punitive feedback is getting people to refrain from doing things they want to do but shouldn't: things like not driving over 55 mph, not cheating on their spouses, and not fudging their income taxes. But when it comes to helping people do what they want to do, positive feedback is best. If you've ever learned to ski, you know that a ski instructor who yells at you doesn't help the situation.

> **DESIGN PRINCIPLE**
>
> *Users get humiliated when software tells them they failed.*

Keep in mind that we are talking about the drawbacks of negative feedback from a software application. Negative feedback from another person, although unpleasant, can be justified in certain circumstances. You could say that a mean coach helps your mental toughness for competition, and the imperious professor at least prepares you for the vicissitudes of the real world. But being given negative feedback by a machine is an insult. The drill sergeant and professor are at least human and have bona fide experience and merit. But to be told by software that you have failed is humiliating and degrading. Nothing that takes place inside a computer is helped by humiliating or degrading a human user.

Aren't there exceptions?

Are there exceptions to the rule of eliminating error messages? Not many. As our technological powers have grown, the portability and flexibility of our digital systems have grown too. Modern computers and smart devices can be connected to and disconnected from networks and peripherals without having to first power down. This means that it is now normal for digital hardware to appear and disappear on an ad hoc basis. Printers, speakers, and file servers can come and go like the tides. With the development of Wi-Fi and Bluetooth wireless protocols, our devices can frequently connect to and disconnect from each other. Is it an error that the computer crashed and restarted without the user's selecting "shut down"? Is it an error if you print a document, only to find that no printers

are connected? Is it an error if the file you are editing normally resides on a drive that is no longer reachable?

None of these occurrences should be considered errors from the user perspective. If you open a file on the server and begin editing it, and then you go out to a restaurant for lunch, taking your notebook with you, the application should see that the file's normal home is no longer available and do something intelligent. It could use a wireless network and VPN to log on to the server remotely. Or it could just save any changes you make locally, synchronizing with the version on the server when you return to the office from lunch. In any case, it is normal behavior, not an error, and you shouldn't have to tell the application what it should do every time it encounters this situation.

Almost all error messages can be eliminated. If you take the correct point of view—that error messages *must* be eliminated and that your app's design is subject to change in search of this objective—you will be surprised by how little really needs to be changed to achieve this. In those rare cases where the rest of the application must be altered dramatically, that is the time to compromise with the real world and go ahead and use an error message. But you need to start thinking of this compromise as an admission of failure—as a solution of last resort.

This said, there are always a few critical situations where users must be notified in an obtrusive, attention-demanding manner. For example, suppose that, during market hours, an investment manager sets up some trades to be executed by the end of the day, but she sends them to the trading desk after market close. She should be interrupted from whatever else she's working on to be warned that the trades can't be executed until the market opens tomorrow. At that point she may no longer want to make the trades.

Improving error messages: the last resort

When it is truly infeasible to redesign your application to eliminate the need for error dialogs, we offer here some ways to improve the quality of error messages. Use these recommendations only as a last resort, when you run out of other reasonable options for actually eliminating the error.

An error dialog should be polite, illuminating, and helpful. Never forget that an error dialog is the application's way of reporting on its failure to do *its* job, and that it interrupts the user to do this. The error dialog must be unfailingly polite. It must never even hint that the user caused this problem, because that is simply not true from the user's perspective.

The error dialog must illuminate the problem for the user. This means that it must give him the information he needs to make an appropriate plan to solve the application's problem. It needs to make clear the scope of the problem, what the alternatives are, what the application will do as a default, and what information was lost, if any.

It is wrong for the application to dump the problem in the user's lap and wash its hands of the matter. It should offer to implement at least one suggested solution right there within the error message. It should offer buttons that will take care of the problem in various ways. If a printer is missing, the message should offer options for deferring the printout or selecting another printer. If the database is hopelessly trashed and useless, the application should offer to rebuild it to a working state, including telling the user how long that process will take and what side effects it will cause.

Figure 21-35 shows an example of a reasonable error message. Notice that it is polite, illuminating, and helpful. It doesn't suggest that the user's behavior is anything but impeccable.
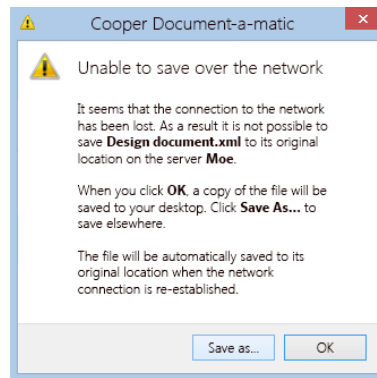
Figure 21-35: If you must use an error dialog, it should look something like this. It politely and clearly illuminates the problem and proposes a good solution. The action buttons and resulting effects are also clearly described.

Alerts and confirmations

Like error dialogs, alerts and confirmations stop the proceedings, often with idiocy. Alerts and confirmations do not report malfunctions. An alert notifies the user of the application's action, whereas a confirmation also gives the user the authority to override that action. These dialogs pop up like weeds in most applications. They should, like error dialogs, be eliminated in favor of more useful idioms, such as those discussed in Chapter 15.

Alerts: announcing the obvious

Alerts usually violate one of the basic design principles from Chapter 18: A dialog is another room, and you should have a good reason to go there. Even if the user must be informed about an action taken by the application, why go into another room to do it?

When it comes down to it, an application should either have the courage of its convictions or should not take action without the user's direct instruction. For example, if the application saves the user's file to disk automatically, it should have the confidence to know that it is doing the right thing. It should provide a means for users to find out what it did, but it doesn't have to stop the proceedings to do so. If the application is unsure whether it should save the file, it shouldn't do so but should leave that operation up to the user.

Conversely, if the user directs the application to do something—dragging a file to the trash can, for example—it doesn't need to stop the proceedings with idiocy to announce that the user just dragged a file to the trash can. The application should ensure that there is adequate visual feedback regarding the action. If the user has made the gesture in error, the application should unobtrusively offer him a robust Undo facility so that he can backtrack.

The rationale for alerts is to keep users informed. This is a great objective, but it need not come at the expense of smooth interaction flow. The alert shown in Figure 21-36 is an example of how alerts are more trouble than help. The Find dialog (the one underneath) already forces the user to click Cancel when the search is completed, but the superimposed alert box adds another flow-breaking button. To return to his work, the user first must click the OK button in the alert and then the Cancel button in the Find dialog. If the information provided by the alert were built into the main Find dialog, the user's burden would be reduced by half.
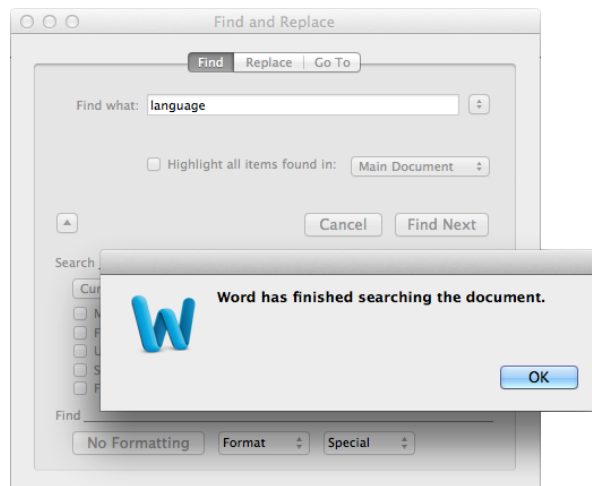
Figure 21-36: A typical alert dialog. It is unnecessary and inappropriate and stops the proceedings with idiocy. Word has finished searching the document. Should reporting that fact be a different facility than the search mechanism itself? If not, why does it use a different dialog?

How to eliminate alerts

Alerts are so numerous because they are so easy to create. Most programming languages offer some form of message facility in a single line of code. Conversely, building an animated status display into the face of an application might require a thousand or more lines of code. Developers cannot be expected to make the right choice in this situation. They have a conflict of interest, so designers must be sure to specify precisely where information is reported on the surface of an application. The designers must then follow up to be sure that the design wasn't compromised for the sake of rapid coding. Imagine if the contractor on a building site decided unilaterally not to add a bathroom because it was just too much trouble to deal with the plumbing. There would be consequences.

Of course, software must keep users informed of its actions. It should have visual indicators built into its main screens to make such status information immediately available to users, should they desire it. Launching an alert to announce an unrequested action is bad enough. Launching one to announce a requested action is pathological.

Software should be flexible and forgiving, but it doesn't need to be fawning and obsequious. The dialog shown in Figure 21-37 is a classic example of an alert that should be put out of its misery. It announces that the application successfully completed a synchronization—its sole reason for existence. This occurs a few seconds after we told it to synchronize. It stops the proceedings to announce the obvious. It's as though the application wants approval for how hard it worked. If a person interacted with us like this, we'd be uncomfortable and find him overbearing. Of course, some feedback is appropriate, but is another dialog that must be dismissed really necessary?
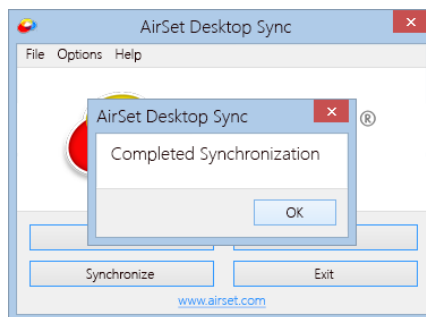
Figure 21-37: This dialog, from AirSet Desktop Sync, is unnecessarily obsequious. We tell it to synchronize and are promptly stopped in our tracks by this important message. Do we really need the application to waste our time demanding recognition that it managed to do its job?

Confirmations: the dialog that cried wolf

When an application feels unconfident about its actions, it often asks the user for approval with a dialog, like the one shown in Figure 21-38. This is called a *confirmation*. Sometimes a confirmation is offered because the application second-guesses one of the user's actions. Sometimes the application feels that it is not competent to make a decision it faces, and it uses a confirmation to give the user the choice instead.
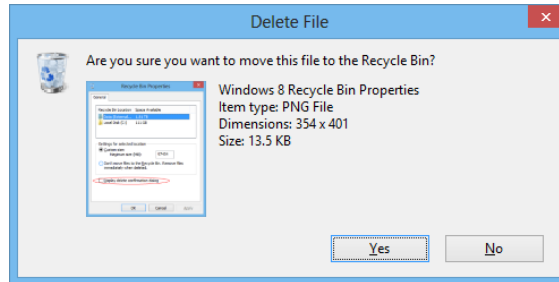
Figure 21-38: Every time we delete a file in Windows, we get this confirmation dialog asking if we're sure. Yes, we're sure. We're always sure. And if we're wrong, we expect Windows to be able to recover the file for us. Windows lives up to that expectation with its Recycle Bin. So, why does it still issue the confirmation message? When a confirmation box is issued routinely, users get used to approving it routinely. So, when it eventually reports an impending disaster to the user, he goes ahead and approves it anyway, because it is routine. Do your users a favor and never create another confirmation dialog.

Confirmations get written into software when a developer arrives at an impasse in her coding. Typically, she realizes that she is about to direct the application to take some bold action, and she feels unsure about taking responsibility for it. Sometimes the bold action is based on some condition the application detects, but more often it is based on a command the user issues. Typically, the confirmation will be launched after the user issues a command that is irrecoverable or whose results might cause undue alarm.

Confirmations pass the buck to users. Users trust the application to do its job, and the application should both do its job and ensure that it does its job right. The proper solution is to make the action easily reversible and provide enough modeless feedback so that users are not caught off-guard.

Confirmations also illustrate an interesting quirk of human behavior: They work only when they are unexpected. That doesn't sound remarkable until you examine it in context. If confirmations are offered in routine places, users quickly become inured to them and routinely dismiss them without a glance. Dismissing confirmations thus becomes as routine as issuing them. If at some point a truly unexpected and dangerous situation

arises—one that should be brought to the user's attention—he will, by rote, dismiss the confirmation, exactly because it has become routine. Like the fable of the boy who cried wolf, the confirmation box won't work if it cries too many times when there is no danger.

For confirmation dialogs to work, they must appear only when the user will almost definitely click the No or Cancel button. They should never appear when the user is likely to click the Yes or OK button. Seen from this perspective, they look rather pointless, don't they?

How to eliminate confirmations

Three design principles provide a way to eliminate confirmation dialogs. The best way is to obey this simple dictum: Do, don't ask. When you design your software, go ahead and give it the force of its convictions (backed up, of course, by user research, as discussed in Chapter 2). Users will respect its brevity and confidence.

> **DESIGN PRINCIPLE**   *Do; don't ask.*

Of course, if an application confidently does something that the user doesn't like, it must be able to reverse the operation. Every aspect of the application's action must be undoable. Instead of asking in advance with a confirmation dialog, on those rare occasions when the application's actions were out of turn, let the user issue the Stop-and-Undo command.

Most situations that we currently consider unprotectable by Undo actually can be protected fairly well. Deleting or overwriting a file is a good example. The file can be moved to a directory where it is kept for a month or so before it is physically deleted. The Windows Recycle Bin uses this strategy, except for the part about automatically erasing files after a month: Users still have to take out the garbage.

> **DESIGN PRINCIPLE**   *Make all actions reversible.*

Even better than acting in haste and forcing users to rescue the application with Undo, you can make sure that applications offer users adequate information so that they never issue a command (or omit a command) that leads to an undesirable result. Applications should use rich visual feedback so that users are constantly kept informed, the same way that dashboard instruments keep us informed of the state of our car.

Occasionally, a situation arises that really can't be protected by Undo. Is this a legitimate case for a confirmation dialog? Not necessarily. A better approach is to provide users with protection the way we give them protection on the freeway: with consistent and clear markings. You can often build excellent, modeless warnings right into the interface. For instance, look at the dialog from Adobe Photoshop shown in Figure 21-39, telling us that our document is larger than the available print area. Why has the application waited until now to inform us of this fact? What if guides showing the actual printable region were visible on the page at all times (unless the user hid them)? What if the parts of the picture outside the printable area were highlighted when the user moved the cursor over the Print button in the toolbar? Clear, rich modeless feedback (as discussed in Chapter 15) is the best way to address these problems.
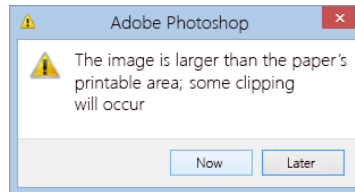
Figure 21-39: This dialog provides too little help too late. What if the application could display the printable region right in the main interface as dotted guides? There's no reason for users to be subjected to dialogs like these.

> **DESIGN PRINCIPLE**
>
> *Provide modeless feedback to help users avoid mistakes.*

Much more common than honestly irreversible actions are actions that are easily reversible but still uselessly protected by routine confirmation boxes. The confirmation shown in Figure 21-38 is an excellent specimen of this species. There is no reason to ask for confirmation of a move to the Recycle Bin. The sole reason the Recycle Bin exists is to implement an Undo facility for deleted files.

The Devil Is in the Details

Although the big-picture principles discussed throughout this book can provide enormous leverage in creating products that will please and satisfy users, it's always important to remember that the devil is in the details.

Frustrating controls and misplaced dialogs can lead to constant low-level annoyance, even if the overall product concept is excellent. Be sure to dot your i's and cross your t's, and ensure that the detailed interactions of your product support your user in his goals, tasks, and aspirations.

If you stick to the concepts behind Goal-Directed Design and use that thinking throughout your framework down to the most minute design details, you will create products that will surpass the competition, make devoted fans of your users, and—perhaps—make the world a better place, one pixel at a time.

DESIGN PRINCIPLES

Chapter 1

- User interfaces should be based on user mental models rather than implementation models.
- Goal-directed interactions reflect user mental models.
- Interaction design is not guesswork.

Chapter 3

- Don't make the user feel stupid.
- Focus the design for each interface on a single primary persona.

Chapter 4

- Define *what* the product will do before you design *how* the product will do it.
- In the early stages of design, pretend the interface is magic.

Chapter 5

- Never show a design approach you're unhappy with; stakeholders just might like it.
- There is only one user experience: Form and behavior must be designed in concert.

Chapter 8

- The computer does the work, and the person does the thinking.
- Software should behave like a considerate human being.
- If it's worth it to the user to do it, it's worth it to the application to remember it.

Chapter 9

- Decisions about technical platform are best made in concert with interaction design efforts.
- Optimize sovereign applications for full-screen use.
- Sovereign interfaces should feature a conservative visual style.
- Sovereign applications should exploit rich input.
- Maximize document views within sovereign applications.
- Transient applications must be simple, clear, and to the point.
- Transient applications should be limited to a single window and view.
- A transient application should launch to its previous position and configuration.
- Kiosks should be optimized for first-time use.

Chapter 10

- Don't weld on training wheels.
- Nobody wants to remain a beginner.
- Optimize for intermediates.
- Inflect the interface for typical navigation.
- Users make commensurate effort if the rewards justify it.
- Imagine users as very intelligent and very busy.

Chapter 11

- No matter how cool your interface is, less of it would be better.
- Don't use dialogs to report normalcy.
- Ask forgiveness, not permission.

Chapter 12

- Eliminate excise wherever possible.
- Don't stop the proceedings with idiocy.
- Don't make users ask for permission.
- Allow input wherever you have output.
- Significant change must be significantly better.

Chapter 13

- Most people would rather be successful than knowledgeable.
- Never bend your interface to fit a metaphor.
- All idioms must be learned; good idioms need to be learned only once.
- Rich visual feedback is the key to successful direct manipulation.
- Visually communicate pliancy whenever possible.

Chapter 14

- An error may not be your application's fault, but it is your application's responsibility.
- Audit, don't edit.
- Save documents and settings automatically.
- Put files where users can find them.

Chapter 17

- Visually distinguish elements that behave differently.
- Visually communicate function and behavior.
- Take things away until the design breaks, and then put that last thing back in.
- Visually show what; textually tell which.

- Obey standards unless there is a truly superior alternative.
- Consistency doesn't imply rigidity.

Chapter 18

- The utility of any interaction idiom is context-dependent.
- A dialog box is another room; have a good reason to go there.
- Provide functions in the window where they are used.
- Use menus to provide a pedagogic vector.
- Disable menu items when they are not applicable.
- Use consistent visual symbols on related commands.
- Toolbars give experienced users fast access to frequently used functions.
- Use ToolTips with all toolbar and iconic controls.
- Support both mouse and keyboard use for navigation and selection tasks.
- Use cursor hinting to show the meanings of metakeys.
- Single-clicking selects data or an object or changes the control state.
- Double-clicking means single-clicking plus action.
- Mouse-down over an object or data should select the object or data.
- Mouse-down over controls means proposing an action; mouse-up means committing to an action.
- The selection state should be visually evident and unambiguous.
- Drop candidates must visually indicate their receptivity.
- The drag cursor must visually identify the source object.
- Any scrollable drag-and-drop target must auto-scroll.
- Debounce all drags.
- Any program that demands precise alignment must offer a vernier.

Chapter 19

- Most mobile apps have transient posture.
- Limit the number and direction of animated screen transitions.
- Use guided tours to orient first-time users.
- Use overlays to explain gestures.

Chapter 20

- Use persistent headers to maintain context.
- Breadcrumbs with lateral links help speed navigation.
- Auto-complete, auto-suggest, and faceted search help users find things faster.
- Make scrolling an engaging experience.
- Infinite scrolling and site footers are mutually exclusive idioms.
- If you have only one version of your site, make it responsive.

Chapter 21

- Use links for navigation and buttons for action.
- Distinguish important text items in lists with graphic icons.
- Avoid scrolling text horizontally.
- Use bounded controls for bounded input.
- Use noneditable (display) controls for output-only text.
- Put primary interactions in the primary window.
- Dialogs are appropriate for functions that are out of the main interaction flow.
- Dialogs are appropriate for organizing controls and information about a single domain object or application function.
- Use verbs in function dialog title bars.
- Use object names in property dialog title bars.
- Differentiate modeless dialogs from modal dialogs.
- Do not use terminating button commands for modeless dialogs.
- Don't dynamically change the labels of terminating buttons.
- Inform the user when the application is unresponsive.
- Never use transitory dialogs as error messages, alerts, or confirmations.
- All interaction idioms have practical limits.
- Don't stack tabs.
- Most error dialogs stop the proceedings with idiocy.
- Make errors impossible.
- Users get humiliated when software tells them they failed.
- Do; don't ask.
- Make all actions reversible.
- Provide modeless feedback to help users avoid mistakes.

BIBLIOGRAPHY

Adlin, Tamara and Pruitt, John. 2010. *The Essential Persona Lifecycle*. New York: Morgan Kaufmann.

Alexander, Christopher. 1964. *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press.

Alexander, Christopher. 1979. *The Timeless Way of Building*. New York: Oxford University Press.

Alexander, Christopher, et al. 1977. *A Pattern Language*. New York: Oxford University Press.

Bertin, Jacques. 2010. *Semiology of Graphics*. Redlands, CA: Ersi Press.

Beyer, Hugh and Holtzblatt, Karen. 1998. *Contextual Design*. New York: Morgan Kaufmann.

Borchers, Jan. 2001. *A Pattern Approach to Interaction Design*. Hoboken, NJ: John Wiley & Sons.

Borenstein, Nathaniel S. 1994. *Programming as if People Mattered*. Princeton: Princeton University Press.

Buxton, Bill. 1990. "The 'Natural' Language of Interaction: A Perspective on Non-Verbal Dialogues." Laurel, Brenda, ed. *The Art of Human-Computer Interface Design*. Boston: Addison-Wesley.

Carroll, John M., ed. 1995. *Scenario-Based Design*. Hoboken, NJ: John Wiley & Sons.

Carroll, John M. 2000. *Making Use: Scenario-Based Design of Human-Computer Interactions*. Cambridge, MA: The MIT Press.

Constantine, Larry L. and Lockwood, Lucy A. D. 1999. *Software for Use*. Boston: Addison-Wesley.

Constantine, Larry L. and Lockwood, Lucy A. D. 2002. *forUse Newsletter* #26, October.

Cooper, Alan. 1999. *The Inmates Are Running the Asylum*. Indianapolis: Sams.

Crampton Smith, Gillian and Tabor, Philip. 1996. "The Role of the Artist-Designer." Winograd, Terry, ed. *Bringing Design to Software*. Boston: Addison-Wesley.

Csikszentmihalyi, Mihaly. 1990. *Flow: The Psychology of Optimal Experience*. New York: Harper & Row.

De Saint-Exupéry, Antoine. 2002.

DeMarco, Tom and Lister, Timothy. 2013. *Peopleware*, Third Edition. Boston: Addison-Wesley.

Dillon, Andrew. "Beyond Usability: Process, Outcome and Affect in Human Computer Interaction." Paper presented at the Lazerow Lecture at the Faculty of Information Studies, University of Toronto, March 2001. Retrieved from `www.ischool.utexas.edu/~adillon/publications/beyond_usability.html`.

Dreyfuss, Henry. 2003. *Designing for People*. New York: Allworth Press.

Gamma, Erich, et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley.

Garrett, Jesse James. 2011. *The Elements of User Experience*, Second Edition. San Francisco: New Riders.

Gellerman, Saul W. 1963. *Motivation and Productivity*. New York: Amacom Press.

Goodman, Elizabeth, Kuniavsky, Mike, and Moed, Andrea. 2012. *Observing the User Experience*. New York: Morgan Kaufmann.

Goodwin, Kim. 2001. "Perfecting Your Personas." *Cooper Newsletter*, July/August.

Goodwin, Kim. 2002. "Getting from Research to Personas: Harnessing the Power of Data." User Interface 7 West Conference.

Goodwin, Kim. 2002a. Cooper U Interaction Design Practicum Notes. Cooper.

Goodwin, Kim. 2009. *Designing for the Digital Age*. Hoboken, NJ: John Wiley & Sons.

Grudin, J. and Pruitt, J. 2002. "Personas, Participatory Design and Product Development: An Infrastructure for Engagement." *PDC '02: Proceedings of the Participatory Design Conference*.

Heckel, Paul. 1994. *The Elements of Friendly Software Design*. San Francisco: Sybex.

Hoober, Steven and Berkman, Eric. 2012. *Designing Mobile Interfaces*. Sebastopol, CA: O'Reilly.

Horn, Robert E. 1998. *Visual Language.* Bainbridge Island, WA: Macro Vu Press.

Horton, William. 1994. *The Icon Book: Visual Symbols for Computer Systems and Documentation.* Hoboken, NJ: John Wiley & Sons.

Johnson, Jeff. 2007. *GUI Bloopers 2.0.* New York: Morgan Kaufmann.

Jones, Matt and Marsden, Gary. 2006. *Mobile Interaction Design.* Hoboken, NJ: John Wiley & Sons.

Kahneman, Daniel. 2011. Thinking, Fast and Slow, NY: Farrar, Straus and Giroux.

Kobara, Shiz. 1991. *Visual Design with OSF/Motif.* Boston: Addison-Wesley.

Korman, Jonathan. 2001. "Putting People Together to Create Good Products." *Cooper Newsletter,* September.

Kramer, Kem-Laurin. 2012. *User Experience in the Age of Sustainability.* New York: Morgan Kaufmann.

Krug, Steve. 2014. *Don't Make Me Think, Revisited.* San Francisco: New Riders.

Kuutti, Kari. 1995. "Work Processes: Scenarios as a Preliminary Vocabulary." Carroll, John M., ed. *Scenario-Based Design.* Hoboken, NJ: John Wiley & Sons.

Laurel, Brenda. 2013. *Computers as Theatre,* Second Edition. Boston: Addison-Wesley.

Lidwell, William, Holden, Kritina, and Butler, Jill. 2010. *Universal Principles of Design,* Revised and Updated Edition. Boston: Rockport Publishers.

Macdonald, Nico. 2003. *What Is Web Design?* Brighton, UK: RotoVision.

McCloud, Scott. 1994. *Understanding Comics.* Northampton, MA: Kitchen Sink Press.

Mikkelson, N. and Lee, W. O. 2000. "Incorporating user archetypes into scenario-based design." *Proceedings of UPA 2000.*

Miller, R. B. 1968. "Response time in man-computer conversational transactions." *Proceedings of the AFIPS Fall Joint Computer Conference,* vol. 33, 267–277.

Mitchell, J. and Shneiderman, B. 1989. "Dynamic versus static menus: An exploratory comparison." *SIGCHI Bulletin,* vol. 20, no. 4, 33–37.

Moggridge, Bill. 2007. *Designing Interactions.* Cambridge, MA: The MIT Press.

Morville, Peter. 2005. *Ambient Findability.* Sebastopol, CA: O'Reilly.

Morville, Peter and Rosenfeld, Louis. 2007. *Information Architecture for the World Wide Web*, Third Edition. Sebastopol, CA: O'Reilly.

Mulder, Steve and Yaar, Ziv. 2006. *The User Is Always Right*. San Francisco: New Riders.

Mullet, Kevin and Sano, Darrell. 1995. *Designing Visual Interfaces*. Upper Saddle River, NJ: Prentice Hall.

Neil, Theresa. 2014. *Mobile Design Pattern Gallery*, Second Edition. Sebastopol, CA: O'Reilly.

Nelson, Theodor Holm. 1990. "The Right Way to Think About Software Design." Laurel, Brenda, ed. *The Art of Human-Computer Interface Design*. Boston: Addison-Wesley.

Newman, William M. and Lamming, Michael G. 1995. *Interactive System Design*. Boston: Addison-Wesley.

Nielsen, Jakob. 1993. *Usability Engineering*. Waltham, MA: Academic Press.

Nielsen, Jakob. 2000. *Designing Web Usability*. San Francisco: New Riders.

Nielsen, Jakob. 2002. `UseIt.com`.

Norman, Don. 2013. *The Design of Everyday Things*, Revised and Expanded Edition. New York: Basic Books.

Norman, Donald A. 1994. *Things That Make Us Smart*. New York: Basic Books.

Norman, Donald A. 1998. *The Invisible Computer*. Cambridge, MA: The MIT Press.

Norman, Donald A. 2005. *Emotional Design*. New York: Basic Books.

Nudelman, Greg. 2013. *Android Design Patterns*. Hoboken, NJ: John Wiley & Sons.

Papanek, Victor. 1984. *Design for the Real World*. Chicago: Academy Chicago Publishers.

Perfetti, Christine and Landesman, Lori. 2001. "The Truth About Download Times." `UIE.com`.

Pinker, Stephen. 1999. *How the Mind Works*. New York: Norton.

Raskin, Jeff. 2000. *The Humane Interface*. Boston: Addison-Wesley.

Reimann, Robert. 2002. "Perspectives: Learning Curves." *edesign* magazine, December.

Reimann, Robert. 2005. "Personas, Scenarios, and Emotional Design." `UXMatters.com`.

Reimann, Robert M. 2001. "So You Want to Be an Interaction Designer." *Cooper Newsletter*, June.

Reimann, Robert M. 2002. "Bridging the Gap from Research to Design." Panel presentation, IBM Make IT Easy Conference.

Reimann, Robert M. and Forlizzi, Jodi. 2001. "Role: Interaction Designer." Presentation to AIGA Experience Design 2001.

Rheinfrank, John and Evenson, Shelley. 1996. "Design Languages." Winograd, Terry, ed. *Bringing Design to Software*. Boston: Addison-Wesley.

Rogers, Yvonne, Sharp, Helen, and Preece, Jenny. 2011. *Interaction Design*, Third Edition. Hoboken, NJ: John Wiley & Sons.

Rombauer, Irma S. and Becker, Marion Rombauer. 1975. *The Joy of Cooking*. New York: Scribner.

Rudolf, Frank. 1998. "Model-Based User Interface Design: Successive Transformations of a Task/Object Model." Wood, Larry E., ed. *User Interface Design: Bridging the Gap from User Requirements to Design*. Boca Raton, FL: CRC Press.

Saffer, Dan. 2010. *Designing for Interaction*, Second Edition. San Francisco: New Riders.

Saffer, Dan. 2013. *Microinteractions*. Sebastopol, CA: O'Reilly.

Sauro, Jeff and Lewis, James R. 2012. *Quantifying the User Experience*. New York: Morgan Kaufmann.

Schön, D. and Bennett, J. 1996. "Reflective Conversation with Materials." Winograd, Terry, ed. *Bringing Design to Software*. Boston: Addison-Wesley.

Schumann, J., Strothotte, T., Raab, A., and Laser, S. 1996. *Assessing the Effect of Non-Photorealistic Rendered Images in CAD*, CHI 1996 Papers, 35–41.

Scott, Bill and Neil, Theresa. 2009. *Designing Web Interfaces*. Sebastopol, CA: O'Reilly.

Shneiderman, Ben, et al. 2009. *Designing the User Interface*, Fifth Edition. Upper Saddle River, NJ: Prentice Hall.

Simon, Herbert A. 1996. *The Sciences of the Artificial*, Third Edition. Cambridge, MA: The MIT Press.

Snyder, Carolyn. 2003. *Paper Prototyping*. New York: Morgan Kaufmann.

Sutton, Robert and Rao, Huggy. 2014. Scaling Up Excellence, BY: Random House.

Tidwell, Jennifer. 2011. *Designing Interfaces*. Sebastopol, CA: O'Reilly.

Tufte, Edward. 1983. *The Visual Display of Quantitative Information*. Cheshire, CT: Graphic Press.

Van Duyne, Douglas K., Landay, James A., and Hong, Jason I. 2006. *The Design of Sites*, Second Edition. Upper Saddle River, NJ: Prentice Hall.

Veen, Jeffrey. 2000. *The Art and Science of Web Design*. San Francisco: New Riders.

Verplank, B., Fulton, J., Black, A., and Moggridge, B. 1993. "Observation and Invention: Use of Scenarios in Interaction Design." Tutorial Notes, InterCHI '93, Amsterdam.

Vora, Pawan. 2009. *Web Application Design Patterns*. New York: Morgan Kaufmann.

Weiss, Michael J. 2000. *The Clustered World: How We Live, What We Buy, and What It All Means About Who We Are*. New York: Little, Brown and Company.

Wigdor, Daniel and Wixon, Dennis. 2011. *Brave NUI World*. New York: Morgan Kaufmann.

Winograd, Terry, ed. 1996. *Bringing Design to Software*. Boston: Addison-Wesley.

Wirfs-Brock, Rebecca. 1993. "Designing Scenarios: Making the Case for a Use Case Framework." *Smalltalk Report*, November/December.

Wixon, Dennis and Ramey, Judith, eds. 1996. *Field Methods Casebook for Software Design*. Hoboken, NJ: John Wiley & Sons.

Wood, Larry E. 1996. "The Ethnographic Interview in User-Centered Task/Work Analysis."

Young, Indi. 2008. *Mental Models*. Brooklyn, NY: Rosenfeld Media.

http://www.smashingmagazine.com/2011/03/07/lean-ux-getting-out-of-the-deliverables-business/

http://dontpaniconline.com/magazine/music/steve-albinis-letter-to-nirvana-before-he-produced-in-utero

INDEX

C

harmonious interactions, 251
  application status, 259–261
  blank slates and, 261–262
  choices, 255
  configuration, 262–264
  contextualizing information, 258–259
  direct manipulation, 255
  feedback, modeless, 256–257
  object status, 259–261
  optimization, 265–266
  persistent objects, 264–265
  reporting and, 261
  tools, 255–256
  user direction, 254–255
  user mental models, 251–252
headers, scrolling and, 583
help. *See also* learnability and help
help screens, mobile applications, 549–550
hierarchies
  excise and, 293–294
  interaction framework, 125–126
hints, 389–390, 618
Holtzblatt, Karen, 70, 111
  Contextual Design, 44, 98
Horton, William, *The Icon Book*, 419
hotspot, 476
HSV (hue, saturation, value), 408–409
hue (color), 408
hyperlinks, 592–593

I

icon buttons, 591–592
icons
  buttons, 456–457
  hamburger, 574
  menus, 451–452
ideas
  generation, 147
  synthesis, 147
identity retrieval, 348
idiomatic interfaces, 300
  branding and, 310
  graphical interfaces, 308–309
  idiom building, 310–312
  learning idioms, 309
idiosyncratically modal behavior, 396, 397–398
immediate commands, 380–381

imperative controls
  buttons, 590–591
  hyperlinks, 592–593
  icon buttons, 591–592
implementation models, 17
  persona expectations and, 112
implementation-centric interfaces, 300–301
improving intermediates, 239
in-app user guides, 394–395
index panes, 438
  table applications, 513–514
indexed searches, 393–394
industrial design framework, 120, 134
  collaboration with designers, 135
  form factors, 135
  form language studies, 136
  input methods, 135
  prototypes, 135
  Synthesis-Generation model, 155
infinite scrolling, 584
inflection, 240–243
information, navigational excise, 277–279
Information Architecture for the World Wide Web
  (Morville & Rosenfeld), 571
information hierarchy, visual interface design,
  410–411
input
  embedded systems design, 559–560
  hints, 389–390
  kiosks, 563–564
  methods, 27
    industrial design framework, 135
    interaction framework, 122
  sovereign-posture applications, 211–212
  text edit controls, 620
insertion, 481–482
insertion target, drag and drop, 487
integration mobile applications, 553–555
interaction design, 11
  behavioral principles, 173–174
  conceptual principles, 173
  interface-level principles, 173–174
  patterns, 27, 174
    architectural, 175
    behavioral, 176
    double drawer sample, 177
    organizer-workspace sample, 177
    postural, 176
    recording, 175–176

Krug, Steve, *Don't Make Me Think, Revisited*, 571

Kuniavsky, Mike, *Observing the User Experience*, 56, 141

J

Jobs, Steve, 435–436

journeys, 136–137

K

key path scenarios, 28, 106
> interaction frameworks, 128

keyboard access, 401

kickoff meeting, research and, 37–38

kiosks, 561
> explorational, 561–562
> input, 563–564
> interaction in public environment, 562–563
> kiosk posture, 230–231
> transactional, 561–562

L

Laurel, Brenda, 304
> *Computers as Theatre*, 103

layout, handhelds, 511–512

learnability and help, 379–380
> assistive interfaces, 385–393
> command modalities, 379–380
>> immediate commands, 380–381
>> information in the world *versus* in the head, 381–382
>> invisible commands, 380–382
>> memorization vectors, 382–384
>> pedagogic commands, 380–382
> contextual help, 385–393
>> assistive interfaces, 385–393
> online help
>> full-text search, 393–394
>> in-app user guides, 394–395
>> indexing, 393–394
>> overviews, 394
> working sets, 384

level of experience, designing for, 243–247

levers, 624–625

life goals, 77–78

Lisa computer, 435–436

list controls, 601–603
> combo boxes, 608–609
> dragging and dropping from, 604–605

earmarking, 603–604
entering data, 607–608
mobile applications, 518–519
ordering, 605–606
scrolling, 606–607
tree controls, 609–610

Lister, Timothy, *Peopleware: Productive Projects and Teams*, 249

literature review, 38

localization, 398–399

logical mapping, 291

logical path, 418

M

Macintosh, 435–436

magic interface, 115

Making Use: Scenario-Based Design of Human-Computer Interactions (Carroll), 104–105

manual affordances, 312–314

mapping
> excise elimination and, 290–293
> logical, 291
> physical, 291

market research, user research and, 35–36

market segments
> *versus* personas, 71–72
> personas and, 95

marketing, teams and, 157

master-apprentice learning model, 44

maximized windows, 442

MDI (multiple document interface), 443–444

mechanical age models, 294–297

memorization vectors, 382–384

memory work, 271

mental models, 17–18
> harmonious interactions and, 251–252
> persona expectations and, 112
> Undo and, 364–366

menu bar, 448
> mobile applications, 535

menus
> accelerators, 452–453
> access keys, 453
> cascading, 453–454
> check mark items, 451
> disabled items, 450–451
> drop-down, 448
> grayed out items, 450–451

attitudes, 83
behavior, ranges of, 68
behavioral variables, 83
completeness in, 87
composite user archetypes, 67
customer personas, 69, 89
design pitfalls and, 64–66
as design tool, 64
edge cases and, 65–66
effectiveness, 66
elastic users and, 65
expectations, 112
goals, defining, 85–86
implementation models and, 112
interview subjects
    behavior patterns, 84
    grouping by role, 82–83
    mapping to behavioral variables, 83–84
market segments, 95
versus market segments, 71–72
mental models and, 112
misconceptions, 93–94
motivations, 68–69, 83
narratives, 90–91
negative, 90
nonusers, 69
organizational, 96
patterns of behavior, 62, 67
persona set, 68
photos, 91–92
primary, 88–89
products
    multiple, 67
    specific, 67
provisional, 97
quantifying, 95–96
versus real people, 94
redundancy in, 87
represented models and, 112
research and, 66–67
secondary, 89
self-referential design and, 65
served personas, 69, 89
skills, 83
social relationships and, 86–87
strengths, 62–64
supplemental, 89
tasks and, 94–95
traceability, 95

versus user profiles, 71
versus user roles, 69–71
validating, 95
photos, personas and, 90–91
physical harm, ethical interaction design and, 169
physical mapping, 291
physical models, 99
physical work, 271
picking problem with 3D objects, 506
pinch in/out, 552
placement in file system, 340
planning, product behavior, 10–13
platforms, 205–206
    versus posture, 205
pliancy, mouse, 470
pluralized windows, 443
pointing devices, 465
    gesture sensors, 475
    hotspot, 476
    touchscreens and, 466
    trackballs, 475
    trackpads, 475
poles, 3D objects, 503
polylines, 499–500
poor product behavior, 4–6
pop-up control panels, tablet applications, 514
pop-up menus, 448
pop-up toolbars, 461
position, visual interface design, 409
positional retrieval, 348
postural patterns (interaction design), 176
postures, 206–207
    automotive interface, 232–233
    desktop, 207–208
        daemonic posture, 217–218
        sovereign posture, 208–213
        transient posture, 213–217
    kiosk, 230–231
    mobile devices, smartphone and handheld
        posture, 225–228
    versus platform, 205
    smart appliance, 234–235
    tablet device, 228–230
    ten-foot, 231–232
    Web, 218–219
        information websites, 219–220
        transactional websites, 221–222
        web application, 222–224
pragmatic interaction design, 171

reversing changes, 341

Save As command, 334–336

Save Changes, 333–334

scanability, 581

scenario-based design, 104–105

scenarios, 102–103

    context, 106

        Requirements Definition, 113–115

        sample, 114–115

    goals, 105

    key path, 28, 106

        interaction frameworks, 128

    persona-based, 105–106

    use cases and, 103–104

    user stories and, 103–104

    validation, 28, 106

        alternative scenarios, 130

        edge-case scenarios, 130

        necessary-use scenarios, 130

schedule, research and, 40

Schön, Donald, 39

scope, embedded systems design, 559

screen

    navigational excise, 274

    sovereign-posture applications, 209–210

scrollbars, 621–623

scrolling

    drag and drop

        auto-scrolling, 488–489

        fine scrolling, 488–489

    infinite, 519

    Internet applications, 580

        footers, 583

        headers, 583

        infinite scrolling, 584

        scanability, 581

    in lists, 606–607

    mouse, 469

    scanability, 581

    visually impaired users and, 402

SDI (single document interface), 443–444

search

    auto-complete, 578

    auto-suggest, 578

    categorized suggestions, 580

    disambiguation, 578

    faceted, 579

    help, full-text, 393–394

Internet applications, 577–580

    mobile applications

        auto-complete, 544

        auto-suggest, 545

        categorized suggestions, 545

        explicit, 544

        query building, 544–546

        recent/frequent, 545

        tap-ahead, 544

        voice search, 544

secondary personas, 89

secondary windows, 437

secondary-action drawers, 536–537

selecting

    additive, 479

    command ordering and, 476–477

    contiguous, 477–478

    controls, 593–594

        check boxes, 594–595

        combo icon buttons, 599–601

        radio buttons, 596–598

        state-switching buttons, 595–596

        switch controls, 598

        toggle buttons, 595

    discrete, 477–478

    exclusion, mutual, 478–479

    group, 480

    visual indication, 480–481

self-referential design, 65

sequence models, 98

served personas, 69, 89

service blueprint, 136–137

service design framework

    customer journeys, 136

    experience prototypes, 137

    service blueprint, 136–137

service framework, 120

service map, 28

session managers, 440–441

shadows, 3D objects, 503

shape, visual interface design, 407

Shneiderman, Ben, 315

sidebars, 438, 455, 463–464

    modeless dialogs and, 631

signposts, excise and, 286–290

single Undo, 367–368

    limitations, 368

sketching interaction framework, 126–127

widgets, table device posture, 230

WIMP (windows, icons, menus, and pointer), 435

windows

    desktop

        best use, 444–448

        full-screen applications, 441

        maximized, 442

        MDI (multiple document interface), 443–444

        minimized, 442

        multipaned applications, 441–442

        overlapping, 439–440

        pluralized, 443

        restored, 443

        room analogy, 444–448

        SDI (single document interface), 443–444

        session managers, 440–441

        tiles, 440

        virtual desktop spaces, 440–441

    primary, 436–437

        content panes, 438

        index panes, 438

        menus, 437

        sidebars, 438

        tool palettes, 438

        toolbars, 437

    secondary, 437

    states, 442–443

wireframes, 3D objects, 504

wizards, 390–391

work

    cognitive work, 271

    memory work, 271

    physical work, 271

    visual work, 271

work flow models, 98

working sets, 384

XYZ

Xerox Alto, 435

zoom, 3D objects, 506

cooper

DESIGNING SOLUTIONS
FROM PRODUCTS AND SERVICES
TO PROCESS AND PRACTICE

"[Cooper U] is a great investment. Both as a personal career move, and for anyone with a budget, who recognizes the value of UX, but doesn't quite know how to put skills and teams together to create practical, useful, goal-directed product designs."

– PETTERI HIISILÄ, COOPER U INTERACTION DESIGN ALUMNI

"We asked Cooper to blow our socks off, and they delivered in spades. They gave us great research and awesome designs. Our feet are a little cold, though!"

– VP, PRODUCT DESIGN, FORTUNE 500 SOFTWARE COMPANY

"Essential to the future of how creative work is done."

– TODD CRAMER, DESIGN LEADERSHIP ALUMNI

"Cooper helped us reflect on our business, our processes, and our product vision, elevating it in ways we didn't imagine."

– IAN SHAKIL, CEO, AUGMEDIX

"I've been reading textbooks, tweets, blogs on UX for months along with taking extended learning classes. But nothing beats getting your hands down and dirty with the pressures of end-of-day deadlines."

– DANTE GUINTU, UX BOOT CAMP ALUMNI

"Interning at Cooper is an internship in name only, from day one I was put on actual, client-facing projects. This meant that I was not only receiving mentorship from my senior design partner, but also developing the invaluable, and often intangible skills of an effective design consultant."

– BRENDAN KNERAM, FORMER COOPER INTERN

cooper

MASTER YOUR CRAFT

We're a team of passionate leaders, innovators, and craftspeople.

Cooper U Courses

We've taught thousands of people around the world. Our courses don't just teach design techniques, they develop leadership skills so you can build the vision you imagine.

cooper.com/training
cooperu@cooper.com

Our Studio

We have top-tier designers, a world-renowned process, and visionary leaders. Our designers make each other better, while solving design challenges that create resounding impacts for our clients and the world.

cooper.com/careers
careers@cooper.com

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.