The Pragmatic Programmers

Forge Your Future with Open Source

Build Your Skills
Build Your Network
Build the Future
of Technology

VM (Vicky) Brasseur
edited by Brian MacDonald

Early praise for *Forge Your Future with Open Source*

This book has needed to exist for a long time. Newcomers to free and open source software now have a thorough guide to participation, and the author's real-world experience shows on every page. Among the book's strengths is that it not only explains what steps to take (and their variants), but also shows new contributors how things look from the project's point of view. I hope and expect to see this book referred to by contributors to projects across the internet.

➤ **Karl Fogel**
Partner, Open Tech Strategies, LLC

If you ever wished you could have a compendium of HOWTO open source from one of the most knowledgeable folks in this biz, this is the book for you. Vicky is an absolute gem and has successfully distilled decades of knowledge into an easy-to-access format that should be required reading for anyone wanting to get into FOSS.

➤ **Katie McLaughlin**
Director, Django Software Foundation, Python Software Foundation

The next time someone tells me they want to learn more about open source, I'll have the perfect book recommendation. Vicky has written the concise, practical guidebook we were missing. *Forge Your Future with Open Source* is an excellent quick-start guide for anyone stepping into the world of open source.

➤ **Rikki Endsley**
Community Manager, Opensource.com, Red Hat

Wonderfully readable, not only as a practical manual, but as an engaging and inspirational introduction to the world of free software, one practical and people-oriented example at a time. This is the book I wish I had read many years ago.

➤ **Chris Lamb**
Debian Project Leader

I've been working in open source for almost two decades. I went to Microsoft a decade ago to open-source .NET and C#. I wish I'd had a copy of VM's book. This book offers valuable historical context and practical guidelines on how and when to work on an Open Source project. *Forge Your Future with Open Source* will no doubt empower the next generation of contributors and I'm envious of their bright futures!

➤ **Scott Hanselman**
Program Partner Manager, Open Source .NET, Microsoft

Vicky's book is the "Goldilocks" guide to participating in open source: just the right information, with neither too much obscure detail nor too little actionable advice. I look forward to recommending it to others.

➤ **Cat Allman**
Board Member, USENIX

Contributing to a free software project is one of the best ways to help the free software movement, and this book is the comprehensive, self-contained guide you need to get started. Brasseur skillfully balances depth and breadth, homing in on key points around the mechanics of contributing as well as the oft-neglected meta areas of effective communication, licensing, and employment ramifications.

➤ **John Sullivan**
Debian Developer

Forge Your Future with Open Source goes where no book has gone before, clearly teaching how to get started as a contributor to open source, explaining why contributing is valuable and rewarding, and exploring the technical and social challenges both new and experienced contributors face, in an honest and practical way.

➤ **Allison Randal**
Board Member, Open Source Initiative

In her inimitable style, VM Brasseur brings a useful cheat sheet for contributing to free and open source software. There is probably something in this book for everyone to learn.

➤ **Karen Sandler**
Co-Organizer, Outreachy

Vicky unflaggingly reminds us that creating software is a liberal art—and the foundational art of Open Source is courtesy. If every reader were to practice some of the advice in this book, the software world would be a more welcoming place.

➤ **Robert "r0ml" Lefkowitz**
Distinguished Engineer, ACM

Open Source runs most of the technology world, from mobile phones to the internet. Despite it being open, there are many hidden rules in how teams work together. Vicky's glorious book removes the arcane barriers surrounding this field and takes us along a journey into Open Source from the practice, the culture, the community, the history, the motivation, and even how we talk to each other. It is a book built on years of practice that not only needed to be written but deserves to be read by anyone wanting to contribute to this field.

➤ **Simon Wardley**
Researcher, Leading Edge Forum

We've left this page blank to make the page numbers the same in the electronic and paper books.

We tried just leaving it out, but then people wrote us to ask about the missing pages.

Anyway, Eddy the Gerbil wanted to say “hello.”

Forge Your Future with Open Source

Build Your Skills. Build Your Network.
Build the Future of Technology.

VM (Vicky) Brasseur

The Pragmatic Bookshelf
Dallas, Texas

See our complete catalog of hands-on, practical,
and Pragmatic content for software developers:
https://pragprog.com

Sales, volume licensing, and support:
support@pragprog.com

Derivative works, AI training and testing,
international translations, and other rights:
rights@pragprog.com

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Copy Editor: Paula Robertson
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

ISBN-13: 978-1-68050-301-2
Encoded using recycled binary digits.
Book version: P1.0—October 2018

Contents

Preface

Here we are—forty years on from the launch of the Free Software movement and twenty years since the "open source" and its related movement were created—and it's still really hard to contribute to most free and open source software projects. There are all of these unspoken rules, unfamiliar language, and a lack of documentation that would be impressive were it not so unfortunate. The web is full of articles about how to contribute, but none of them cover everything you need to make your first contribution. You end up playing contribution Whac-a-Mole,[1] with a new problem or unspoken rule popping up the moment you think you know what's going on. It's all so frustrating sometimes.

Don't Panic, I'm here to help. Welcome, my friends, to the book that finally makes sense of contributing to open source.

What's in This Book?

In these pages, you'll find everything you need to start contributing to free and open source projects. First, we'll cover the history and philosophies of these movements, since without that knowledge, you'll trip and fall on the very first step of your journey to becoming a contributor. After that, we'll investigate the benefits of contributing to free and open source and help you select a project that suits your needs, so both you and the project can benefit from your contribution. Obviously we'll cover how to make the contribution itself, but we'll also discuss the many ways you can contribute without writing a line of code. All of those unspoken rules will be revealed, and we'll even talk about how to start your own open source project.

What's NOT in This Book?

This book will not hand you a list of free and open source projects where you should start contributing. Not only would such a list be out of date the moment

1. https://en.wikipedia.org/wiki/Whac-A-Mole#Colloquial_usage

it was published, but it also wouldn't be the right list for everyone. Thousands of free and open source software (FOSS) projects exist in the world today. It would be silly to list a few, then send all of you stampeding off to contribute to just those. With so many projects to choose from, you can find a project that matches your specific skills and interests. In fact, there's an entire chapter to help you do just that.

This book also does not recommend which tools to use for contributing. The most effective tool is the one that works for you (as long as the end product meets the requirements of the project).

I've gone to great lengths to try not to influence your choice of project or tools, while giving you the information and support you need to make your own decisions. You do you, honey.

Who Should Read This Book?

From experienced software professional to new student, anyone who wishes to contribute to FOSS will find value in this book. While most people think FOSS contributions are only the realm of programmers, nothing could be further from the truth. Software development is a multidisciplinary undertaking. Writers, testers, designers, project managers, marketers… There's a place for everyone in free and open source software.

While this book contains some technical concepts, it does not assume that you are a programmer, that your contribution will be code, or even that you're overly familiar with software development. Free and open source software needs all sorts of contributions, submitted by all sorts of people.

Why Is This Book Not Openly Licensed?

Yeah, I thought someone might ask that.

With the growing awareness and importance of open source along with the explosion of new projects released every year, it's more important than ever that there be a resource to enable and support the immense number of new contributors we're going to need to help maintain that software. This book is that resource.

While this book needed to exist, and I was well equipped and well placed to write it, it wasn't going to happen if I did it on my own. Without external help or motivation, I know that I would never finish a project this huge. I mean: writing a book? That's really intimidating.

To make this happen, I needed help. Enter Pragmatic Bookshelf. Their experience and support could guide and motivate me to create this book, but only if they administer the copyright of it. This allows them to do things like negotiate translations and similar administrative duties, with which I have no experience whatsoever.

The choices were:

1. I assign copyright of the book to Pragmatic, then have their invaluable support to help bring it into the world, and Pragmatic chooses a book license based on their current business practices.
2. I retain copyright, but I do it alone and unsupported, so the book is never started, finished, nor released under any license at all.

This book is an important resource that is long overdue and will help thousands of people and free and open source software projects. That it finally exists is more important to me than my copyright. So I chose Option 1. I have no regrets. Pragmatic are great people.

Suggestions, Errata, or Questions?

Free and open source software is all about the community coming together to collaborate and build something amazing. This book is no different. Do you have suggestions for how to improve the book for future generations of contributors? Problems with the content? Share your thoughts using the errata submission form on the book's catalog page on the Pragmatic Bookshelf website.[2]

Also, if you have any questions at all about contributing to free and open source software, join us on the #fossforge channel on the Freenode Internet Relay Chat (IRC) network. A web interface[3] is available to make joining easy, as well as a quickstart guide[4] if you're not familiar with IRC. The channel community will gladly help support you in your journey from novice to contributor.

Credits

I mostly wrote this book on a 2016 MacBook Pro, in Markdown, using MacVim as my text editor and git as my version control, though more than a few pages were written on a 9.7" iPad Pro using the Textastic text editor and

2. https://pragprog.com/titles/vbopens/errata
3. https://webchat.freenode.net/?channels=%23fossforge
4. https://opensource.com/life/16/6/irc-quickstart-guide

WorkingCopy git client. The diagrams are my creation, using OmniGraffle. The font in the diagrams is *Open Sans*, created by Steve Matteson and licensed under the Apache 2.0 open source license. The handwriting font used in several examples is *Nothing You Could Do*, created by Kimberly Geswein and licensed under the Open Font License. The Kannadan font used in *(as yet) unwritten content* is *Kedage* by the Indian Language Technology Solutions Project and is licensed under the GNU General Public License version 2.

Acknowledgments

Books don't happen easily, and they don't happen solely through the force of will of their authors. True to the spirit of free and open source software, a lot of people contributed to the creation of this work.

To every free and open source community member and leader who was patient and generous with their guidance and advice over the years and who helped me learn what was necessary for this book: Thank you.

To the Opensource.com community moderators, whose brilliance and insight never fail to inspire me to be a better human and contributor: Thank you.

To the technical reviewers, and particularly to those who gave up part of their 2017 holiday to review the first half of the book: Thank you. The reviewers were (in alphabetical order by first name): Alessandro Bahgat, Andrea Goulet, Ashish Bhatia, Ben Cotton, Daivid Morgan, Derek Graham, Donna Benjamin, Emanuele Origgi, Fabrizio Cucci, Glen Messenger, John Strobel, Johnny Hopkins, Karen Sandler, Karl Fogel, Katie McLaughlin, Máirín Duffy, Maricris Nonato, Mark Goody, Matthew Oldham, Michael Hunter, Mitchell Volk, Nick McGinness, Nouran Mhmoud, Peter Hampton, Raymond Machira, Rikki Endsley, Robin Muilwijk, Scott Ford, Stephen Jacobs, Tibor Simic, and Zulfikar Dharmawan. If any errors or omissions still exist in the book, the fault is entirely mine for ignoring their advice.

To Chethan R Nayak, for providing the Kannadan translation used in *(as yet) unwritten content*: Thank you.

To Sage Sharp, for wisely suggesting I add a section to *Prepare to Contribute* about roles commonly found in FOSS projects: Thank you.

To Ben, John, Katie, and Rikki, for their invaluable counsel during the title selection process: Thank you.

To the Pragmatic team, for believing this book was a good idea and providing a happy and supportive home for it: Thank you.

To Brian, my editor and my friend, who came to me with a crazy idea and who helped me turn it into reality, without whom I *literally* could not have done this (pun intended): Thank you.

To everyone on the channel, who knows who they are and who are there for me through it all: Thank you. I love each and every one of you and I will never tire of saying so.

And finally to you, who will help shape the future of technology through your free and open source contributions: Thank you.

CHAPTER 1

The Foundations and Philosophies of Free and Open Source

When we think or talk about free and open source software, there's a strong tendency to focus on that last bit: the software. Software, as we all know, is just made out of code, right? So isn't free and open source software, therefore, all about the code? It's all programming, but it's programming that anyone can use, like, for FREE, man. After all, that's what the *free* means in "free and open source", right? You can use it, but there's no cost. Yup, that's open source. Book done. We can all move along.

You've probably already guessed that the previous paragraph was a steaming pile of misinformation. Unfortunately, it's based on a lot of the common misconceptions about free and open source software. These fallacies are repeated and perpetuated to the point of being seen as common knowledge. As is often the case with things like this, not everything that is common qualifies as knowledge.

It's About the People

For instance, despite what many believe, free and open source software is not only about the *software*; it's also about the *people*. People build the software, employing varied skills like writing, testing, designing, and (yes) programming. People maintain the software and form tight-knit communities to support both the software and its users. It was through the vital and deep-seated convictions of people that free and open source software exists at all. Those convictions form the basis of a philosophy of freedom and sharing that enabled the world-changing idea, "software should be Free," to evolve into the massive social movement that we today know as open source.

To participate in free and open source software, it's critical that you understand that, while it's tightly entwined with software and technology, it is fundamentally a social movement. Social movements are composed of people, and as we know, people are difficult, squishy, amazing things. Contributing to free and open source is not simply a mechanical process of pushing code from here to there. To contribute, you must understand the underlying social constructs and philosophies that are common to all free and open source software projects.

Why Learn About the Philosophies?

"But!" you interject, "I'm not here to learn about philosophy and stuff. I just want to contribute! Why tell me all this?"

Without these philosophies, there would be no free and open source software. While it may not always be obvious, the freedoms and beliefs of the founders of the free and open source movements underpin everything in each project you use and contribute to. The participants in most free and open source projects are aware of these philosophies and will expect you to be, as well. The few minutes you spend reading this chapter will provide valuable context that allows you to better understand the motivations behind many of the actions you'll see taken in open source projects.

It could be that after learning the basic philosophies, you find you're either drawn to or repelled by them. This is an important realization to have at this point. It will guide you toward those projects that best suit your own beliefs, away from those that don't, or perhaps away from contributing at all. If that's the path you choose: Congratulations! Few people are self aware enough up front to avoid devoting so much of their free time to a pursuit that doesn't appeal to them. The time you spend learning and thinking about these philosophies now can save you days, weeks, months, or more in the future.

Free and Open Source Software Is Everywhere

Free and open source software is everywhere. Your car, TV, and even your light bulbs probably run software using the *Linux* kernel and related operating system. Your phone either runs on an open source platform—*Android*—or it has a proprietary platform but runs apps written in an open source language—*Swift*. The movies you watch may have been created using the *Blender* free and open 3D rendering suite, and they certainly were converted or edited with the help of *ffmpeg*, a free software tool for manipulating digital media files. You may open your open source browser—*Firefox*—to watch a live stream

delivered by the free *Open Broadcaster Software*. You may then place an order from an online merchant, who built their website using the free and open *Wordpress*, *Drupal*, or *Joomla*. Thanks to the *OpenSSL* cryptographic library and tools, you know that your financial information will remain secure.

Free and open source software (*FOSS*) has become the default choice for programming languages, infrastructure, databases, content management systems, and web servers among many other categories of technology. There are millions of free and open source projects, performing billions of different tasks. Every year GitHub, an online service for hosting and developing software and a major supporter of open source, releases a report of the GitHub and open source world. It calls this study *The Octoverse*. The 2017 Octoverse report[1] shows more than 25 million open repositories on GitHub alone, and this number is just a fraction of the open projects available.

As mentioned earlier, free and open source is more than just software: it's people. Each project is built by people for people. People use, contribute to, and support the projects. And people form organizations dedicated to cultivating and advancing the free and open source software movements. These organizations exist all over the world, in nearly every region. In the USA, you can support the Free Software Foundation,[2] the Software Freedom Conservancy,[3] the Open Source Initiative,[4] or Software in the Public Interest,[5] among others. In Europe, you have Free Software Foundation Europe,[6] Open Source Projects EU OSP),[7] and Open Forum Europe (OFE). Australasia is supported by Linux Australia,[8] Opensource.asia,[9] and FOSSAsia.[10] Groups like Free and Open Source Software For Africa (FOSSFA)[11] and OpenAfrica[12] support, teach, and spread free and open source technologies across many countries in Africa. Central and South America are also highly active in the free and open source world, thanks

1. https://octoverse.github.com
2. https://www.fsf.org
3. https://sfconservancy.org
4. https://opensource.org/
5. https://www.spi-inc.org
6. https://fsfe.org/index.en.html
7. https://opensourceprojects.eu
8. https://linux.org.au
9. http://opensource.asia
10. https://fossasia.org
11. http://www.fossfa.net
12. https://africaopendata.org

to groups like Software Livre Brasil,[13] FLISOL,[14] and Grup de Usuarios de Software Libre Perú[15] among dozens of others.

Other Open Movements

The open ethos isn't limited to software. A number of related movements have sprung up in the past few decades, each dedicated to sharing, transparency, and collaboration.

Wikipedia[16] is the most well-known and highly trafficked of these open movements. Anyone in the world is encouraged to contribute to its ever-growing knowledge base. The majority of the content on Wikipedia is available under a license furnished and maintained by Creative Commons.[17] Creative Commons is an organization that promotes the free sharing and reuse of creative works like music, writing, art, and data by providing copyright licenses that can be applied to them. This standard and well-understood body of licenses helps people share their works while still protecting their valuable copyright.

Wikipedia and Creative Commons are far from the only non-software open movements. Open Knowledge International[18] empowers society through open data. Internet Archive[19] aims to provide free and open access to all the world's knowledge. Open access academic journals ensure the free and open flow of fundamental research. The Open Source Seed Initiative[20] maintains open access to plant genetic resources that might otherwise be locked behind patents. These are just a few of the many ways that the free and open ethos has permeated our culture.

This philosophy of open access and sharing goes back thousands of years, but how did it become so prevalent in software?

13. https://softwarelivre.org
14. https://flisol.info
15. https://www.softwarelibre.org.pe
16. https://wikipedia.org
17. https://creativecommons.org
18. https://okfn.org/about/
19. https://archive.org
20. https://osseeds.org

The Origins of Free Software

Before you start to contribute to free and open source software projects or join their communities, you should probably know something about the nature and philosophies of FOSS and how it got where it is today.

In the early days of computers, all software was free to acquire, use, inspect, modify, and share. Researchers, computer operators, and computer hardware manufacturers all gladly distributed their software works to others. At the time, the profits were in the hardware sold, not in the software that ran on it. No one had yet considered that software could be a revenue stream, largely because each model of hardware was highly specialized, such that the software written for one model would not run on another. A single piece of software could not be widely used, so there was no profit from selling it. If the software enabled the sale of more of the highly profitable hardware, then computer manufacturers were thrilled that people would share that software with each other. It was exactly like today– when you might buy a game console because it's the only platform with the game you want to play–but with mainframes instead of gameframes.

All good things come to an end. Eventually manufacturers recognized not only the value the software provided to users but also the amount of effort that went into developing it. Where there's value there's profit, so these companies started software development as its own industry distinct from the creation of the computer hardware on which it ran. As the profits began to roll in for the software developers, some operators—who were used to using and sharing software—started to resent not only the new cost of acquiring software but also that they could no longer modify it for their needs and then share the updated software with others.

In 1983 Richard M. Stallman (RMS),[21] frustrated that software operators were no longer free to inspect, modify, and share software, announced the launch of the GNU Project.[22] This project is dedicated to the creation of a UNIX-compatible operating system built of components that are entirely free to use, modify, and distribute. Two years later, the GNU Manifesto[23] followed. It declared the fundamental beliefs of the project and launched Free Software as a movement.

21. https://en.wikipedia.org/wiki/Richard_Stallman
22. https://en.wikipedia.org/wiki/GNU_Project
23. https://en.wikipedia.org/wiki/GNU_Manifesto

The Four Freedoms are the core of the free software movement. Those freedoms—which in standard programming fashion, are numbered starting from zero—are:

0. The freedom to run the software however you wish and for whatever reason you wish.
1. The freedom to study the software source code and make whatever changes you wish.
2. The freedom to copy and distribute the software (modified or not) however you wish.
3. The freedom to make improvements to the software and then share the improved version however you wish.

Any software that does not guarantee these freedoms to its users cannot be considered "free" because it limits the users' rights in some way. To help guarantee these rights and freedoms, RMS, the GNU Project, and the newly formed Free Software Foundation (FSF)[24] created software licenses that leverage the pre-existing concepts of copyright. The FSF *copyleft* licenses (a play on the word *copyright*) provide more than just the permission to use software released under them; they ensure that software can never violate the Four Freedoms. While many people believe that the Four Freedoms are Stallman's greatest invention, in fact his most far-reaching and brilliant contribution to software is the recognition that copyright can be used in this way, and that careful copyright licensing can enforce software freedom. This invention paved the way for the open source movement that followed.

The Origins of Open Source

The free software movement grew in popularity and awareness throughout the 1980s and 1990s and attracted the attention of business interests. The release of the Netscape web browser as free software in 1998 amplified this attention. While businesses were intrigued by the potential of open software development, many were less thrilled with the strong political, philosophical, and activist nature of the free software movement and its supporters.

In early 1998, soon after the release of the Netscape code, several free software supporters gathered to discuss how the movement might make itself more palatable to business interests, in hopes of increasing the scope, reach, and contributors for open software development. They decided a rebranding was

24. https://www.fsf.org

in order and chose the term *open source*—coined by Christine Peterson[25]—as the name for this version of the movement. Many members of the group then created the Open Source Initiative (OSI)[26] as a focal point for their efforts.

One of the first tasks of OSI was codifying what it means to be an open source software project. The Open Source Definition describes the ten responsibilities and requirements a project must fulfill if it wants to qualify as an "open source" project. The OSI has a detailed description of the definition on its website,[27] but the definition can be summarized as:

1. The project must be freely redistributable—even if sold and even if it's part of a larger collection of software.
2. All source code must be available and distributable.
3. Modifications and derived works must be allowed and distributable under the same license terms.
4. If distribution of modified code isn't allowed, it cannot prevent distribution of patch files (snippets of source code that can be applied to include new fixes or functionality) along with the unmodified code.
5. In no way can the license under which the code is distributed discriminate against any person or group. All people must be allowed to use the code on the same terms, even if they're bad people like Nazis.
6. Similarly, the license also can't single out industries, companies, or other types of undertakings. All groups and ventures must be allowed to use the code on the same terms, even if those groups support horrible things (again, like Nazis).
7. The license applies to anyone who receives a copy of the software without needing any additional permissions.
8. The license can't restrict someone from extracting the project or code from a larger collection. If they do extract it, it's available to them under the same license terms as the whole, larger collection.
9. If the software is distributed as part of a larger collection of code, the license can't put any restrictions or requirements on that other code.
10. The license applies to all technology and UI applications of the software to which it's applied.

The OSI provides an annotated version[28] of the Open Source Definition. This version is valuable for understanding the meaning and importance of the

25. https://en.wikipedia.org/wiki/Christine_Peterson
26. https://opensource.org/
27. https://opensource.org/osd
28. https://opensource.org/osd-annotated

definition. It details rationales and supporting information for each of the criteria for a project to qualify as "open source."

Most of these criteria apply to the *license* under which a project is distributed. To aid people in selecting a license that meets all of the criteria, OSI reviews licenses and maintains a list of OSI-approved open source licenses.[29] If a project claims to be "open source" but is not released under an OSI-approved license, then the project cannot call itself "open source."

This focus on license is a critical part of free and open source software. It's the license and its directives that make a piece of software open source, not merely the availability of the source code. Without the application of an OSI-approved license, code can be at best "source available" but not open. The legal mandates contained in the license ensure that the code is available and that people are free to do with it what they wish (within the constraints of the license). Code and projects that do not have license files, even if they have been bequeathed to the public domain, are not open source.

Difference Between Free Software and Open Source

One question that everyone asks when they first discover FOSS: "What's the difference between Free and Open Source?" This is a surprisingly contentious question, but a very good one to ask. From a code and project perspective, there's very little effective difference between the two. Most of the licenses that the FSF considers "Free" are also OSI-approved, and many of the OSI-approved licenses support the Four Freedoms and therefore, are also considered "Free" by the FSF. There are some outliers on each side, but there is far more similarity than difference between the two families of licenses. In most cases, Free Software is also Open Source. In slightly fewer cases, Open Source is also Free Software.

The difference between Free and Open Source comes down to one of philosophy and motivation. For supporters of Free Software, the effort has a strong moral purpose. Just as all people should be free from oppression and abuse, all software should be free from any restrictions of use, reuse, and distribution. To do otherwise is to limit the potential of the software and the people who use it. This is the driving force behind the Free Software movement: Freedom.

Open Source, on the other hand, finds its motivation in what it deems more practical matters. To supporters of open source, business, science, art, and all other endeavors that employ software are better served if the source for

29. https://opensource.org/licenses

that software is publicly available. To them, it's simply logical that opening the source enables types and levels of innovation that would be impossible with proprietary (closed source) software. This logic appears to be supported by the explosion of open source-based software companies and services in the nearly twenty years since the advent of "open source" as a movement.

To dramatically oversimplify it: Free software sees software freedom as a moral matter; open source sees it as a practical one. This is not, however, a hard and fast rule, nor is it a matter of two separate and disagreeing factions. The "difference" between free and open source software is actually a spectrum of a single belief that humanity is better served when software is freely and openly available. Supporters of free and open source software all fall somewhere on that single spectrum, but they all believe that freely and openly available software is a very good idea indeed.

From the perspective that matters most for this book—the nuts and bolts of contributing to a project—there is virtually no difference between free and open source software. Looking solely at contribution processes, there's usually no way to tell whether a project is free software or open source until you look at the `LICENSE` file.

An Aside About Terminology

As you participate in free and open source software projects, you'll find that people sometimes are a bit sensitive about the terminology used to refer to their movement of choice. While from a contribution point of view, there isn't much effective difference between free or open source projects, from a philosophical point of view, there is. The Freedoms guaranteed by free software form a fundamental belief system for many free software advocates. Therefore, some of them become sensitive to free software projects being referred to as "open source." To them, doing so dilutes the emphasis on Freedom embodied in the movement and removes the opportunity to teach new people about the freedoms and their moral importance. Regardless of your personal opinion, respect the Free Software movement and do not call free software projects "open source."

Also, you'll often see Free Software referred to as Free/Libre Software. This stems from the ambiguity of the word "free" in the English language. To those who are unfamiliar with the philosophy that underlies the movement, "free" may mean purely "free of charge" or "gratis." Because it's unlikely these people paid for the software, it's perfectly reasonable for them to think that there's no deeper meaning behind the word. "Libre," on the other hand, is not burdened with the multiple meanings that "free" carries. Stemming from "liber,"

the Latin word for "free" (as in freedom), Libre in modern languages is unambiguous in its meaning…for those who know its meaning, that is. Whether using "free" or "libre," the free software movement must educate those who use it in the underlying philosophy of the software they use, contribute to, and perhaps distribute.

Whichever type of project you join—free/libre or open—take note of how it prefers to be labeled and respect that choice.

Because the contributing process is similar for both free and open source projects (inasmuch as there is similarity between projects at all), and because I support both the free and the open source philosophies, in this book I use "free," "open source," and "FOSS" ("free and open source" abbreviated) interchangably with a preference for "free and open source." I don't use "free/libre and open source" or "F/LOSS", because I find these terms clumsy, and after the introduction to free and open source above, entirely unnecessary. There is no ambiguity when "FOSS" is used in this book, so there's no need for "F/LOSS."

A Brief Introduction to Copyright and Licensing

A lot of the content above has been all "license" this and "license" that without a lot of context on what a license actually is and why it's such a big deal, particularly for free and open source software.

So it's time for a very brief introduction to copyright, a complicated matter without which free and open source software wouldn't exist. As you saw above, Richard Stallman realized that he could use the existing copyright laws and systems to ensure software would always remain Free through careful licensing. Copyright therefore underpins everything in FOSS. Without it, and without an understanding of it, FOSS is not possible. Keep in mind: copyright law is a complex subject, so this is only a rudimentary introduction. Also, I am not a lawyer. What follows is not legal advice, only guidance to help you understand some of the basic concepts and complications of copyright.

When you create something, be it artwork, music, writing, software code, or any other creative endeavor, by default you own the copyright over that thing. This is an oversimplification, because in some countries and jurisdictions, you have to register something to get copyright. It's not as common anymore, but it's common enough that you might want to check on how copyright is assigned in your country.

However the assignment happens, as the copyright owner, you have the right to control how that thing can be used. This control comes through *licensing*

the work. A license is a legal document used to give people or entities permission to use copyrighted material. If someone else would like to use your work in any way, you can provide them a license that details the specific ways they may use your creation. A creator can apply the statement "All Rights Reserved" to their work to indicate that they don't want anyone to reuse or repurpose their work in any way; the creator has reserved the reuse or repurposing rights for themselves alone.

Things become complicated when there are multiple creators of a work. Each one of the creators, by default, retains copyright over the portions that they contributed to the whole work. If you program a piece of software, you have copyright over the code you wrote for it. If I come along and add a unit test for your software, I have copyright over the code I wrote for that test. The entire piece of software now has two copyright holders involved somehow.

Free and open source software licenses can help when there are multiple copyright holders for a single piece of software. These licenses often (but not always) contain a statement requiring contributions to a project (the unit test in the above example) to be contributed and released under the same license as the original work. This helps to keep the copyright and licensing complexities more comprehensible. As you can imagine, in a large project, questions of copyright could easily become mind-bendingly complicated.

Whatever creative work you contribute to a project, unless you agree to assign your copyright elsewhere (as can happen in a work for hire or a Contributor License Agreement situation, both covered later in the book), you retain copyright over your contribution and—if the project is released under an OSI-Approved License—your contributions will be publicly available. This means you can build a professional portfolio without fear of breaking copyright law or violating someone else's copyright.

This is not the case for creative work you contribute for your employer. Internships, freelance, hourly, and full-time jobs are all what is called *work for hire*. Unlike FOSS contributions, by default, the copyright on any work you contribute to a work for hire situation *belongs to the organization paying you*. Once you contribute that work to the organization, you no longer have any rights over it at all, and you *may not share it* in any form without very express and very written permission. It is *illegal* to share any creative work for which you do not have copyright or that is not licensed in such a way that it may be made public. This holds true for code, designs, documentation, project plans, or anything else that you create in a work-for-hire situation.

If you are interviewing or applying for a new position, and the prospective employer asks for work samples, you *must not* share anything that you created for past or current employers unless you can demonstrate that they have given you permission to do so. If you share private and proprietary work of past employers, how do you think that makes you look to your potential employers? Answer: Like a thief. You will have just proven to them that you cannot be trusted to keep their secrets. Why would they want to hire someone like that?

A portfolio comprising contributions to free and open source software contributions avoids the legal, moral, and reputational risks of sharing samples from proprietary work-for-hire creations. It not only allows you to highlight your skills, but it also demonstrates that you are ambitious and passionate enough about technology that you're willing to dedicate time outside of work to learn and contribute back to the community.

Because nothing is simple where copyright law is involved, there are, of course, exceptions to the work-for-hire copyright ownership rule. This comes in the form of employment agreements, proprietary information assignment agreements, and similarly named and intentioned legal documents. These usually come into play when you start employment with an organization, and they detail who owns the intellectual property (has the copyright) for what creations in which situations. Often these will declare that anything created on company property (computers) and/or on company time is the property of the company. However, thanks to the rise of free and open source software, some companies such as GitLab[30] and GitHub[31] have employment agreements that allow employees to retain copyright over their free and open source software contributions for the duration of their employment, regardless of when or how these contributions were created. This practice isn't yet common, and you should carefully read and review your employment agreements before signing them, regardless.

On the other side of the copyright ownership exception coin, we have Contributor License Agreements (CLA). These are discussed further in Chapter 3. Some (but not all) CLAs include the requirement that the contributor assign the copyright of all of their project contributions to the organization that oversees the project. This gives the organization the ability to enforce that copyright, or even to change the license under which the project is distributed, without having to bother every contributor to ask their permission. CLAs are

30. https://about.gitlab.com/2017/12/18/balanced-piaa/
31. https://github.com/blog/2337-work-life-balance-in-employee-intellectual-property-agreements

legal documents, and like all legal documents, it's important that you read them before you sign so you know what you're getting yourself into.

Types of Free and Open Source Software Licenses

The best place to learn about the various types of free and open source software licenses is the Open Source Initiative Licenses list.[32] It can be a little overwhelming at first, so to get you started, here's a quick introduction to the two basic types of FOSS licenses: copyleft and permissive.

Per the Open Source Definition, mentioned in *The Origins of Open Source*, both types of licenses share the requirement that anyone who uses works licensed under one of them must be able to view, modify, and share the source of the work. The difference comes after that: What can the user then do with the work? Can they change the terms under which people can use it? Or must the work be redistributed under the same terms by which the user received the original work?

For software distributed under a *permissive license*, someone who makes a change and redistributes the software is *permitted* to change the terms and conditions under which someone can use the new distribution (also known as a *derivative work*). In other words, the creator may change the license of the derivative work to one that's different from the original work. This affords the person releasing the new distribution a lot of flexibility in defining how the derivative work may be used. Permissive licenses also allow a creator to use a work released under this type of license in a proprietary work. When that proprietary work is released, it can remain proprietary. The permissive license of its component(s) does not force the creator to release the work under any sort of free or open source license. Two popular permissive licenses are the Apache License[33] and the MIT License.[34]

While permissive licenses allow a creator a lot of flexibility when distributing a derivative work, copyleft—or reciprocal—licenses protect a work from being relicensed under what may end up being a more restrictive set of terms and conditions. Once a work has been released under a copyleft license, the license ensures that the work can never be released under a license that may in any way remove or diminish any of the original rights and freedoms (specifically, the Four Freedoms mentioned in *The Origins of Free Software*) granted to the user by the license. A redistributed or derivative work released under a

32. https://opensource.org/licenses
33. https://opensource.org/licenses/Apache-2.0
34. https://opensource.org/licenses/MIT

copyleft license must also not add new restrictions to what the user may do with the work. This ensures that this work, once freed, will forever be free. Copyleft licenses also have a requirement that any derivative works made from software licensed under one of them and distributed must be released under the same terms and conditions as the copyleft-licensed work. This is the *reciprocal* nature of this type of license: if your creation benefits from a copyleft licensed work, then anyone who receives your creation must similarly benefit from your work. The GNU General Public License (GPL)[35] is the most common copyleft license. Other copyleft licenses include the GNU Lesser General Public License (LGPL)[36] and the Mozilla Public License.[37]

It probably won't surprise you to hear that, as with every other legal issue discussed in this book, what you've just read is an oversimplification of how these two different types of licenses actually work. Each type contains licenses that are more or less permissive and more or less reciprocal. Generally speaking, of the OSI-approved licenses, the MIT License is considered the most permissive and the GPL one of the most reciprocal. All other licenses fall somewhere along the spectrum between these two.

Now You Have a Strong Foundation

All of this history, philosophy, and law is complicated, I know. Don't feel you have to understand it in depth to contribute to free and open source software. Having the background knowledge of the philosophies, knowing that there are two basic types of licenses and the general characteristics of each type, is more than enough. There are millions of FOSS contributors in the world, and most of them get by very well with no more license knowledge than what you've just learned in this chapter.

Now that you have some sense of what FOSS is and what it's done for the world, you may be wondering what it can do for *you*. The next chapter fills in that blank.

35. https://opensource.org/licenses/gpl-license
36. https://opensource.org/licenses/lgpl-license
37. https://opensource.org/licenses/MPL-2.0

CHAPTER 2

What Free and Open Source Can Do for You

The previous chapter explained the history and philosophies behind free and open source software (*FOSS*). For many people, this philosophy is reason enough to contribute, but others need more motivation to devote their free time to participate in FOSS projects. If you're reading this book, you obviously have some interest in contributing, but do you really know what you hope to get out of it? Why would you invest your precious time in something for which you don't get paid?

Contributing to free and open source software doesn't have to be a purely altruistic pursuit. Contributors gain a lot for the effort they invest, and all of those advantages will pay off as their careers evolve.

FOSS Benefits to Your Skillset

Most obviously, contributing to free and open source software allows you to learn and practice new skills in a safe environment. It's possible to learn these skills on the job or in the classroom, but FOSS allows you a larger variety of options not only in skills to learn, but also in opportunities to practice them and gain experience. Sometimes, it may even be a safer place to practice those skills. If you do something wrong on the job, you may be reprimanded or possibly fired. If you do something wrong in class, your grade could suffer. In FOSS, if you do something wrong, you apologize and seek help to learn how to do it better.

This is, of course, an oversimplification of the matter. There still are repercussions for making mistakes when contributing to free and open source software. Thanks to the strongly social aspect of FOSS, sometimes these repercussions can have sustained impact. While you can revert a bad contribution, you can't do the same for hurt feelings. Despite that risk, contributing to free and

open source projects is still a relatively safe way to learn new skills that you can apply to your life and your career.

So, what are those skills, anyway?

Communication

Free and open source is composed of people, and therefore, contributing to FOSS projects can do wonders for your communication skills. By necessity, the community around a free and open source project will be distributed, often worldwide. This poses interesting communication challenges for getting anything done in the project. All communication usually is asynchronous due to differences in time zones and maintainer availability. Asynchronous communication often is impersonal communication, which can cause problems. The lack of real-time feedback like body language and facial expressions can lead to misunderstandings and delays. These same problems are common in "real-world" jobs, particularly for distributed teams. You gain experience by contributing to FOSS projects, and this helps you interact better in your day-to-day life and work.

Among the communication skills you can learn by contributing to free and open source is how to ask questions. Blurting out an open-ended and context-free query on the mailing list or issue tracker can lead to a lot of frustration and additional back-and-forth before someone can provide an answer. For instance, "Hey, is anyone else having problems running the latest version on their MacBook?" is a question that can lead to a lot of inefficient back and forth communication as people try to narrow down exactly what sort of a problem you're having. "I'm trying to run the latest version on macOS, but it keeps crashing with a `FILE NOT FOUND` error. Is this a known issue?" is a much better question and much easier for the community to answer. You've told them the version of the software you're running, the platform you're running it on, the behavior you're seeing, and the error message that accompanies it.

You also can learn how to set up expectations. Will you deliver this feature this weekend, or will it be delayed due to family obligations? Will you be able to complete a task on your own, or will it require the assistance of someone else? This type of communication prevents a lot of disappointment and delay in a project where each person depends on the work of others to proceed with their own.

While this asynchronous communication is necessary for a distributed team of project maintainers, it can lead to an unintentional lack of knowledge of or empathy for the people at the other end of the line. This often ends with

someone accidentally saying something that's offensive to others in the group. What is intended as a joke can come across as a personal slight or attack. This is particularly common in diverse communities, with equally diverse cultures and social interaction styles. Paying attention to your words and intentionally practicing your communication styles while contributing to a FOSS project makes you a much more pleasant team member for all of your jobs from there on out.

Finally, the distributed and asynchronous nature of a free and open source project requires that all communication be not only effective but also efficient. You enhance your value to any team by learning things like which types of messages are best suited to which medium: short and ephemeral? use chat; discussion and archived? mailing list; benefit from immediacy? conference call. Knowing how to write a good bug report—one that provides the context, expectations, and actual behavior witnessed—also helps you learn how to write other documents and messages more effectively. Paying attention to how to use your words more effectively and efficiently will make you a more productive communicator overall. The training that contributing to free and open source software provides in how to communicate effectively and efficiently will pay dividends throughout your career. *(as yet) unwritten content* goes into the topic of communication in detail.

Collaboration

If you take courses at a college or university, you undoubtedly have had the experience of doing a group project. You and several of your teammates are partnered up to complete a task. The goal of this is to teach you how to break down a project and collaborate on it with each person sharing the load. The reality usually is that the attempt at collaboration is contrived, and one or two people end up shouldering most of the load for the others.

You'll be happy to hear that this is not what true collaboration is. Free and open source software, due to its inborn distributed nature, requires true collaboration to work well. If there's more than one person involved in the project development, then some sort of collaboration processes emerge. The processes themselves vary from project to project, and they won't always work smoothly, but they typically will be far more effective than those you learned in school. So what are some of these collaboration processes?

For starters, there's the division of labor. Whereas in school you may have been stuck doing the lion's share of the work on an assignment, that's unlikely to happen on a collaborative FOSS project. There are multiple reasons for this. For one thing, as someone starts a task and realizes that it may be

larger than they originally thought, in open source they usually start a discussion about the task and how it can be broken up or otherwise staged in smaller parts. This public discussion encourages others to chime in, not only with their thoughts but also with their time to help work on some of those smaller parts. There's no shame in FOSS for saying that a task is too large for one person to tackle alone.

Another reason for dividing the work into smaller pieces is risk management. We'll cover *atomic commits* later on in *Make a Contribution*, but summarized: commiting smaller, discrete pieces of work rather than huge chunks makes the work much easier to review; a small commit has a better chance of receiving a thorough review, and therefore bugs are easier to spot. Atomic commits are also simpler to roll back should something go wrong. Both the review and the easy rollback mitigate the risk of fatal bugs slipping into the project.

Finally, there's the matter of *bus factor*. This is a term you may hear frequently in software development.

> *Bus Factor* is a number equal to the number of team members who, if run over by a bus, would put the project in jeopardy.

A macabre metric, no doubt, but also a helpful one. The worst possible bus factor for a project (or part of a project) is *one*. If only one person is familiar with that piece of the project, and that person goes away, the project will find itself in an uncomfortable position. Therefore, dividing up the labor on a feature or task increases the bus factor for that part of the project. Now, rather than just one person being familiar with that piece of the project, two, three, or more people are. When more than one person is familiar with the work, someone is always there as a backup should one of those people move on for some reason (hopefully on a bus rather than under it).

Tools

Almost as important a lesson as collaboration itself are the tools that make that collaboration possible. While the tools vary from project to project, the general project management, communication, and collaboration ideas those tools represent remain the same both across free and open source projects and even into the private sector. For instance, *issue tracking* not only allows a project to track its bugs and features, but it also helps provide oversight and accountability for the work being performed. If used properly by the addition of copious notes, issue tracking also forms a valuable historical resource that can enable future generations to learn from the experiences of those who came before them.

Without *version control*, real collaboration on free and open source would be nearly impossible. Version-controlled files can be edited by multiple people—sometimes even simultaneously—and then have all of the edits merged into a canonical version of the file. The messages included whenever a change is committed to a version controlled project (*commit messages*) themselves are another valuable historical resource. It's best practice for a commit message to provide details not only of *what is changed* in the commit but also *why* it was necessary and *what problem* the commit fixes. By reviewing a series of good commit messages, it's possible for other contributors to the project to follow its evolution and better determine how to engage with the project and the community around it.

Issue tracking and version control commit messages are two forms of asynchronous communication. Free and open source software collaboration would not function without async communication. The community of contributors for a project may span the globe and certainly will span a variety of personal schedules. Were collaboration to rely purely on real-time communication, no one would ever get anything done. For this reason, many free and open source projects rely heavily on asynchronous discussion methods such as mailing lists. People can read and collaborate on their schedules, and the project can keep moving forward. The ability to express your ideas efficiently in a textual method like a mailing list is a skill that will serve you throughout your career, and few opportunities to learn it will be as practical as participating in a mailing list for a free and open source project.

Best Practices

School is a great place to learn about Big O Notation or the golden ratio, but it's usually not as good for learning about current industry best practices. College graduates entering the workforce often find that while their coursework was heavy on theory, it was relatively light on the practice, technologies, and trends that are required for success on the job. Schools aren't to be faulted in this. They do a great job, but are time constrained in a way that industry is not. Curricula take time to develop, so institutions of higher education often must teach technologies and practices that are at the tail end of current industry usage.

Not so with free and open source software. Because FOSS is constantly moving, evolving, and innovating, many of the current industry best practices either originated in free and open source software development or were perfected by it. Version control, feature branches, unit and integration tests, continuous integration and deployment (CI/CD), design patterns… When you

contribute to FOSS, you master many concepts and best practices that you may never get to learn in another environment. More importantly, because you're hands on with these concepts, you have the opportunity to learn not only how they work, but also why they're important to do at all, and learn first hand the difference they make to a successful software project.

Technologies

Despite the fact that it's the very first skill benefit most people consider when they start thinking of contributing to FOSS, new technologies are in fact the least important skill you can learn. Of all of the benefits you can gain by participating in free and open source software development, the technology used by a project—while interesting—may be the benefit with the least staying power across the course of your career.

If you have a career in tech—at a software firm, or working with technology in a different context—your entire career will become a continuous parade of new technologies. Some people are able to build an entire career around a single technology (COBOL, for instance), but the majority of us must constantly be learning The Next Big Thing to stay relevant and employable.

Therefore, the technologies you know and use on a daily basis will be constantly shifting. Not so, all of the other skills mentioned in this section. So once you learn how to collaborate well with a group of distributed and diverse individuals, that's information you'll use for the rest of your life. The people skills you can learn from participating in free and open source software can serve you far better than the technological skills.

That said, you will have plenty of opportunities to learn new technologies with FOSS. Heck, considering how integral free and open source solutions have become to the infrastructure underlying most software and technology today, you may even get the opportunity to help build The Next Big Thing that you would otherwise have to learn from books and blog posts.

FOSS Benefits to Your Career

Many people in technology forget that software isn't the only thing that needs developing; their careers do, too. While your managers and mentors can help here, your career development is your responsibility. It's up to you to make sure you're always learning and moving your career in a direction that makes the best sense for your goals and needs.

Free and open source software can be invaluable here. At work, you learn and use the technologies and architectures that are required for work projects.

These technologies may help pay the bills but may not be what you need to move your career in the direction you want. FOSS, however, offers you endless options for technologies and architectures. Once you determine your goals, you can turn to FOSS to see which projects will help you reach them.

Public Portfolio

Your contributions to free and open source software projects become a public portfolio of your skills and how you've advanced them over the years. As you start contributing to projects, start a log or portfolio for tracking all of your contributions. Don't simply rely on the projects' version control systems and hosting providers, as those can change. If you don't keep your own log of contributions, you can easily lose track of the smaller but still important contributions you make to projects. Finally, maintaining your own portfolio allows you to track those types of contribution that can't appear in a version control system, such as acting as a volunteer coordinator at a community event or mentoring new contributors. Maintaining your own record of all types of contributions makes it very easy to share your contribution portfolio with prospective employers.

Portfolio as Resume?

It is important to stress, however, that despite what many in our industry would like to believe, at *no point* does this portfolio of FOSS contributions replace a resume; it supplements it. A curriculum vitae (CV) or resume shows prospective employers two things: what you've done for past professional positions and what difference you made with those actions. This last point—the difference you made—is very important to communicate to prospective employers. They don't want new team members who have simply done things. They want team members who have done the right things, for the right reasons, and moved the entire team and company forward in some way: someone who made a difference.

While your resume will show your potential employer what you've done, your portfolio reveals how you did it. This is important, of course, but it's not as important as the what. That's because every team has its own particular preference for the how. Your portfolio may show them that you can create effective technical documentation for multiple audiences, but your resume will show them that your documentation reduced contact to the company call center, saving tens of thousands of dollars in support representative time in the first year alone. Therefore, don't give in to the trend to replace your resume

with a portfolio. By preparing both, you'll make a strong and positive impact on potential employers.

FOSS Benefits to Your Personal Network

When you mention the word "networking" to many in software development, often they'll do one of two things. Either they'll start telling you about this one time they had to fix their family's router, run their own DNS server, or brought down the entire work subnet because of a typo. Or, if they realize that by "networking" you mean interfacing with other humans, they may blanch and start nervously scanning the room for the closest exit.

Unfortunately, much of our popular and technological culture has trained us to think of networking as an Intimidating Event: a bunch of people gather in a room, shake hands, introduce themselves, and then say smarmy things to each other to drum up new business leads or sell something. While, yes, this sort of thing can qualify as networking, it's more of the exception than the rule. At its most basic, just as computer networking is simply a method for computers to communicate, human networking is simply people communicating with other people. That's it. It doesn't require a special event and it doesn't require special skills or tools beyond what's required to interact with the clerk at your local shop.

Besides some of the negative connotations and misinformation under which many of us work where networking is concerned, there's also the problem that a lot of us are more comfortable interfacing with computers than with other people. Our educations are focused more around solving equations, diagraming sentences, or memorizing dates than about how to hold extemporaneous conversations with our fellow humans. Communicating well requires practice, intention, and attention. If you haven't had the training or opportunities to get that practice, then that communication can be a very scary and uncomfortable thing to approach at first. Don't worry: it gets better once you start getting that practice.

If it's so difficult and uncomfortable for many people to network with others, why should they bother? What's in it for them?

You've probably heard the old phrase, "It's not what you know, it's whom you know." This is networking in an oversimplified nutshell. As you progress through your career, the people you meet along the way can have a marked impact (hopefully in a good way). This doesn't necessarily mean they'll hand you a job, though that does sometimes happen. The most important benefits of these relationships are the discussions, introductions, and information

sharing that happen in them. The information could be a pointer to a new technology that will solve a problem that's been vexing you, a what-if question that leads to the launch of a new product, an introduction to a new collaborator or mentor, or a lead on a new position. These benefits and more can come from building and maintaining collegial professional relationships. More than any technology you will ever use or create, the relationships you foster will help you thrive in your career.

Free and open source project participation provides the opportunity for you to meet a broader variety of people than you're likely to in your day-to-day professional life. Many projects include contributors from all over the world and of all culture, skill, and experience types. Contributing to and becoming a member of the communities around these projects gives you instant and easy networking. Simply by listening to and respecfully engaging with the people in the community, you have successfully networked. Congratulations! That wasn't so bad, was it? That's because participation in a FOSS project provides a ready-made shared context and conversation starter. It's very easy to open a dialogue with a stranger when you know that they share an interest and are working toward the same goals as you.

The relationships formed through contributing to free and open source projects may be the most valuable and lasting benefit. These are people who can be there for you when you need advice, feedback, collaborators, or just a good laugh.

Benefit from Preparation

Now that you have a better idea of how contributing to FOSS can benefit your life and your career, there's one more thing to do before you can start looking for a project to contribute to: learn the lay of the land. Your project hunting will go a lot better if you know what files and social structures to look for. The next chapter will prepare you with everything you need to get started.

CHAPTER 3

Prepare to Contribute

You've probably already figured out that contributing to free and open source isn't quite as easy as slinging some code at a project. After all, if it were that easy, there wouldn't be any need for this book. While the steps required for contribution can vary by project and by type of contribution, they generally follow this sort of progression:

1. Realize you want to contribute
2. Find a project
3. Find a task
4. Set up your environment
5. Work on your contribution
6. Submit your contribution
7. Receive feedback and iterate on your contribution
8. Contribution accepted!
9. GOTO 1

You've already realized you want to contribute, otherwise you wouldn't be reading these words. Congratulations on completing step 1 of the process! Look at how far you've come already!

Before we get started finding a project and a task for your first contribution, there are some concepts and terms that you should know. Learning these now will make it much easier to understand what you're seeing when you're reviewing projects. Think of this chapter as setting up some familiar guideposts on your path toward your first contribution.

Ways to Contribute

Throughout most of its history, when people have spoken about contributing to free and open source software, they've mostly meant making programming

changes. This led many people to believe that contributions are all about the code, and non-coders are neither needed nor welcome.

Nothing could be further from the truth!

Free and open source software is…well…*software*, so naturally rather a lot of code is involved. But anyone who's ever used software (all of you) realize that there's more to a successful software project than simply the code behind it. There's user interface and user experience design, and documentation as well. That documentation and user interface may require translation to other languages. All of this—code, user interface, documentation—requires testing and review for potential bugs and for stylistic consistency. Testing—either by the team or by end users—leads to bug reports. Bug reports mean that someone needs to triage those bugs to determine reproducibility and severity. And, of course, none of this is possible if there aren't people who are dedicated to organizing and managing the entire process, or people whose focus is to spread the word about and market the software.

To help you visualize the many different ways you might contribute to FOSS, check the tasks below that you think you could do for a project:

☐ Programming (any language)	☐ Accessibility design
☐ UI design	☐ UX design
☐ Web design	☐ Graphic design
☐ Documentation writing	☐ Documentation editing
☐ Translation (any language)	☐ Code testing
☐ User interface testing	☐ Accessibility testing
☐ Bug triage	☐ Release management
☐ Project management	☐ Community management
☐ Event organization and coordination	☐ Public relations and outreach
☐ Marketing	☐ Security review and testing

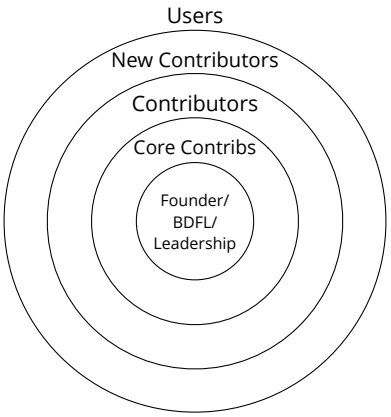
While this list has a lot of items, it's not complete. Some projects may have needs that aren't represented here. For instance, an open hardware FOSS project may need people who understand electrical engineering, while an open education project needs contributors who have a strong pedagogical background to write and review lesson plans.

Please don't feel you have nothing to contribute to free and open source software if you aren't a programmer. As you can see from the list above, FOSS requires many more skills than simply coding. There's a place for everyone to contribute in free and open source.

Common Project and Community Roles

As you saw in the chart in the previous section, the world of free and open source software needs many different skill sets to create successful projects. It also needs people to take on several different roles. Often you'll find a single person who has taken on a number of roles (especially in smaller projects); other times a project has multiple people taking on a single role to share the responsibility. However it ends up organized in the project, several roles are always at work at any one time.

What exactly are these roles? You've probably already guessed that they vary from project to project, but they typically fall into a few fairly standard categories. Academic authors *Walt Scacchi [Sca07]* and *Y. Ye and K. Kishida [YK03]* found it useful to use *the onion metaphor* to describe the categories of roles in free and open source software projects, where the most active and/or invested roles of the community are in the center, and the level of activity/investiture decreases as you work your way outward through the layers of the onion. Following is an example of a generalized onion model for free and open source community roles:



We'll go into a description of each of these roles in a moment, but first it's worth reinforcing that there's more than one way to organize a project, leading to different roles or categories of roles. Which roles a project has and needs are dictated by the project's technical and community needs and its governance structure, not by implied external pressures nor by best practices. Therefore, for any given project, you may not see all of these roles represented. They are, however, the ones most commonly found in FOSS.

At the core of nearly every project, you find the leadership. While the project founder often is a part of the leadership, it's not uncommon that the founder

has gone on to other things and left the project in the capable hands of other people. Sometimes a founder takes on the role *Benevolent Dictator For Life*, or as its more commonly known, BDFL. If the project has a BDFL, then when someone says, "The buck stops here," the role of "here" is played by the BDFL. This person has final say in and can veto all decisions. Typically, though, all leaders of FOSS projects—BDFL or otherwise—work toward consensus rather than impose their authority (hence the *benevolent* part of the title).

One step removed from the core of the community onion, you'll find the *core contributors*. These are typically the most senior or most experienced people in the project. Usually few in number, they provide guidance and mentorship for all other community members, and each one of them is a holder of a *commit bit*. This means they have the authority to approve a contribution (commit) to be merged into the main repository of the project. Having a commit bit is a big responsibility, and it's only given to the most trusted community members. If a core contributor gives you advice or feedback, you can trust that it comes from a place of experience and should be heeded.

Next in the onion are the non-core contributors. These folks provide somewhat regular contributions to the project and are fairly actively involved in most of the discussions. Frequently these contributors pitch in to review contributions from others, as well as provide advice and mentoring for newer contributions. While the BDFL and core contributors may be the heart of the project, these non-core contributors are the lifeblood.

Your onion layer is next! New contributors are, yes, contributors like those in the previous layer, but you're a special group that deserves a layer of your own. New contributors like you are still in their apprenticeship, still learning the ropes of how to operate within the project and its community. Given time and practice, you and the others in your layer will transition to being normal contributors and will be able to provide advice and mentoring for the new contributors who come after you. Projects that pay attention to their new contributor layer—making sure that it's easy for people to join that layer, the layer is well-populated, and these people are given the support necessary to become successful contributors—typically have very strong communities. These projects aren't as common yet, but they're worth seeking out.

On the outer layer of the onion are the users of the project. These folks are just as important as any other layer of the community. Without people using the project, there isn't much reason for the project to exist at all. Users also provide invaluable feedback, bug reports, and feature ideas to help keep the project alive and evolving. If the project starts adding features that the users didn't ask for or don't agree with, it might be a red flag that the leadership

way down in the core of the onion has lost sight of what the project is about and what the users need. The project's vitality depends upon meeting the needs of the users and helping to solve their problems. In that way, the users are possibly the most important layer of all.

These are just the most common roles found in free and open source projects, but knowing these few should help you navigate the hierarchy of most of the projects you'll interact with as you enter the world of FOSS contributing.

Files You Should Know About Before You Start

Before you start searching for and reviewing potential projects for your contribution, you should have some familiarity with what files and features you may see. Not all of the files mentioned here exist in all projects, but they're common enough that knowing about them makes it much easier for you to navigate projects.

Most of these files are located in the root directory of a project, but once in a while, you come across a project that placed these files elsewhere. If you don't see one or more of these files in the root directory, see whether the project has a `docs` or similarly named documentation directory. You may find the file you're looking for there. It's also possible, again, that the file simply does not exist in that project.

README

Typically the very first thing you see for a project is its `README` file. This is the project's face to the world. The `README` file tells you the name of the project and what it's intended to do, giving you a quick snapshot to see whether it's a project that might be useful or interesting to you.

The contents of `README` files vary. Some projects use the file simply to name the project and point you to other resources. Others include those other resources—installation instructions, developer setup, example usage—in the `README` file itself. The contents of this file are entirely up to the project.

Regardless of the contents, the `README` file should be your first stop when you visit any project. It can give you a very good sense of what the project is and where to look for more information about it.

LICENSE (also COPYING)

The `LICENSE` file (also spelled `LICENCE`) declares the terms under which folks are permitted to use, modify, and distribute the project. This file is also sometimes

called COPYING, particularly for projects that use a version of the GNU Public License (GPL),[1] but the purpose remains the same.

As you recall from *The Foundations and Philosophies of Free and Open Source*, if a project is not licensed under an OSI-approved license,[2] it cannot call itself an "open source" project. Doing so violates the definition of the term "open source." If a project is not licensed at all, then it is not "open source;" it is merely "source available." Furthermore, those who use or distribute projects that have no license are infringing on the copyright of the project's creators and putting themselves at risk of legal action.

It's only through that LICENSE file that a project can be "open source," and only through that LICENSE file that the project can legally be used, modified, and distributed. If you come across an interesting project that isn't licensed at all or has a license that isn't OSI-approved, be very careful before you contribute to it and throw yourself into a complicated and suspect copyright situation.

CONTRIBUTING

As a first-time contributor to a project, the CONTRIBUTING (also sometimes called CONTRIBUTORS) file is your best friend and bosom buddy, one you disregard at your peril. The CONTRIBUTING file sets out how the project prefers to receive contributions, the requirements and parameters a contribution must meet to be accepted into the project.

When you make a contribution to a project—whether it's your first contribution or your forty-first—always follow everything that the CONTRIBUTING file tells you to do. If you have any questions about its contents, always ask the community before you proceed. Once you receive an answer, be a good citizen and update the CONTRIBUTING file with the new information.

There's no standard format or contents for a CONTRIBUTING file. Each project includes what it thinks its contributors need to know about its particular contribution process. Some projects have separate contributor guidelines depending on the type of contribution. For instance, the Apache HTTPD web server has separate guidelines for reporting bugs, for contributing code patches, and for contributing documentation.[3] Other projects have all instructions in a single CONTRIBUTING file. The Public Speaking Resource project[4] handles their contribution guidelines in this way. There's no way to predict

1. http://gplv3.fsf.org
2. https://opensource.org/licenses
3. https://httpd.apache.org/dev/
4. https://github.com/vmbrasseur/Public_Speaking

what contribution guidelines a project will emphasize or what processes a project follows, so always be sure to look for a CONTRIBUTING file before you get started with your contribution.

If a project does not yet have a CONTRIBUTING file, but you want to make a contribution, what do you do then? For starters, you can look at past contributions to see how those were implemented and handled. Once you have that information, ask the community. "I'm going to make a contribution in *this way*. Is that OK?" If you always verify before simply tossing a contribution to the project, you'll always have a much better chance of your contribution being accepted. Once you've verified the process with the community and made your contribution, be a community superhero by writing it up in the first version of the project's CONTRIBUTING file. You'll be doing the community and its future contributors a huge favor, and you'll rack up yet another contribution.

Code of Conduct

The Code of Conduct (CoC) is a document that—thankfully—is appearing in more and more projects every year. The CoC sets forth the types of behavior that are both welcome and unwelcome in that project community, the consequences for unwelcome behavior, and where and how community members can report it. The intention of the CoC is to encourage behavior that creates a welcoming and safe place for all contributors, regardless of their gender, race, religious beliefs, age, or other characteristics and to provide recourse to those who have been victims of or witness to unwelcome behaviors. The existence of a Code of Conduct is a sign that a project values the safety of its community and welcomes contributors of all stripes.

Implementing a Code of Conduct on a project often is the cause of a lot of (sometimes not very friendly) conversation across the community. Because of that, the document is rarely the same across projects. To appease community members who are nervous about applying any limitations on interactions, some projects create a very minimal, "Be excellent to each other" Code of Conduct. Others create detailed documents that list behavior expectations, examples of unwelcome behavior, and enforcement instructions. While there's no standard for a project Code of Conduct, many projects now use some derivative of the Contributor Covenant,[5] originally created by Coraline Ada Ehmke.[6]

5. https://www.contributor-covenant.org
6. https://where.coraline.codes

A Code of Conduct is a valuable document, but it's only as strong and as useful as its enforcement. Without a community that stands by the words in the document, a CoC is no more than a writing exercise. When first viewing a project and its community, it's usually difficult to tell whether it's able to enforce its CoC in an effective and empathetic way. This shouldn't stop you from contributing to or joining a community. Having a Code of Conduct at all is a sign that the community is at least willing to do the right thing, a sign that is very welcome and welcoming to prospective community members.

Styleguides

It's probably no surprise to hear that free and open source software projects often have Very Strong Opinions™ on how things should be done, at least on their turf. We'll get into some of those Opinions in *Make a Contribution*. For now it's enough to know that these Opinions exist, and submitting a contribution that ignores those Opinions is a good way to irritate the project maintainers and community.

Projects that have these Very Strong Opinions usually take the time to codify them in *styleguides*. Depending on the project, you may find a styleguide for programming, for writing, for graphic design… It all depends upon the needs and preferences of the project. Sometimes these guides are included directly in the `CONTRIBUTING` file, other times they're standalone documents. Whichever way they're implemented, you must always read and follow these guidelines if they exist.

Knowing that styleguides usually spring from project-specific Very Strong Opinions, you probably won't be surprised to hear there's no standard at all for them. You never know what will or won't be included in a styleguide, so it's very important to read them (assuming they exist for a project). Sometimes you'll see styleguides reused between projects or used as a basis for a project's own styleguide. For instance, many projects use the Google Styleguides[7] for their coding guidelines. Others, like the OpenStack family of projects, rely on documentation styleguides such as that from IBM.[8] Until you check the styleguide, you won't know what style the project prefers for its contributions.

If a project does not have any sort of styleguide, it doesn't mean that the project lacks those Very Strong Opinions. It's more likely that they simply haven't yet gotten around to writing down those Opinions. Therefore, as you work through your contribution, try to note any stylistic preferences the

7. https://github.com/google/styleguide
8. https://docs.openstack.org/doc-contrib-guide/writing-style/general-writing-guidelines.html

project maintainers express. Once you've completed your contribution, you have yet another opportunity to don your FOSS superhero cape by writing up the project's stylistic preferences in their very first styleguide(s) and then linking to it in the CONTRIBUTING file.

Other Handy Files You May See

Those were the files you're most likely to encounter when browsing free and open source projects, but a few more are relatively popular, particularly in older or very well-established projects.

The INSTALL or INSTALLATION file is pretty much exactly what you would expect: instructions for how to install and optionally configure the project for use. This file is more common in projects that use make[9] for compiling and installing the software, but there's no reason it couldn't be included in any project (and it often is).

CHANGES or CHANGELOG is, again, fairly self-explanatory. This file contains a human-readable summary of all of the releases for the software and the changes that comprise each. The CHANGES file can be very handy if you're trying to determine whether the version of the software you're using includes a certain bug fix. It's also helpful for new contributors to see the development trajectory of the project.

The AUTHORS file is becoming less common in free and open source projects as they instead rely on version control logs to fulfill a similar purpose. It's still common enough (and valuable enough a tool) to warrant mention. The AUTHORS file lists all people or entities/companies who have made copyrightable contributions to the software. This file may include contact information for these people, but since that can violate a contributor's privacy, this information isn't often included anymore. Having a single, canonical list of copyright holders for the software can simplify copyright statements (that can now say merely, "Copyright 2018, The Authors") and also ease the process of changing a project's license. All copyright holders agree to provide their contributions under a certain license; changing it requires that they all approve to relicense their contributions. Without a list of all copyright holders, this already complex relicensing process can become nightmarish.

Issue Tracking

One of the key characteristics of free and open source software projects is that they are just that: *projects*. As projects, some form of project management

9. https://www.gnu.org/software/make/

is usually required to make sure all development proceeds smoothly. One of the most important of these is the *issue tracker*.

Issue tracking, bug tracking, ticketing system… Different terms but all the same concept: an issue tracker is where a project *tracks* individual *issues* in the project. Yeah, I know, with functionality like that, how did they ever come up with the name "issue tracker?" It's a mystery. Jokes aside, issue trackers are vital for making sure the project knows what is going on, when, and by whom.

The features of issue trackers vary by tracker provider, and many projects don't even use all of the features available. Some projects use the tracker solely for logging bugs in the software. Others use it for bug tracking, feature requests, support questions, design discussions, team conversations and debates… It all depends on the needs and workflow of the project.

The only wrong way to use a project's issue tracker is "anything different from how the project uses it." Don't inject your own preferences or workflow into a project's issue tracker. Sometimes a project documents its issue workflow. If it does, follow it. If it doesn't, have a look at completed ("closed") issues to see which workflow was used for them. As always: ask the community if you have any questions or even just to verify your assumptions. It's better to ask now than to do the wrong thing and make a lot more work for you and for the community.

Common Communication Routes

Since nearly every free and open source project has contributors spread all over the world, communication is vital to success. Over the decades, FOSS has evolved a series of tried and true communication routes that enable efficient, persistent, and effective communication across a variety of use cases. These routes fall into three basic categories: entirely asynchronous (email, issue tracking), semi-asynchronous (real-time chat), and synchronous (audio/video calls, in-person meetups). *(as yet) unwritten content* goes into detail about each of these communication routes.

For some FOSS projects, the selection and use of communication routes fall into that bucket of Very Strong Opinions that I mentioned previously. Each project uses their own combination of routes and process to meet their own needs, so make sure you seek out documentation and advice about this before you participate in any project discussions. Incorrectly using communication routes is a common way new contributors leave a poor first impression on a community they hope to join.

If you're going to participate in FOSS, you need to be comfortable with email. Many free and open source projects rely heavily on mailing lists. A mailing list allows a project with contributors distributed across time zones to receive and reply to conversations when it's most convenient for them. Mailing lists also allow people to take the time to think through and craft their responses for a discussion. This is particularly helpful and welcoming to community members whose primary language is not the same as that of the project. These people make insightful and valuable contributions to discussions but require a little more time to translate those thoughts into, for instance, English from Polish. Add to that the archivability and searchability of email threads, and mailing lists become a powerful tool for collaboration in free and open source software projects.

While mailing lists can allow for rich and nuanced conversations, there's nothing like a real-time chat for building comraderie and helping to coordinate a complex process. Many FOSS projects use a real-time chat system of some sort. Internet Relay Chat (IRC)[10] is a very popular option, but far from the only one. Other options include Matrix,[11] RocketChat,[12] and Mattermost.[13] The selection and use of a real-time chat system has taken on nearly religious significance in some free and open source software communities of late. Rest assured that no matter what chat system is in use by the projects in which you participate, a great deal of conversation (and possibly arguments) went into its selection and maintenance.

Respect the communication routes chosen and used by the project, as well as rules and guidelines they've set forth for their use. If you strongly object to the routes a project uses, rather than complain about it (passive-aggressively or otherwise), I advise you to select a different project to which to contribute. Your complaints will fall on deaf ears, and you'll simply alienate the community you'd wish to join. Respect their choices and the process that went into making them.

Contributor License Agreement/Developer Certificate of Origin

A few free and open source software projects require all contributors to agree to either a Contributor License Agreement or a Developer Certificate of Origin before their contributions can be merged and distributed with the software.

10. https://opensource.com/life/16/6/irc
11. https://matrix.org
12. https://rocket.chat
13. https://about.mattermost.com

While the number of projects that require this is still relatively small, it's increasing every year, as more projects join free and open source software foundations. Before you get started, it's worth knowing about these documents and how they might impact your contributions.

Some projects—especially but not exclusively, those developed under the aegis of a large corporation—require all contributors to sign a *Contributor License Agreement* (CLA). A CLA is a document resplendent with intellectual property implications and therefore, a controversial matter for some free and open source software practitioners.

The contents and prescriptions vary by CLA, but basically, one exists to make sure that you (or your company, if you're contributing on their behalf) have the right to share your contributions, agree that the project has a license to alter, distribute, and administer those contributions, and you agree that you will never revoke that license. Sometimes the document also includes a transfer of copyright from the contributor to the project or project's organizing body. The intention of the CLA is to minimize potential legal complications of distributing the work, as well as to potentially make it easier to change license.

As mentioned, CLAs are controversial for some people and projects. Many object that the requirement to sign a CLA before making a contribution not only slows down the entire contribution process and adds administrative overhead to the process, but also discourages many people from contributing at all. Other people object to the idea of signing over their copyright to another entity (again, that's not a feature of all CLAs).

Recently, the *Developer Certificate of Origin* (DCO)[14] has become a more popular alternative to CLAs. A short and simple document, the DCO ostensibly achieves the results of a CLA without the administrative overhead or related slowdown in contributions. A DCO relies upon a contributor signing their contribution using the `-s` or `--signoff` flags of the git version control system. This signing denotes that they have the right to distribute their contribution and do so under the same conditions as the project license. This means that the DCO can only be applied to contributions that can be committed to the project's git version control system…assuming the project uses git at all. If the project uses Subversion, CVS, or another version control system, it may not be able to use the DCO. So the DCO is not the right solution for all projects nor all contributions, but some projects find it a welcome change from CLAs.

14. https://developercertificate.org

You're Ready to Find a Project

OK, you're now equipped with the guideposts you need for a very basic navigation of a free and open source project. The next step is a fun one: Find a project where you can make your first contribution!

CHAPTER 4

Find a Project

The question I hear most often from people wanting to contribute to free and open source software is, "How can I find a project to contribute to?" or even just, "Where do I start?" You may have heard of open source and know that it's possible to contribute. You may even know why you want to contribute and what you want to get out of it, but it's rare for people to have recognized that a lot of thought needs to go into the choice of where to contribute. Mostly, what all potential new contributors know is that they want to contribute in some way. You're probably in this camp, and that's better than OK—it's great. You've passed the first milestone for contribution: wanting to contribute at all. The second milestone is finding the project that's right for you.

Finding a free and open source project to which to contribute isn't as simple as choosing a random bug in a random project. You can do it this way, sure, but you're unlikely to be successful or to have a positive experience. Before you dive in, give yourself a better chance of success: take the time to find a project that matches *your* goals and values. This, of course, implies you can actually articulate your goals and requirements, so that's where we'll start.

I won't lie: defining your goals and requirements and finding the right first project can take some time to do properly, but it's a very good investment. What isn't a good investment? Spending days, weeks, or months trying to contribute to a project that isn't a good fit for you.

Set Your Goals

You may know that you want to contribute to free and open source software in some way, but can you put your finger on exactly why you want to do this? The answer is a lot harder than it seems at first. Some people may answer, "to get experience," or "I believe software should be Free," or "my teacher/mentor told me it would be a good idea." While these might be motivations, they're

not goals. These statements are vague and difficult to pin down, therefore, it's also difficult to tell whether you've succeeded in them. Goals must be specific and actionable, otherwise they're just smoke in the wind.

After reading the past few chapters, you may have some more thoughts about why you want to contribute to free and open source software. *The Foundations and Philosophies of Free and Open Source* covered the philosophies underlying FOSS. These may resonate well with your own philosophies, values, and ethics in such a way that cultivating and spreading these philosophies may factor into your personal goals for contributing to free and open source. *What Free and Open Source Can Do for You* detailed some of the many professional benefits you might reap when contributing to FOSS. Some of these benefits may suit your own purposes, and may even have inspired you to think of personal benefits that were not mentioned (it was far from an exhaustive list).

Regardless of whether you feel you have a firm grasp on your reasons and goals for contributing, collect your thoughts and write them down. Doing so not only gives you a snapshot of your current state of mind, but it also gives you something to which you can refer later on. You might revisit your goals to update them, or—if you're having a bad day—to remind yourself why you're putting up with all of this in the first place.

Grab your favorite writing device and a cup of a tasty beverage and sit down to collect those thoughts. Give yourself permission to write anything that comes to mind, in whatever order those thoughts fall out of your head. There are no wrong answers or thoughts here, so collect them all in this brainstorming session without passing judgment or trying to organize them. The time to organize is later, after you've gotten all of your goal-related thoughts out of your brain and into the open where you can view them all at once. An example of a possible brainstorm is shown in the figure on page 41.

Once you've collected all of your goal-related thoughts, set them aside for a short while before you move on to the next step. Allowing your brain to rest will help give you a better perspective when you start to organize your thoughts, and may even allow a few straggling thoughts to bubble up and be captured in your brainstorm. So take a break: mow the lawn, do the dishes, play a game with your kids, watch a movie, or even just sleep on it.

OK, is your brain all rested? Good, because now we get to the hard part: taking all of those thoughts and organizing, consolidating, and focusing them into a list of goals.

Look at your list of thoughts. Are there any that are vague? Expand on them until they're specific. Are there any that are similar? Collect them together.

Goal brainstorming:

What do I want to get out of this? Why am I even doing it?

- ~~the teacher told me to~~
- ~~I don't want to fail the assignment~~
- practice my CSS
- I don't know any Javascript and want to learn some
- meet cool new people!
- we had a unit on accessibility in UI design, could I learn more about that?
- it would be really cool if something I designed were used by a lot of people
- could I put that in my portfolio?
- would force me finally to learn how to use git
- does command line stuff count as UI? could I design that?
- learn how to work with programmers better
- learn from experienced designers
- do hardware projects need designers? could I do some industrial design-type things?
- get better at writing

As you review them, make sure you understand the why behind every thought. If there are any for which you have no discernable reason beyond, "it seemed like a good idea at the time," disqualify them from the goal process and set them aside. Iteratively refine and collect your thoughts into categories until you've consolidated them into the few core things you would like to achieve by contributing to free and open source software. How many of these core goals constitute "a few" is up to you and your needs. Each goal should be specific, concise, and actionable, something you can state to someone and have them immediately understand what you hope to achieve. Vague goals are difficult to make progress on. For instance, "Practice programming" is a vague goal. Programming what? In what language? How will you tell when you've accomplished this goal? On the other hand, "Become more proficient and fluent at server-side Javascript" is specific and actionable. This is a goal that is easy for you to focus on and just as easy to see whether you're making progress toward it. See the figure on page 42.

My goals:

- Gain greater proficiency in CSS
- Start learning Javascript
- Learn how to use git with design artefacts
- Improve the accessibility of at least one UI
- Practice communicating and collaborating with programmers
- Write a design proposal
- Find a designer mentor
- Add at least one new piece to my portfolio

Remember: These are your goals, sprung from your own thoughts and needs. While they may possibly resemble those of others, these goals are entirely unique to you. Be true to your personal needs and goals; don't simply take the goals handed to you by a teacher or mentor. Own your goals, take responsibility for meeting them, and you are much more likely to be successful in your FOSS contributions.

An advantage of these being your own, personal goals is that you are free to change them as needed. These goals are not carved in stone. As your life and career evolve, your goals should as well. Revisit this page from time to time and review the goals you've written here. Do your goals still ring true? Do they still meet the needs of your life? If not, how should your goals change and, more importantly, why? If necessary, go through the entire exercise again from brainstorm to goals to ensure that you're still targeting goals that are good for you, your life, and your career. Don't spend years driving yourself toward goals that no longer serve your needs.

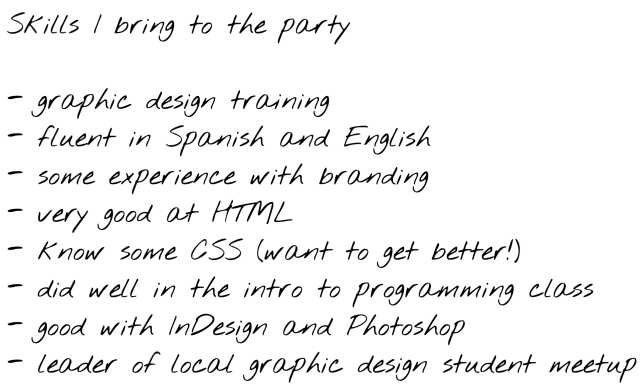
Collect Your Requirements

You have your goals figured out, so you're ready to go out into the FOSS world and find a project to contribute to, right? Nope, not quite yet. Your goals are only one piece of the puzzle. You also need to know your personal requirements for the project you select. Think of these as the criteria the project must meet to be a good fit for you. Contributing to a project that isn't a good fit is like wearing the wrong size shoes: They may look cute, but after taking a few steps, you'll be in quite a lot of pain. To maximize your chances of success with your first contribution, take a few minutes to figure out what size you should wear.

What do I mean by *requirements*? These are project characteristics that meet your own particular needs. Only you know what sort of characteristics are required for you to be successful, but I'll list some of the most common things that people should consider when looking for a free and open source project to which to contribute.

Skills

For starters, what are your skills? What can you offer to a project? Are you a great writer or editor? How about translation? Graphic design? User experience specialist? Know certain programming languages? Have experience with electronics? Maybe you have experience managing people, writing technical specifications or grants, or organizing events? All these skills and more are in demand for free and open source projects and communities. Take a few minutes to write down all of your skills that may be potentially relevant to contributing to FOSS.



Those are the things you can do, but what about the things that interest you?

Interests

You're much more likely to enjoy and stick with contributing to free and open source software if you're working on a project that interests you, rather than working on the first one you come across. Besides enjoying it more, if you choose a project for an interest you already know something about, then you have *domain knowledge*. This is knowledge about how things operate in that interest area. For instance, if you sew, knit, or fix cars, then you already know all of the terminology for sewing, knitting, or car repair. If you find a project related to one of these areas of interest, you'll more easily understand what the project does and perhaps even how it works.

There's a free and open source project for every hobby and interest area. When most people think of FOSS they immediately think of operating systems (Linux), infrastructure, databases, or web development. If these are your interests, you're in luck, since there's always a lot of work that needs to be done on these projects. But there are also projects for ham radio, sewing, game development, digital art, machine learning, astrophysics, geography, 3-D printing, education… The list goes on and on.

What sorts of things interest you? What are your hobbies? What classes did you enjoy in school? Take a few more minutes to write down all of your areas of interest.

Time Availability

Another very important requirement is your time availability. A single parent with three young children will have much different time availability than a second year university student. Before you start looking for a project to which to contribute, be honest with yourself about how much time you think you can devote to contributing to free and open source software. Some projects have a much steeper learning curve than others, so if you have only a little bit of time, you may need to limit your project selection to one that has a reputation as being very supportive of and helpful to new contributors.

No matter which project you choose, it's very possible to contribute even if you have only a couple hours a week to devote to it. Every contribution is

valuable, even the small ones. Be realistic about your time investment and take on only what you can manage. You can always ramp up your contributions later should more time become available to you.

Goals

The goals you defined earlier are also a part of your requirements for project selection. It doesn't make a lot of sense to contribute to a project if it's not going to help you move toward your goals in some way. Take the time to revisit them if it's been a while since you last did.

Skills, interests, time availability, and goals. These are your specific requirements, and they are unique. If you compare your lists to anyone else's, you may find some overlap, but you're more likely to find more differences. These requirements are yours and yours alone, and only you can define them. Others may be able to help you brainstorm or refine your lists, but no one can tell you what your personal requirements are.

And remember: all of these requirements can and will change over time as your life situation evolves and professional experience grows. Don't be afraid to revisit these requirements and refresh or alter them later. Doing so can help provide a lot of clarity if you're ever feeling a bit lost about where to look for your next project or challenge.

Collect Candidate Projects

OK! You have goals! You have interests! You have requirements! Now all you need is a project. How hard could it be, right? Wellllll…

As I mentioned in *The Foundations and Philosophies of Free and Open Source*, millions and millions of free and open source projects are in existence today. How are you supposed to apply those goals/interests/requirements to millions of projects? Answer: by limiting the pool of candidates.

Start by looking at the projects you already use and enjoy. If you're a Linux user, then you probably have a lot of free and open source software projects that you use on a daily basis. Blender,[1] GIMP,[2] KDE[3] or GNOME,[4] and all of the tools associated with them are all FOSS projects. But daily use of free and open source isn't limited just to those who run Linux on their machines.

1. https://www.blender.org
2. https://www.gimp.org
3. https://www.kde.org
4. https://www.gnome.org

FOSS projects are everywhere: Drupal,[5] Moodle,[6] Visual Studio Code,[7] iTerm,[8] and more! Look at the software you use every day, then check to see whether it's a FOSS project. While these large and very visible projects may not be the best starting point for someone new to FOSS contributions, they may have smaller satellite projects (such as libraries, plugins, extensions) that are perfect for someone who's just starting out.

Even if the software itself is not free or open source, it's possible that an ecosystem has sprung up around it that is. For instance, if you use the Unity[9] engine for game or video development, you'll find that a lot of the plugins for it are released under OSI-approved licenses. If you're a Mac or an iOS developer, you're probably using tools or libraries that are released as open source. Nearly all browsers allow for third-party extensions now. Many of those extensions are available as FOSS projects. So take the time to inspect your software and its ecosystems. You're likely to find you've been using and enjoying free and open source software and didn't even know it.

All of those interests you listed previously make for a great starting point for locating free and open source projects. Open your favorite web browser, fire up your favorite web search engine, and type an interest name followed by the words "open source" into the search field. The results will undoubtedly point the way to a lot of FOSS projects that you never knew existed. For instance, if I type `woodworking open source` into my search engine today, I receive 772,000 search results. `sewing open source` returns 1,920,000 results. `painting open source` returns an eye-popping 7,880,000 results from this search engine. Enter each of your interests into a search engine in this way and see whether any intriguing free and open source projects are revealed. If they are, add them to your list of candidates.

Another way to locate interesting free and open source projects is to browse popular version control service providers. As I write this, the most popular of these providers for free and open source software are GitHub,[10] GitLab,[11] and BitBucket,[12] but there are others (including self-hosting by the projects themselves). Most of these services offer a way to explore the public repositories

5. https://www.drupal.org
6. https://moodle.org
7. https://code.visualstudio.com
8. https://iterm2.com
9. https://unity3d.com
10. https://github.com
11. https://gitlab.com
12. https://bitbucket.org

they serve. Sometimes the service provides a special page for this purpose, highlighting and categorizing projects by topic, programming language, popularity, or some other characteristic. These services and pages can be a great way to discover projects you might not locate otherwise.

Your network and local community can be a great resource for finding free and open source projects to which to contribute. Do you have friends who have contributed to FOSS? How about your social network (Twitter, Facebook, and the like)? Ask them about their experiences and whether they can recommend projects that might be a good fit for you. Nearly as important, ask them whether they've had any bad experiences with projects. It's much better to learn now that a project has a toxic community or is difficult to contribute to than to learn it later the hard way. You can also just put yourself out there on social media and offer your services. "I would like to contribute to a FOSS project! My skills are... Does your project need my help?" I have seen this work to good effect, but the success of this method depends a lot upon the reach of your message. If it doesn't get in front of the right people, you're unlikely to receive many helpful responses.

While you're doing your research to locate candidate projects, simply add them to your list as shown in the figure on page 48. There's no need to research or compare them yet, and it'll be easier to compare them once you have a better idea of the options that exist for you. Also, it could be that a few projects keep reappearing in your searches. The more often you come across a project while doing searches that are targeted toward your requirements, the more likely it is that the project may be a good fit for you. Regardless, invest an hour or two to collect a nice pool of candidate projects and to familiarize yourself with the landscape of free and open source projects that exist in the world.

Select a Project

You've done a lot of work by this point, so the next step may not take you very long. It's time to select a project where you can start contributing. You could do this the old fashioned way by throwing a dart at a dartboard covered in potential projects, or you could do it the smart way by comparing the list of projects you've built with your list of requirements. It's possible that there won't be a single project that meets all of the requirements on your list. That's OK. As long as it meets some of them, you'll still be moving toward your personal goals.

While matching your requirements is a very important feature for any potential project, it's not the only one you should take into consideration.

Candidate projects

- InKscape
- Scribus
- Blender
- GIMP
- Krita
- Godot
- Twine
- ORX
- melonJS
- Aseprite
- Zuluru
- GoldenCheetah
- Sonic Pi
- HeaterMeter

There's also the matter of how easy it will be for you to contribute. This will be your first contribution, after all. Why not give yourself the best possible chance of success by choosing a project that makes contributing more straightforward? It may feel like stacking the deck in your favor…and you'd be right. But there's nothing wrong with that, is there? If you start your free and open source software contributions with an easy win, you'll be much more motivated to continue down the path of contributing elsewhere.

Have a look at each project on your list, starting with the documentation. Does the project have a `CONTRIBUTING` file or similar documentation guiding people through the contribution process? Does it have robust documentation for setting up a developer environment? Are the communication routes for the project documented and active (people who ask questions receive answers)? If so, you may have a good starter project on your hands. Next, have a look at the project's issue tracker. Are there any open bugs or features that you think you might be able to tackle? Maybe some of them are tagged as *Help Wanted*, *First Timers Only*, *Newbie*, *Good First Issue*, *Up For Grabs*, or some similar flag to highlight them for people like you.

It's certainly not required that a project do all of these things. Thousands of very good projects are out there, supported by healthy communities, that do not meet all of the criteria in the preceding paragraph. However, if you find

a project that has implemented even one of those criteria, you'll find that it will make your first time contribution experience much more pleasant than contributing to a project that has not.

Once you've reviewed all of the projects on your list for how easy they may be to contribute to, your choice of starter project may now be obvious to you. If it's not, don't worry. For some people, it's better to explicitly list the pros and cons for each project, then analyze and review them all together this way. There's no right or wrong method for coming to your decision. Do what's best for you and the way your brain works.

Remember, though: when you decide on a starter project, that decision is not carved in stone. You may find that the community is not as welcoming as you hoped, or once you start contributing, you aren't getting what you need out of it. If that's the case, it's perfectly OK to stop contributing there and find another project where you can devote your time. I caution you, though: before you quit, consider whether you might be the problem. In your zeal to contribute, it may be that you're not doing as good of a job as you could in communicating or understanding the contribution process. Ask the project community for feedback, help, and mentoring (just don't expect to be spoon-fed). If these things aren't forthcoming or don't help you feel more comfortable with the project, don't hesitate to move on to something that is a better fit for you.

Select a Task

You have a project! Congratulations! Now you're ready to get started on that first contribution, right? Well…sorta. Before you can make that first contribution, you have to figure out what it will be. You have to decide upon a task.

It could be that you already have something in mind. You may have discovered a bug or typo in documentation, or docs that are missing altogether and you'd like to add. Perhaps there's a bug in the software, something that's been bothering you for a while and which you can easily reproduce. Maybe you use a certain library for a work project, but to continue, you need to add a feature to the library API.

Whatever the task, before you start work on it, search the project's issue tracker to see whether it exists. Don't limit your search purely to open or active issues, either. Search the closed issues to see whether your idea was proposed before, but the project decided not to pursue it for some reason.

If your idea doesn't exist in the issue tracker, open a new issue. This serves two purposes. First, it warns the project that a contribution may be on its way. Second, it allows the project maintainers to review the task and confirm that

it's something the project needs or wants. It can be very disheartening to put a lot of work into a contribution only to learn afterward that it's not a good fit for the project, so do take the time to write it up in an issue in advance.

If you don't already have a task in mind, a good source for one is the project's issue tracker. Most every project has one of these, though they may use a different name for it; bug database and ticketing system are two other common names for this. Most of these systems include some way to 'tag' issues to make them easier to categorize and locate. The tags vary from project to project, but often a project has a tag that's used to mark certain issues as suitable for new contributors to tackle. Examples of tags that may mean this are: *easy*, *starterbug*, *newbie*, *help wanted*, or *good first ticket*. If a project tags issues as suitable for new contributors, they usually mention that and what tag to look for in their contributor guide, so look there as well. Whether the project has tagged issues in this way or not, review the issues and select one that looks achievable, considering your personal skillset and experience.

While working on finding a suitable task, never underestimate the power of Just Asking. Do your own research to familiarize yourself with the project, its needs, and its communication routes. Pick the most appropriate route—this will vary by project—and introduce yourself. Let the community know who you are, that you're new and excited to help out, and briefly state your skills so they'll have some idea of your current capabilities. If you already have an issue in mind, let them know which one and verify that it would be appropriate for you to work on. If you haven't chosen an issue yet, ask whether someone can direct you to one or whether you might help someone with a task they're already working on. When you write to the community in this way, please be patient and respectful of their time. They may not be able to reply to you very quickly. It's not personal; it's just that they all have their own lives and obligations to attend to as well and may be many time zones away.

As you filter through tasks to find a good one for you, I encourage you to start small. Yes, you have goals you wish to fulfill through your contributions, but free and open source participation is a marathon, not a sprint. Take the long view, particularly when you're starting out. Small tasks lead to a quicker payoff and better chance of success than trying to tackle a large feature or tricky bug. This payoff takes the form of the endorphin hit you'll get when your first contribution is accepted by the community, and it feels great. The larger and more complicated the task you select, the longer you postpone getting that payoff, so start small. Baby steps are still steps and still move you toward your goal.

Similarly, simple and repetitive tasks not only allow you to contribute quickly, but they also help you make friends and influence people in the community. By taking on these important but less fun tasks, you not only free up the time of more experienced community members, but you also show them that you're willing to dig in and do what it takes to lend a hand and work your way up through the ranks of the community.

What Is "Success"?

All through this chapter I've repeatedly said, "do this thing to maximize your success" or similar statements to that effect, but I've never defined what *success* is.

That's because I don't get to define what success is for you; only *you* can. Without goals, without requirements, you can never truly know whether you're making the right choices for your own needs. Those goals and requirements are very personal things. Your goals will not match my goals; your requirements will not match my requirements; your success will not match my success. Despite that, some general characteristics of your contributions can signal whether you are or are not on the path toward your success:

- You're able to make a first contribution with minimal fuss
- You're welcomed by your first community
- You learn and grow from your experience
- You gain the confidence to help others contribute, too

You will not see all of these characteristics at once. For instance, you won't necessarily have gained the confidence to help others contribute until you've contributed a few times yourself and are more familiar with the process (at least for that project). It's possible to be welcomed by a community before you make your first contribution to it. So if you don't see a particular characteristic, don't worry. It may just be waiting around the next corner. However, if after trying to contribute for a couple of months, you still don't see any of these characteristics, do consider whether the project you've chosen is the right fit for you. Don't be like Don Quixote, tilting at windmills to no end. If you're not making any progress, it's OK to set that project aside, re-collect your notes on project selection, and try another one. You won't meet your goals if you end up a crumpled heap at the foot of a windmill somewhere, so stop tilting at them.

CHAPTER 5

Make a Contribution

At this point, it can be tempting to just jump in and start working on your contribution. For some people and some contributions, this might even be successful, but for the rest of us, it's not usually that easy. Don't worry, though: you can do this, and by the end of this chapter, you'll know what to do to make your contribution successful.

Making your first contribution to a project can be complicated. If the project isn't well documented, if its community isn't very communicative, if your contribution is complex, if you don't have a lot of time or other resources available… Plenty of things could cause your first contribution to go a little less smoothly than you'd wish.

It's OK at this point to pause and think through the process and your contribution before you submit it to the project. Don't dive in headfirst; wade in instead. Basically, there are five large parts to any free and open source contribution:

1. Prepare
2. Craft
3. Test
4. Submit
5. Revise

Let's walk through each part of this process.

Prepare for Your Contribution

There's plenty for you to do even before you can start in on your first contribution. The more you prepare in advance, the more likely it is that your contribution will be well received. There's a reason that *Fail to prepare? Prepare*

to fail is such a commonly used phrase: it's true. The time you invest before you dive into your contribution will pay great dividends later on in the process.

Review the Issue Tracker

If you didn't already do so in the prior chapter, invest some time to review the issue or bug tracker for your chosen project (see *Find a Project* for more information about the issue tracker). It's an amazing resource for learning what a project has done in the past, what it's currently trying to accomplish, what it's looking to do in the future, and just as importantly as all those: what it's decided it doesn't need to do at all.

Regardless of whether the project tags its issues as suitable for a new contributor, reviewing the open issues in its issue tracker can lead you to a number of potential contributions. As you skim the issues, look for those that are interesting in some way. Are bugs reported that have bitten you in the past? Maybe there are issues that were opened but have no activity yet, or issues marked as needing work but are not yet assigned to nor claimed by anyone. Picking up tasks that no one else has had the time to do can be a great way to make your mark in a community.

Set Up Your Environment

Creating and testing your contribution usually requires setting up a testing environment of some sort. Often the project has documentation describing how to do this. Sometimes the documentation is the steps to install the project itself. Regardless, you'll need some way to verify that your contribution actually works or looks the way expected and intended.

Once in a great while, you'll find a project that has provided a container or virtual machine image to give you the ideal testing environment. This is quite rare though; it can take a lot of time and effort to maintain those images, and time/effort are two things in very short supply for most projects.

If the project doesn't provide steps to set up a testing environment or otherwise install the software, ask the community for help. It could be that the steps for this are in a less obvious location. If the project does not have documentation for this, take notes while you're setting up your testing environment. Once you're done, convert these notes into documentation (and also your first contribution to the project). Your efforts will help everyone from that point forward.

This testing environment doesn't only apply to code or technical writing contributions. If you're helping with the project website, its user interface, or

even performing translations, you need to test your changes before sharing them with the community. Figure out what sort of testing environment you need for your specific type of contribution and make sure it works before you lift a finger to start crafting your contribution.

Text Editors

The workhorse of most FOSS contributions is the text editor. Without a good text editor, it's very difficult to make a contribution at all. It may seem like a simple thing, but it turns out which text editor you use can make a big difference when doing software development.

The two text editors you hear about most in free and open source software development are *vi* (vim) and *emacs*. Both are venerable and beloved to the point of a near-religious rivalry between users of each editor. Both also have well-earned reputations for being difficult to learn. I encourage you to become familiar with them at some point, but it's not necessary to learn either to contribute to an open source project. Other text editors will do just fine.

There are many different text editors out there in the world, but not all of them are good for software development. For instance, while it's possible to edit text in Microsoft Word or Microsoft Notepad, neither of these are suitable editors for software development. The text output of these programs contains control characters that can make your code or documentation unreadable by many other programs. They also use different line endings (carriage returns) than many programs expect, which can cause a lot of problems.

A good text editor for software development outputs nothing except Unicode or ASCII characters. Which editors are available depends on your operating system, but some popular ones are Notepad++,[1] Sublime Text,[2] Atom,[3] Kate,[4] and Geany.[5] If you develop on a Windows system, the WordPad application will work well for most text-based contributions like code or documentation.

There's so much talk about text editors in some open source circles that you might wonder whether it's acceptable to use an integrated development environment (IDE) such as Visual Studio Code,[6] Xcode,[7] or Eclipse.[8] The

1. https://notepad-plus-plus.org
2. https://www.sublimetext.com
3. https://atom.io
4. https://kate-editor.org
5. https://www.geany.org
6. https://code.visualstudio.com
7. https://developer.apple.com/xcode/
8. https://www.eclipse.org/home/

answer is: Yes, you can definitely use an IDE to create your free and open source contributions. Just make sure the end product meets the criteria and styleguide for the project. The bottom line is, the best tool for the job is the one that you can use and that generates the output you need. Don't let anyone tell you otherwise. You do you, honey.

Do Issue Triage

Before jumping in and working on a fix for an issue, pause and do some triage first. In a medical sense, triage is reviewing wounds to determine how severe they are. In a technical sense, it's reviewing issues to confirm you understand the problem, can duplicate it, and that it's not already fixed elsewhere. In many projects, issue triage also includes setting a priority for fixing the issue, but that is not usually possible for new contributors to determine, as they lack the big picture view that more experienced contributors have. You will learn a lot more about triaging bugs in *(as yet) unwritten content*.

Doing issue triage takes time up front, but it saves much more time during implementation of the fix for the issue. Triage allows you to confirm that the issue you selected is still, well, an issue. It could be that the problem was resolved in another commit or that the issue itself is just out of date. Triage also allows you to confirm not only whether the issue can be duplicated, but also that you fully understand the requirements of the fix. This understanding leads to more efficient and effective fixes and a much smoother contribution process.

To triage your issue, you must be able to duplicate it or view it in some way. This usually means using that shiny new testing environment you just set up. Review the issue for the steps to reproduce the problem or any other hints about where to look to view the problem itself. If you're able to reproduce the problem, you have a much better chance of understanding what's going on and where to start looking to solve it.

As you triage your issue, document all of your discoveries: the steps to duplicate the issue, what you expected to see, what you actually did see, and any additional requirements (technical or otherwise) that aren't listed in the issue. Add this to a public note in the issue itself. Think of the issue as a lab notebook and you're the scientist seeking a discovery. Documenting everything allows other community members to confirm your work and provide guidance if necessary before you invest a lot of time in crafting a fix for the issue. Documenting your triage notes also helps the next people who look at the issue. Should your triage show that the issue fix is beyond your current skill

or interest level, your notes allow the next person who works on the issue to make progress that much more quickly.

Read the Docs (or Write Them)

No matter what, as you're working with issues, make sure you follow the project's workflow for them. This may be documented (in either the CONTRIBUTING file or elsewhere), but often the workflow is a matter of tribal knowledge. If you find this to be the case, ask a community member for advice and guidance before making a clumsy faux pas with issue handling. Once you have that advice and guidance, write it down for posterity. This important documentation can not only help future contributors, but it can also be your first contribution to the project!

Craft Your Contribution

Once you have some idea whether your contribution is needed for the project, you're ready to create it.

The specifics of how you create it naturally will vary depending on the type of contribution: documentation, user experience, design, code, or other types. Each contribution type obviously will have its own creation process.

Whatever that process is, before you start, do double-check whether the project has already defined some guidelines for it. As covered in *Prepare to Contribute*, many projects provide styleguides and contributor instructions. For instance, if your contribution is code, the project may require it to include both unit and integration tests or perhaps that it pass certain linter rules. If documentation, the project may follow a certain writing styleguide such as AP[9] or IBM, or may require contributions to be written in a specific dialect (British English instead of American English, for instance). Website or graphic design contributions may need to stick to the project's branding guide or to an accessibility style guide.[10] Always double-check the contribution guidelines before you get too far with your work. Doing so may save you a lot of time later on.

Gotchas

If you haven't spent much time in free and open source yet, you may be blindsided by some topics that you wouldn't think should matter, but for

9. https://www.apstylebook.com
10. http://a11y-style-guide.com

various historical and social reasons matter a great deal. Two of those topics are spaces versus tabs, and tab sizes.

Spaces, Tabs, and Tab-size

The controversy between using tab characters or space characters for indenting your (typically code) contribution often catches new contributors off guard.

A single space character looks more or less the same on every screen, in every text editor, and on every platform.

A single tab character, however, can be interpreted and displayed differently by every text editor. Some editors display a tab as eight spaces, others as four. Most good text editors allow the user to define how many spaces to use when displaying a tab character, also known as "set the tab-size". Some text editors allow the user to enter a tab character but then replace it in the text with space characters equivalent to the user's preferred setting.

So why is this such a big deal?

Many projects value consistency in appearance across all editors and platforms. For them, it's very helpful to know editing a file on Windows will present a display similar to editing it on Linux. This prevents surprises when editing files and allows peoples' brains and eyes to learn where to look on the screen for what information. It provides for a visual consistency in the same way a linter or styleguide provides for consistency in content. Projects that prefer this consistency will dictate that contributions use spaces for indentation. They also will dictate the preferred tab-size or indentation size (usually 4 spaces, but 2 and 8 are sometimes found as well; people who use other sizes might be monsters).

Other projects prefer to allow their contributors to control how their display looks. Contributors who would rather work in a more compact editor window can set their tab-size to 2 spaces, for instance, while contributors who desire a larger visual difference between indent levels can set their tab-size to 4 or 8 spaces. Projects that prefer this visual flexibility will dictate that contributions use tabs instead of spaces for indentation.

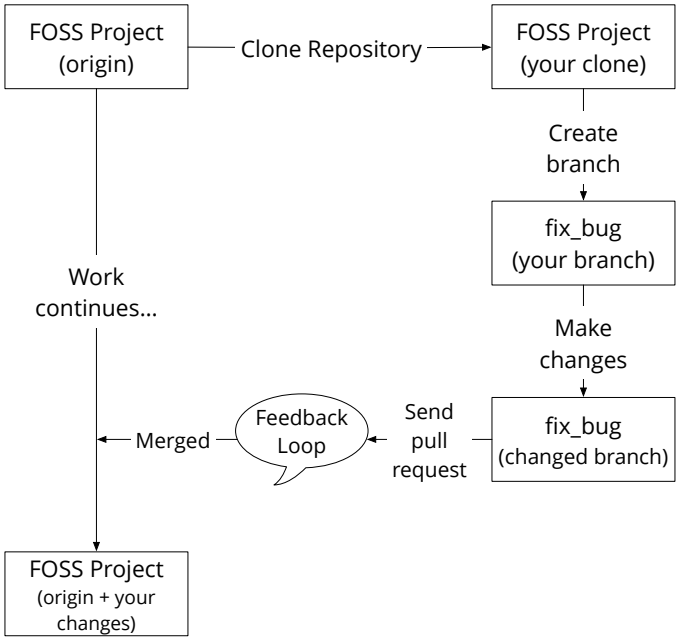
Finally, some projects use programming languages, such as Python,[11] where the whitespace is *significant*: If you indent in one of these languages, that indentation affects the program. And if your indentation is a different size from other peoples' indentations, it can cause a lot of chaos. Paying attention to tabs, spaces, and whitespace is critical to projects that use these programming languages.

11. https://www.python.org

While we all end up having our own preferences between spaces and tabs, when contributing, the only right way to do it is that defined by the project. Even if the project's preferences are not your own, always respect and follow the project's rules. If the project to which you're contributing prefers either spaces or tabs, stick to their preference or risk offending the project community and having your contribution rejected. If the project has not expressed a preference, ask the community about it before starting work. If nothing else, you often can default to indenting with spaces with a tab-size of 4.

Clone and Branch

The first step in any contribution is to retrieve a local copy of the repository (*repo*). In git terminology, this local copy is known as a `clone`, but some hosting services use the term `fork` instead. In the git contribution process, both words refer to the same step, though the two words can mean different things in a FOSS context.[12]



The next step after cloning the repository is to create a branch. When you create a branch, you name it and figuratively plant a flag in the repository to say, "I hereby claim everything from here forward in the name selected for the branch." As long as you stay on that branch, all of your work will be iso-

12. https://opensource.com/article/17/12/fork-clone-difference

lated from every other branch. This allows you to work on multiple different issues at once (by creating multiple branches), but most importantly, it prevents you from sharing changes that you don't want to. In the background, a branch is just a named pointer to a certain git commit, but that's a level of detail that you can read up on later if you want.[13] The important part is that a branch is just a pointer, not a copy of the repository. Therefore, branches in git are cheap, quick, and easy to create and destroy. Easy branches are one of the big advantages of git over earlier version control systems like Subversion or CVS.

A common mistake at this point (and one I've made myself in the past) is to start making changes and working directly on this new copy of the repo. While this can be OK, the best practice is instead to create a new branch of your copy of repository and then perform your work on it. This is called using a *feature* or *topic* branch. Feature branches are just branches of a repository where you perform work on only one thing—one feature—at a time. For instance, if you're working on an issue, you would create a branch just for fixing that issue. Once the issue is complete and the pull request has been accepted, it's no longer needed. You can delete the branch.

Here's an example of a new branch created for this chapter of the book:

```
Pliny:Book brasseur$ git checkout -b makeacontribution
Switched to a new branch 'makeacontribution'
```

Working in this way enables you to work on multiple features or topics at once without contaminating the work for one with the work for another. It allows for a very rigid separation of concerns that prevents committing unneeded or prototype work. It also allows for much easier updates should your pull request require some changes before it can be merged. Simply commit and push new changes to the pull request's feature branch, and they're automatically applied to the request. It's a tidy and efficient process.

While this is currently the most common approach to making a contribution to a FOSS project's repository, it's by no means the only one. Before you start your cloning-branching, always make sure to verify the process against the project's CONTRIBUTING file.

Atomic Commits

OK, so *now* you can start working on your contribution. As you do so, make sure to follow the old adage: *Commit early; commit often.* Tightly scoped—also

13. https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell

known as *atomic*—commits are safer commits. With an atomic commit, you easily can see what you've changed, because your commits are scoped to a single (usually small) topic, feature, or bug fix. This reduces the risk of contributing unnecessary changes. Atomic commits are also much easier to review afterward and to back out should something go wrong. When you make atomic commits, they affect and touch as little of the project as possible, therefore reducing the potential ripple effects of your changes.

Let's get metaphorical: Think of your complete contribution as an essay. It's composed of different paragraphs, each containing a complete thought, but each also requiring the context of the other paragraphs to meet the overall goal of the essay. An atomic commit is like a paragraph: it's a complete thought. Each time you finish a thought, commit it to the repository. If your contribution requires several different steps to complete (rename variables, pull duplicate code into a new function, call the new function in the correct locations), each step should be a separate, small commit. You may end up with several commits before your contribution is complete, but that's OK. It's much better to commit your work at the end of each thought than to risk losing all your work by waiting until the end of the contribution to save it to the repository. Some projects want you to use a *squash* or *rebase* feature in the version control software to consolidate all of those small commits into a single, larger atomic commit, so make sure to read the `CONTRIBUTING` file before submitting your contribution to the project.

Using Version Control for Non-Code Contributions

"But," you ask, "what if my contribution isn't code? Do I have to care about version control systems?"

A very good question! The answer, as you have probably already guessed, is "Yup."

Depending on the project, non-code contributions may not be maintained in the version control system (VCS). Documentation may be in a wiki, for instance. Designs may be in a shared drive system. It could be that you never have to use git, Subversion, Mercurial, or any of the other version control systems that are common across free and open source as well as proprietary software development.

However, considering how helpful it can be for any project to maintain all its related files in a single repository, it's likely that even if your contribution is not code, you'll still have to submit it to the VCS. Documentation, test plans, designs, and all other digital resources can be stored and shared using a version

control system. You can even use one for your own personal writing or design projects. Doing so not only provides off-site backup of these important files, but it also kills off the Frankenstein's Monster file naming schemes, such as logo-new-FINAL-FINAL2-FINALwithedits-FINALapproved-OKreallydonenowhonest.ai. Instead of changing the file name, you simply commit it to the VCS. All previous versions are still there for you to access later if needed.

Even if the project does not use a version control system for non-code contributions, it's still helpful for you to learn about them. You are likely to find that the majority of community members for most projects are programmers. Learning the VCS terminology and how it is used builds empathy with the programmers, which will make it easier for you to communicate with the programmers in the project, and for you to understand the overall software development process. This is particularly helpful if your career path will have you working with programmers in the office.

So while it may not be necessary for you to learn the details of using a version control system for your own contributions, learning at least the basics will make you a more effective contributor and community member.

Test Your Contribution

As you're creating your contribution, make sure you test to confirm that it works at all (let alone does what you think it will). You might laugh, but a lot of highly skilled and experienced contributors have been tripped up by assuming their contribution will work only to learn after submitting it that the contribution is broken or totally wrong. Testing adds time up front but saves it later on. Testing should be continuous throughout the development of your contribution, but is especially important before you submit your contribution to the project.

Regardless of the type of contribution you're creating, test it against the appropriate version of the project to make sure it works as expected. If your contribution is code, provide both unit and integration tests as well as manually testing yourself. If your contribution is documentation or some other type, test how your change will appear in the official documentation repository, website, or wherever it may appear. No matter what, don't assume it's right. Even if it's just a small change, take the time to confirm not only that your change is correct, but also that you haven't accidentally jostled something else on your way.

Many projects have a continuous integration and deployment (CI/CD) service, such as Travis[14] or CircleCI.[15] This service runs all unit, integration, linter, and other tests on all submissions to confirm they meet project standards. If your selected project uses such a service, always pay attention to its results.

It is, by the way, completely OK if your contribution causes CI/CD to fail (*break the build*). This is actually very good news! Your contribution had a problem, but it hasn't been merged, so there's no harm done. You get the opportunity to fix your contribution and to improve in the process. You can learn a lot about a project by reviewing the different ways your contribution breaks the build.

To help others learn from your mistakes, consider documenting the build errors and the things that trigger them. This can be a great aid to new contributors who follow after you.

Diff Your Work

Before you submit your contribution, always do a diff on it. diff is a very old and very useful utility that's now built into most version control systems and IDEs. It simply shows you the differences between two files. In the case of version control, it typically shows you the differences between the files currently in your repository and the most recently committed versions of those files. It's also relatively easy to diff your repository or branch against other repositories, branches, or commits. This means you can see precisely how your branch differs from another branch (even one not on your computer), allowing you to confirm that your contribution will include only the changes necessary to complete your contribution.

You'll find some type of diff functionality in all version control systems and in most IDEs. Many operating systems also provide a diff utility. Check the documentation for your tools to see what options are available and how to use them.

Here's an example of what a diff looks like from the git version control system. In this diff, I changed a setting so the Glossary (included at the end of the book) would be included in the build:

14. https://travis-ci.org
15. https://circleci.com

```
Pliny:Book brasseur$ git diff ffcb48d e590486 jargon.pml
diff --git a/Book/jargon.pml b/Book/jargon.pml
index d0d452c..9a6ae33 100644
--- a/Book/jargon.pml
+++ b/Book/jargon.pml
@@ -1,6 +1,6 @@
 <?xml version="1.0" encoding="UTF-8"?>
 <!-- -*- markdown -*- -->
 <!DOCTYPE appendix SYSTEM "local/xml/markup.dtd">
-<appendix stubout="yes">
+<appendix stubout="no">
        <title>Glossary</title>
```

It looks like there's a lot going on here, but once you get the hang of it, reading diffs can be pretty easy. I asked git to show me the differences between two versions of the file by using their commit hashes: `git diff ffcb48d e590486`. Because these commits included other files, and I only wanted to see the changes in the jargon file, I included its file name (`jargon.pml`) in the diff command. The diff returned a list of lines that changed between those two versions of the file. The line that was in the first version of the file (`ffcb48d`) but changed in the second (`e590486`) is prepended with a - character. The line that was changed or added in the second version of the file is prepended with a + character. Usually, unchanged lines are included on either side to help provide context.

There's obviously more happening in that diff, but these two +/- lines are the most important part. There are options you can pass to the diff command to make it display things differently,[16] but this is the gist of it and pretty much all you need to get started.

Submit Your Contribution

You may have crafted the Best Fix In The World, but it doesn't become a contribution until you actually submit it to the project. So how do you do that?

The contribution submission process is going to vary depending upon your contribution type (document, design, code, or another type) and the requirements and constraints of the project to which you've chosen to contribute.

Read the Docs

Each project will have a different preferred workflow for contributions, so remember to check the `CONTRIBUTING` file (see *Prepare to Contribute*) before trying to submit your contribution. This file probably contains some sort of directions for how to submit a contribution to the project. If it doesn't, ask the commu-

16. https://git-scm.com/docs/git-diff

nity for instructions or guidance. Once you've learned how the process works, share that knowledge with the community by updating the CONTRIBUTING file to help those who follow after you.

Lucky for you, you're looking to submit a contribution in a world where the git version control system is the most common way to contribute. BitBucket, GitLab, and especially GitHub are the reigning champions in the open source repository hosting world, each of them support git, and each of them make contributing a lot easier. While there are other hosting options, you'll find the overwhelming majority of projects on one of these services. This leads to a more or less standard set of processes for contributions, code, or otherwise. Isn't it nice to read that something is somewhat standardized, after all these pages of "every project does it differently"?

Introducing the Pull Request

The primary mechanism for submitting a contribution to these services is called a *pull request*. Some services call it a *merge request*, but this refers to more or less the same process. We'll use "pull request" or "PR" here, since these are what you'll hear people use most often.

The term "pull request" comes from the git commmand request-pull and was popularized in its current form by GitHub. In git, as a distributed version control system, each person can have their own copy of a repository, and each copy could be the source of other copies. One of these repositories is considered canonical. This repository often is called origin or master in the git documentation. To have a change in your version of the repository included in the canonical version, you make a *request* for the maintainers of origin to *pull* your changes into the canonical repository.

The pull request process is very well documented elsewhere,[17] therefore I won't go into it in detail, but I do think it's valuable to spend a few minutes giving you an overview. This will help you know what to expect when the time comes for you to submit your contribution.

Remember that diagram of the contribution process? No? That's OK, you can find it over on page 59. It'll be handy for you to refer back to it during the following explanation.

Starting from the origin in the upper left corner and working clockwise: you clone the repository, create your feature branch, and then make the changes necessary for your contribution. As you're working on your contribution,

17. https://git-scm.com/docs/git-request-pull

other people are continuing to submit and merge other changes into origin and evolving the project. Once you submit your pull request, you enter a feedback loop with community members, working with them to refine your contribution. After you've collaborated with them to put the final shine on it, a community member will pull (merge) your contribution into the project.

Make the Pull Request

Now you're ready to open that pull request to the origin repository. The actual steps for this vary by tool and by repository provider, so make sure to read their instructions before going forward. Whatever provider is used, the process will require some sort of a commit message. Chris Beams has an excellent article[18] detailing not only how to write a good commit message, but also why it's so important to do so. I recommend you read it, but I'll summarize some of the highlights here.

The individual steps may vary, but each process will ask for a description and some, for a title as well. Be descriptive. A title of "fixed stuff" with a blank description is not helpful to anyone. You want to make your pull request as easy as possible for the reviewer to understand. Titles should be brief (50 or so characters if you're using English) and should summarize the contents and intents of the contribution. Descriptions should be as detailed as necessary: don't skimp on words. Descriptions should include not only what you changed, but also why you changed it. If you're working on an issue, the description should reference that issue number. If you format the issue number with a hashtag at the front of it ("#42"), then many issue tracking systems will automatically link the issue with the pull request. This is very handy for contributor and reviewer alike.

An example of a pull request for the fixes on my book repository:

```
TITLE:
Add jargon file to the build

DESCRIPTION:
The jargon file has been commented out of the
build because we were in beta and doing a drip
of one chapter per beta release.

We've finally reached a point where all other
chapters are released, so it's time to include
the jargon file into the build so it can be
released as well.

Flipped the stubout value accordingly.
```

18. https://chris.beams.io/posts/git-commit/

```
Resolves issue #42
```

Before you actually submit your pull request, check the contributions guidelines one more time to make sure you're formatting and submitting your PR in the manner the project prefers. For instance, some projects prefer that you squash all of your commits for your contribution into a single commit.[19] Also, just to be completely sure you're only submitting changes required for this contribution, do one more diff of your work. This is especially important if you were working on multiple branches in your clone. Doing a diff before you send your pull request helps you confirm not only that you're sending the PR from the correct branch, but also that you're sending to the correct branch on the other side.

Patch: The Other Contribution Method

While the pull request process is the most common method for submitting a contribution to a free and open source project, it's not the only one. Free software came into being in 1983. Open source has existed since 1998. Git and the pull request process joined the world in 2005 and didn't become standard operating procedure until GitHub popularized it after the company was founded in 2008.

From 1983 until now, as you can imagine, there have been different processes for contributing to free and open source projects. At least one is in use today, so while I won't go into detail (abundant documentation is available on the internet), it's helpful for you to know about the other option that exists.

The predominant form of contributions prior to the invention of pull requests was *patch files*. A patch is a specialized diff that is dumped to a file and can then be shared to others and applied to a project. The process for creating and applying a patch file varies by version control system and by project. Because patch files were used extensively for so many years, you'll often hear people refer to all contributions as "patches," even if the contribution is submitted as a pull request.

While pull requests are the most common form of submission used today, some projects still rely on patch files for receiving contributions, including the Linux kernel. No matter which version control system your selected project uses (even if it's git), always review the project's contribution guidelines before assuming the submission process. Patch or pull request or passenger pigeon, always know the method for submitting a contribution before you get started.

19. https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History

Review, Revise, Collaborate

Before your contribution is merged (pulled) into the origin repository, someone will review it to confirm that it does what you think it does, that it does something the project needs, and that it conforms to all project styleguides and standards. They probably will have questions, feedback, and suggestions about your contribution. Collaborate with them to get your contribution into an acceptable state. When you apply their feedback, use atomic commits to your feature branch. As you push these changes, they automatically appear on the pull request. This means you won't have to do anything special once you've applied all of the feedback and suggestions. Your contribution can simply be merged into the origin repository.

Congratulations! You've just made your first contribution! High fives all around!

Submit Work in Progress for Early Feedback

The process I just described implies that you should wait until your contribution is complete before sending a pull request to the project, but that's not always the case. Sometimes it can be very helpful to send a *work in progress* pull request while you're still creating your contribution. Just put WIP: at the start of your request title to let the reviewer know that you haven't finished the work quite yet. Also mention in the description that this is a work in progress and include any questions you have.

Why would you submit a pull request before your work is complete? For starters, doing so allows you to receive feedback early in the creation process. This can help you avoid going down some dark, thorny paths. Also, higher quality contributions come from receiving early feedback. The earlier and more often you receive feedback, the more likely it is that your contribution will be of a high quality. Finally, sending a work in progress pull request allows the project to see that someone is working on something, so they won't be surprised when a contribution appears in their pull request queue.

An example of a work in progress pull request:

```
TITLE:
WIP: Testing new section ordering

DESCRIPTION:
I've re-ordered the sections of this chapter in hopes
they'll flow better. WIP PR so Brian can have a look
and let me know what he thinks before I go much further
with the writing.

For issue #40
```

A Note on Feedback

This is a good time to pause and talk about feedback.

Not to put it too lightly, but feedback is great. Without feedback we keep making the same mistakes. Without feedback we can't learn and grow and evolve. It's one of the keys that makes free and open source collaboration work.

Unfortunately, most of us have a very hard time receiving feedback, let alone accepting it. We identify too closely with our contribution, such that criticisms of it—no matter how valid—are taken personally and put us on the defensive.

It doesn't help that most of us also have a hard time giving feedback, often delivering criticisms without empathy or in ways that are directed more at the person than at their contribution.

Both receiving and giving feedback are skills that can be learned and honed through practice. As you enter into this world of free and open source contributions, I encourage you to remember these tips:

- *You are not your contribution.* Even if the person providing the feedback is unskilled at it, and their criticisms come across as personally directed, try not to take their comments in that way. Try to focus on the aspects of their feedback that relate directly to your contribution, then guide the feedback conversation toward these elements.
- *It's not personal.* Problems found with your contribution are not problems found with you. You've put a lot of time and effort into that contribution, so naturally you feel a bit attached to it and that's OK. It's right to feel pride in what you've created and accomplished. But it's better to recognize that there's always a way to improve your contribution. Collaborate with those providing feedback to help evolve the contribution, the project, your knowledge, and your skills.
- *Feedback is a gift.* When people provide feedback on your contribution, they are freely sharing their knowledge and experience with you. You can use this feedback to grow into a more skilled contributor, then one day pay that gift forward as you provide feedback to others. This is part of the beneficial cycle that allows free and open source to grow.
- *Feedback and questions help make you better at what you do.* That's because feedback and questions help you see things you never have before and expand your mind and experiences in ways you never anticipated. None of us are perfect. None of us are all knowing. All of us have been in your position before: feeling excited at the newness but more than a little

lost in it as well. It's OK. Ask questions. Ask for feedback. It's the only way not to feel lost, and we all want to help you.

- *If you get angry at some feedback, step away for a bit to cool off before responding.* It happens: a piece of feedback will get under your skin. Perhaps it was the way it was phrased. Maybe it's dismissing an implementation about which you have very strong opinions. Or maybe the person who gave the feedback is just an indelicate chowderhead. Like I said: it happens. Just because you're angry does not mean you have to react immediately. Replying in the heat of the moment rarely ends well for anyone involved. Take some time to cool off before responding. Go for a walk. Play with your pets or your kids. Spend some time on a hobby or other project. Fire up a good movie or video game. Whatever it takes, give yourself some space from the offending comment. Once you've had the time to cool off and think it over some more, then you can *respond* rather than *react*.
- *Always Assume Good Intent.* Above all, always assume good intent with all feedback. No matter how poorly a piece of feedback may be delivered, the person providing it is still giving you that gift of their knowledge and experience. They're not (usually) doing it to show off; they want the best for the project, for the contribution, and for you. Respect that and them and help them help you provide the best contribution you can. They mean well. Do you?

Tidy Up

Now that the project merged your contribution, you no longer need that feature branch that you created. It won't hurt anything if you leave it lying around, but it doesn't take long for these branches to build up and make a lot of clutter. Deleting it right after your pull request is accepted not only tidies up your testing environment, but it also makes it easier to locate the branches you need later and reduces the chance that you'll accidentally work on this now-dead branch. Removing a branch is quite easy from the command line. Here's an example where I removed a branch from my book repository:

```
Pliny:Book brasseur$ git branch -d makeacontribution
Deleted branch makeacontribution (was 74da8bc).
```

Note: This was a local branch. It's also possible to push a branch to the remote `origin` repository. An example of a command to delete a remote branch:

```
Pliny:Book brasseur$ git push origin --delete makeacontribution
```

Check the documentation for git for further instructions on branch use.[20]

Special Considerations for Windows-based Contributors

Is Microsoft Windows your jam? If so, you should know that you'll unfortunately have a much harder time contributing to most free and open source projects. Due to a couple of decades of Microsoft fear-mongering against open source, the majority of projects evolved such that they don't support Windows at all, not for building, contributing, nor using. While Microsoft has realized the error of its ways and embraced free and open source software, the FOSS legacy of not supporting Windows will take a lot longer to fade away. As you look at free and open source software projects, you'll find that most of them assume you're using a computer running Linux, one of the BSDs, or macOS. It's rare that a project has documented support for Windows and its users.

As a Windows user, you'll likely hit one or more of these common problems when you try to contribute:

- Installation and other scripts are written for shell/bash only.
- Path separators in scripts or the software itself assume Linux/BSD/macOS.
- Differences in the case sensitivities of file names (Linux: case-sensitive; Windows: case-insensitive). This is particularly a problem when using git.
- Windows defaults to a shorter maximum path length than Linux.
- Windows is rarely or poorly supported as a build platform.
- In general, the Windows development tooling is much different from Linux, the BSDs, or macOS.
- Merely being a Windows user or creating on Windows is seen as an imposition to the project.

With effort, you can overcome nearly all of these problems. Thankfully, Microsoft itself has done a lot of work lately specifically to support Windows users who wish to contribute to free and open source projects. Some of the solutions you may need are:

20. https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell

- Use a cross-platform development tool such as Visual Studio Code.[21]
- Leverage the new tools available in the Windows Subsystem for Linux,[22] (WSL).
- Make sure you are using the latest Microsoft build tooling, which is c99/c++14 compliant.
- Use a virtual machine of a Linux or BSD variant.
- Use a container image of a Linux or BSD variant.

These things will help a lot with the technical hurdles to contributing, but what about the social? What if you come across one of those projects where merely being a Windows user makes you a second-class citizen? If all you have is Windows, and you meet resistance for your Windows-based contributions of any sort, there may not be much you can do. Those project maintainers have an operating system prejudice. You can try your best to convince them that your Windows-based contribution is a valuable addition to the project, but you may not be successful. Even if you are, you're unlikely to root out the technical prejudice ingrained in the project. Rather than deplete yourself by fighting a losing battle, consider thanking the project maintainers for their time and then finding a more welcoming and open-minded project and community.

There's More to Contributing Than Just Code

Choosing your contribution, triaging issues, creating and testing your work, submitting pull requests… Now you know exactly what to expect when you submit your first contribution to a free and open source project. Of course the details will vary a bit, but the overall picture will probably look a lot like what you've just read. It may seem like a lot, and you're right: it really is. Despite that, I know you'll get the hang of it. Your contributions may still take a lot of time—some contributions can be quite complex—but the process itself will become smooth. As with all other skills, all you need is practice.

While everyone who wants to participate in FOSS should be familiar with the contribution process in this chapter, it doesn't apply to all of the different types of contributions. What if, for example, you want to contribute but aren't a programmer? What types of contributions can a person make if they don't (or don't want to) code? The answer is: plenty, and the next chapter will detail a few of them.

21. https://code.visualstudio.com
22. https://blogs.msdn.microsoft.com/wsl/

Index

Thank you!

How did you enjoy this book? Please let us know. Take a moment to email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line "Book Feedback."

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2018 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

And thank you for your continued support,

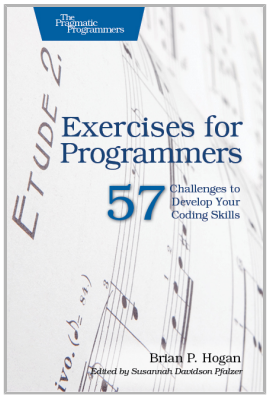
Andy Hunt, Publisher

Level Up

From daily programming to architecture and design, level up your skills starting today.

Exercises for Programmers

When you write software, you need to be at the top of your game. Great programmers practice to keep their skills sharp. Get sharp and stay sharp with more than fifty practice exercises rooted in real-world scenarios. If you're a new programmer, these challenges will help you learn what you need to break into the field, and if you're a seasoned pro, you can use these exercises to learn that hot new language for your next gig.

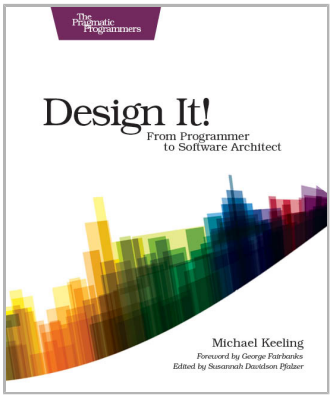


Brian P. Hogan
(118 pages) ISBN: 9781680501223. $24
https://pragprog.com/book/bhwb

Better by Design

From architecture and design to deployment in the harsh realities of the real world, make your software better by design.

Design It!



Don't engineer by coincidence—design it like you mean it! Grounded by fundamentals and filled with practical design methods, this is the perfect introduction to software architecture for programmers who are ready to grow their design skills. Ask the right stakeholders the right questions, explore design options, share your design decisions, and facilitate collaborative workshops that are fast, effective, and fun. Become a better programmer, leader, and designer. Use your new skills to lead your team in implementing software with the right capabilities—and develop awesome software!

Michael Keeling
(358 pages) ISBN: 9781680502091. $41.95
https://pragprog.com/book/mkdsa

Release It! Second Edition

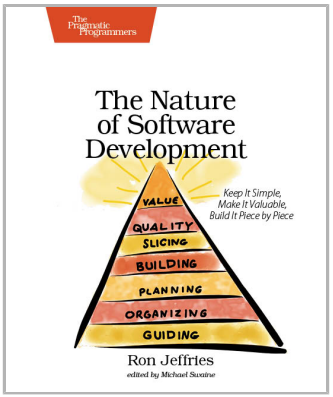
A single dramatic software failure can cost a company millions of dollars—but can be avoided with simple changes to design and architecture. This new edition of the best-selling industry standard shows you how to create systems that run longer, with fewer failures, and recover better when bad things happen. New coverage includes DevOps, microservices, and cloud-native architecture. Stability antipatterns have grown to include systemic problems in large-scale systems. This is a must-have pragmatic guide to engineering for production systems.

Michael Nygard
(376 pages) ISBN: 9781680502398. $47.95
https://pragprog.com/book/mnee2

Pragmatic Programming

We'll show you how to be more pragmatic and effective, for new code and old.

The Nature of Software Development



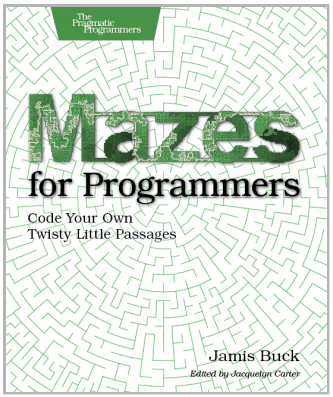
You need to get value from your software project. You need it "free, now, and perfect." We can't get you there, but we can help you get to "cheaper, sooner, and better." This book leads you from the desire for value down to the specific activities that help good Agile projects deliver better software sooner, and at a lower cost. Using simple sketches and a few words, the author invites you to follow his path of learning and understanding from a half century of software development and from his engagement with Agile methods from their very beginning.

Ron Jeffries
(176 pages) ISBN: 9781941222379. $24
https://pragprog.com/book/rjnsd

The Joy of Mazes and Math

Rediscover the joy and fascinating weirdness of mazes and pure mathematics.

Mazes for Programmers



A book on mazes? Seriously?

Yes!

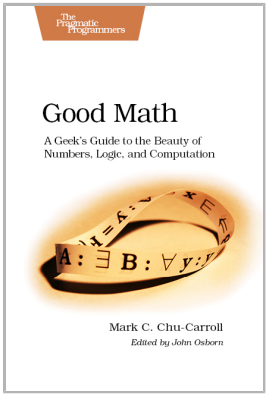
Not because you spend your day creating mazes, or because you particularly like solving mazes.

But because it's fun. Remember when programming used to be fun? This book takes you back to those days when you were starting to program, and you wanted to make your code do things, draw things, and solve puzzles. It's fun because it lets you explore and grow your code, and reminds you how it feels to just think.

Sometimes it feels like you live your life in a maze of twisty little passages, all alike. Now you can code your way out.

Jamis Buck
(286 pages) ISBN: 9781680500554. $38
https://pragprog.com/book/jbmaze

Good Math



Mathematics is beautiful—and it can be fun and exciting as well as practical. *Good Math* is your guide to some of the most intriguing topics from two thousand years of mathematics: from Egyptian fractions to Turing machines; from the real meaning of numbers to proof trees, group symmetry, and mechanical computation. If you've ever wondered what lay beyond the proofs you struggled to complete in high school geometry, or what limits the capabilities of the computer on your desk, this is the book for you.

Mark C. Chu-Carroll
(282 pages) ISBN: 9781937785338. $34
https://pragprog.com/book/mcmath

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

https://pragprog.com/book/vbopens

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up-to-Date

https://pragprog.com

Join our announcement mailing list (low volume) or follow us on Twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

https://pragprog.com/news

Check out the latest Pragmatic developments, new titles, and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*