Client-Centered Software Development

The CO-FOSS Approach

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Client-Centered Software Development

The CO-FOSS Approach

Allen B. Tucker

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-1-138-58384-9 (Hardback)

Library of Congress Cataloging-in-Publication Data

Names: Tucker, Allen B., author.
Title: Client-centered software development : the CO-FOSS approach / Allen B. Tucker.
Description: Boca Raton, FL : CRC Press/Taylor & Francis Group, [2019] | Includes index.
Identifiers: LCCN 2019010378| ISBN 9781138583849 (hardback : acid-free paper) | ISBN 9780429506468 (ebook)
Subjects: LCSH: Application software--Development. | Computer software industry--Customer services. | Consumer satisfaction.
Classification: LCC QA76.76.D47 T839 2019 | DDC 005.3--dc23
LC record available at https://lccn.loc.gov/2019010378

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To Meg,
my inspiration
and lifelong partner

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

List of Figures

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

List of Tables

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Foreword

Client-Centered Software Development: The CO-FOSS Approach provides a much needed guide and resource for undergraduate software development or capstone courses that seek to engage students in a real-world software-development project.

Such a course offers unique and daunting challenges. As someone who has taught such a course intermittently over my 30+ year career, the goal was always to give students a real sense of what software development is like. But the challenges are many. How do you identify a project that can be done and done well in a 14-week semester? How do you manage teams of undergraduate CS majors, with different skill sets and motivations? What combination of platforms and software tools can be used effectively under such constraints? How do you evaluate student effort and contributions? What happens to the "product" once the semester ends? These are just some of the issues.

In this book, Allen Tucker has laid out a well-tested and practical model for addressing these challenges. The development approach is called *CO-FOSS*, which stands for *client-oriented* software development using *free and open source software*. The class project involves developing and deploying a software product for an actual client, which is typically a local non-profit organization that needs mission critical software but cannot afford to hire a professional software-development company. The software platform and tools used in the project are all freely available and openly licensed. The book is full of instructive examples that cover all of the parts and stages of a substantial software-development project. It ends with a practical and innovative model for supporting the student-built product after the semester has ended. This is very important – many of the software-development projects one finds in undergraduate courses end up sitting on shelves.

It's great to see the evolution of the type of FOSS-development course that this book describes. Ten years ago or so, I and other faculty tried to organize such courses under the banner of the *Humanitarian Free and Open Source Software* project (HFOSS). The idea then was to get students involved with existing FOSS projects, particularly those that served "humanitarian" purposes. The goal was to teach students about FOSS development – something that was not typically part of the CS curriculum at the time – by getting them engaged as contributors to some real FOSS projects. We collaborated with the Sahana project (a disaster management system), OpenMRS (a medical records system), GNOME (Linux-based accessibility software), TOR

(privacy-based browser software), the Mozilla project, and others. While we had many successes, and while many students made significant contributions to these projects, the logistics of managing collaboration with such projects within a one-semester course proved difficult. The CO-FOSS model addresses the challenges that the HFOSS approach faced in creative and practical ways. This book shows that you really can get students involved in meaningful FOSS development in a one-semester course.

The book is organized into three main parts. The *Organization Stage* section is written primarily for the instructor and provides practical advice on identifying a client and creating a plan for a doable software product that would help that client, as well as constructing the syllabus for the course. A key part of the syllabus is a carefully thought-out sequence of milestones that, if followed, will lead to successful completion of the project.

The *Development Stage* section is written primarily for students and is meant to be read and followed during the semester. It concisely covers all of the main elements of software development with numerous practical examples: creating development teams, object-oriented design, database design, user-interface design and development, software documentation, and support. Among other things, this section has brief but authoritative discussions of:

- FOSS licensing
- The LAMP, MAMP, and WAMP server stack – i.e., Linux, Apache, MySQL, and PHP
- Software hosting (e.g., GitHub) and issue tracking
- Communication software such as Skype and Slack
- Creating and using unit tests for all parts of the software
- Principles of Model-View-Controller design
- Effective debugging tools and strategies
- IDEs for various programming languages, including PHP, Python, Ruby, and Java
- Database essentials, including normalization and CRUD functions
- Principles of software security
- Writing useful documentation and user-help features

Each of these topics is supported with helpful examples, including many code snippets, taken from successful CO-FOSS projects that Allen and others have conducted at various undergraduate institutions, including Bowdoin College, Dickinson College, University of New Hampshire, Whitman College, and others. Importantly, the projects created at these schools are hosted on

GitHub, and available to be used as models or even templates, depending on the type of software product a client needs.

The *Deployment Stage* section describes how to transition the product from the classroom to professional support so that the product can live on. An important feature of this section is the role played by the *Non-Profit FOSS Institute (NPFI)*, an organization started by Allen that provides help in identifying and supporting professionals who can realistically be expected to host the software and manage its ongoing debugging and support. When no such professionals can be found, NPFI shoulders some of these tasks itself. This is an incredibly powerful resource, which has the potential to make all the difference between a class project that dies once the class ends and a software product that truly adds value to the client's mission.

Some other important features of the book include:

- Milestones: Each of the nine chapters include a short list of milestones. These serve both as a means of keeping the project on track, and also as assignments that can serve as the basis for evaluating student work.
- Course organization: In addition to providing a template that can be used to model the course syllabus, the book provides helpful ideas on how to evaluate student work. Like other parts of the book, these have the benefit of being based on courses that have tried and tested many of the ideas in the book.

Designing and implementing a software-development course in an undergraduate CS program can be intimidating. It exposes the instructor to risks not found in other courses: Will he or she be able to manage the relationship with the client? Will the students be able to create a quality piece of software, and will they see it as an important education experience? Will the instructor receive credit for taking such risks and going beyond the usual course expectations? On this last point, it is worth noting that more and more schools seem to be encouraging "community involvement," and many have set up centers designed to serve as interfaces between town and gown. The model described in this book would fit in well with such institutional initiatives.

This book provides a workable model that helps mitigate some of these worries. The projects used as examples throughout the book serve as a proof of concept for what can be done, and the book itself serves as a step-by-step guide to getting it done. If you are considering an undergraduate software-development course that teaches the principles of FOSS development, you won't go wrong by starting with this book.

Ralph Morelli
Professor of Computer Science (Emeritus)
Trinity College
Hartford, CT
April 20, 2019

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Preface

Software development is a complex and dynamic field. Its complexity appears in many forms – the sheer variety of software clients and applications, the rapid evolution of software development tools, the wide range of skills among professional developers, the rapid evolution of computing platforms, and the diversity of strategies used to develop the software itself.

This book is about one particular strategy for software development called the "CO-FOSS approach." The term CO-FOSS is short for "Client-Oriented Free and Open Source Software." A project using the CO-FOSS approach aims to develop a customized software product for a single client, either from scratch or (more likely) by reusing open source components from prior projects.[1]

The client for a CO-FOSS project is typically a non-profit humanitarian, educational, or public service organization, such as a Ronald McDonald House, a local school system, a Habitat ReStore, a food distribution organization, or a senior center. The key here is that the client has a genuine need for new software that will streamline a mission-critical operation, such as volunteer calendar scheduling, inventory management, donation tracking, or room scheduling.

The CO-FOSS approach has been evolving since 2008. It has been used in intermediate and capstone undergraduate computing courses where teams of students learned the principles of software development while they gained practical experience implementing a real-world software product. The key distinction for CO-FOSS in this setting is that the software product itself is real: the students are developing it for a real client, so both the risks and the rewards are high in comparison with a more traditional software development course with no real product.

Organizing such a course requires an unusual effort by the instructor. Because some of this effort may be unrewarded by typical institutional measures for excellence in teaching, the instructor must view the benefits of taking this "outside the box" approach as worthwhile. Additional support for making this

[1] The term "CO-FOSS" was coined in a 2014 study by MacKellar, Sabin, and Tucker [25], which discusses the results of using this approach in courses at three different types of institutions. The original idea of "client-oriented FOSS" was presented in a 2011 book by Tucker, Morelli, and de Silva [43], where it was contrasted with the idea of "community-oriented FOSS." While both ideas engage students with FOSS development, the latter creates a more generic product that is not customized for a single client.

extra effort may come from the instructor's home institution or from various outside sources such as the Non-Profit FOSS Institute.[2]

Our experiences with the CO-FOSS approach have yielded the following benefits:

1. Undergraduate computing students are uniquely motivated by community service experiences that are embedded within their formal academic training. Uniformly, they report great satisfaction when using their computing skills to develop software that serves the larger community (e.g. the page https://npfi.org/student_evaluations/ shows the complete student evaluations for the software development course taught at Whitman College in 2015).

2. Client organizations benefit by receiving free customized software that directly supports their mission-critical activities. For example, the Ronald McDonald House in Providence, RI received volunteer database and scheduling software called *Homebase* developed by a 5-student team in that 2015 Whitman College course (see https://npfi.org/projects/the-rmhp-homebase-project/). That software is still in use today.

3. The fact that a CO-FOSS product is free and open source software allows any of its source code to be reused and refitted to suit the needs of a future project. For example, the *Homebase* software mentioned above was adapted from an earlier version developed in 2013 by Bowdoin College students for the Ronald McDonald House in Portland, ME. Thus, an open source license like the GNU General Public License or the Mozilla Public License is an essential element of the CO-FOSS approach.

4. Students gain experience learning about key elements of the software development process, including coding, testing, refactoring, and writing user documentation, as they would in a conventional software development course. However, these students also gain practical experience by working within a team, communicating with a client, using a client-centered development model, sharing a code base, and reusing legacy code – experience that prepares them well for entry into the modern software industry.

This book aims to provide instructors, students, clients, and professional software developers with a roadmap to guide them through the development of a new CO-FOSS product from conceptualization to deployment. We use

[2]Throughout this book, any word or phrase that appears in `typewriter font` represents a link to a Web page that provides more details. Of course, those links work only for the e-book version. Readers using the print version should be able to locate most of these Web pages by doing a Google search for that word or phrase.

our own experiences with this approach to illustrate each step in the process, detailing its technical elements, its methodologies, and its outcomes.[3]

The CO-FOSS approach views each project as having three connected elements that form a *triad*, as pictured in Figure 1. The *student team* is one element of the triad, the *client* is the second, and the *professional developer* is the third. The goal of a triad is to design, implement and deploy a customized software product that supports a specific mission-critical activity of the client.[4]

The instructor is involved in all three elements of a CO-FOSS project. The instructor organizes it, leads the student team through project development, and delivers the completed software product to the professional. The students, who are intermediate-level computing majors, develop the software using both the requirements document and a client-centered approach. The professional installs the completed software on the client's server, and then provides ongoing support thereafter.[5]

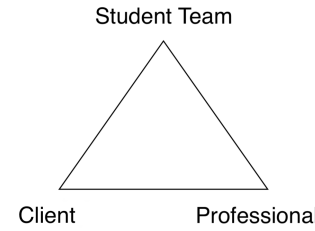

FIGURE 1 The Triad.

The project has three stages: a 2-month *organization stage*, a 3-month *development stage*, and a 1-month *deployment stage*, as shown in Figure 2.

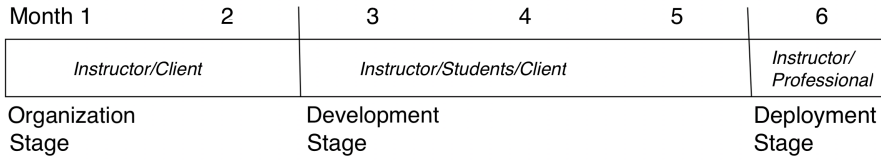

FIGURE 2 The Three Stages of CO-FOSS Development.

So the instructor provides the glue that holds these three stages together, as outlined below:

[3]All the examples in this book use the PHP/Javascript/MySQL/HTML platform, since that is the platform on which our own CO-FOSS projects were built. So while instructors may find the organizational aspects of this book to be useful, this book may be supplemented by reference materials that uses a different platform, such as Django or Rails.

[4]Absent the student team, a CO-FOSS product can always be designed, implemented, and deployed by a professional developer working directly with the client.

[5]Because the requirements and the design document are prepared during the organization stage, the course itself should be properly labeled "software development" rather than "software design."

1. During the *organization stage*, the instructor identifies the client's mission-critical software need, and then develops the requirements for a software product that will fulfill that need. This includes eliciting an initial set of use cases from the client, developing an initial design document and course syllabus that has an implementation timeline embedded, and forming the student team.

2. During the *development stage*, the instructor, student team, and client representative use a client-centered process to create a software product that fulfills the requirements. That is, in a 1-semester course, the team iteratively develops the product and refines the requirements, taking into account the client's feedback at each iteration.[6]

3. During the *deployment stage*, the instructor turns the product over to the developer who installs the product on the client's server (website). Here, the developer and the client also collaborate to iron out any lingering issues for the product and agree on a long-term support plan going forward.[7]

To introduce the CO-FOSS approach, this book has an introductory chapter followed by three Parts. The introductory chapter provides an overview of software, open source licensing, and major software development methodologies. Everyone should read this chapter first and complete **Milestone 1** at the end of the chapter before continuing.

Each Part thereafter explores one of these three stages by sharing our knowledge of designing, developing, and deploying a CO-FOSS product using the *triad* as an organizational framework.

> Part I is written mainly for the instructor and secondarily for the client. It covers the details of finding a client and a CO-FOSS product to be developed, defining that product's requirements, and organizing a course in which students can develop the product. The instructor should complete **Milestones 2** and **3** before continuing to Part II.
>
> Part II comprises the bulk of the book and is written mainly for the instructor and the students. It covers the principles and practice of client-centered software development, with many examples from CO-FOSS projects that our students have completed in recent years. The chapters

[6] We know of several CO-FOSS courses that spread the project's development stage over two semesters, either with the same group of students or with two different groups of students. In one case, two different groups of students worked on the same project in successive offerings of the same course. In another case, the same group of students worked on the project over a unified 2-semester capstone "Software Projects" course. Both of these approaches are viable when the institutional setting allows that flexibility.

[7] The recent rise of Web application hosting services, often called ‘‘platform as a service" or PaaS, may reduce or eliminate the need for a professional developer to be involved in the deployment stage. In this case, the instructor should be willing to maintain the software after the project has been deployed.

in Part II should normally be taken in order, and each chapter's own **Milestone** should be completed before continuing to the next chapter.

Part III is written mainly for the instructor and the professional developer, providing guidance on deploying a new CO-FOSS product, supporting it, and disseminating it to the larger open source community. The last **Milestone** appears at the end of this chapter and its completion signals completion of the entire project.

The Table of Contents shows how the chapters are laid out in each of these three Parts. Of course, the devil is in the details, so let's get started!

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Acknowledgments

The CO-FOSS approach to software development is people-intensive. I am fortunate to have worked with many extraordinary people who have contributed to the CO-FOSS projects described in this book. I gratefully acknowledge:

The student developers at Bowdoin and Whitman Colleges, for their willingness to make the connection between academic work and community service by completing these projects successfully: Adrienne Beebe, Hartley Brody, James Cook, Johnny Coster, Moustafa El Badry, Felix Emiliano, Connor Hargus, Jerrick Hoang, Richardo Hopkins, Noah Jensen, Phuong Le, Alex Lucyk, Dylan Martin, Ruben Martinez, Nolan McNair, Jackson Moniaga, Jesus Navarro, Luis Munguia Orta, Maxwell Palmer, David Phipps, David Quennoz, Oliver Radwan, Sam Roberts, Luis Rojas, Taylor Talmage, Xun Wang, Nicholas Wetzel, Ivy Xing, and Judy Yang.

The non-profit clients, for providing real-world settings in which the software could be developed, customized, and deployed: *The Blue Mountain Action Council of Washington* (Kathy Covey and Jeff Mathias); *Ronald McDonald House Charities of Maine* (Gabrielle Booth, Robin Chibroski, Georgia Doucette, Whitney Linscott, Ashley MacMillan, Alicia Milne, Gretchen Noonan, Karla Prouty, and Raymond Ruby); *Ronald McDonald House Charities of Rhode Island* (Susan Czekalski, Michelle LePage, and Joanna Powers); and *Second Helpings of South Carolina* (Bruce Algar, Lili Coleman, and Jon Peluso).

The professional developers, for supporting the software on the clients' servers: Artopa LLC (David Tripp), Coursevector LLC, The Non-Profit FOSS Institute, Pragmatics, Inc. (Dr. Long Nguyen), and Vivio Technologies, Inc.

My faculty colleagues, for helping me understand the challenges of teaching FOSS development as an academic and humanitarian enterprise: Jim Bowring, Grant Braught, Janet Davis, Greg Hislop, Steve Huss-Lederman, Bonnie MacKellar, Craig McEwen, Ralph Morelli, and Mihaela Sabin.

The reviewers of this manuscript, for providing a wealth of conceptual and detailed suggestions for improving it: Jim Bowring, Janet Davis, and Steve Huss-Lederman.

Dr. Jennifer Tucker, for helping me develop the idea of the Non-Profit FOSS Institute, and then serving as its first Executive Director. And finally my wife, Meg, for her lifelong commitment to education and humanitarian volunteerism, and especially for introducing me to the first CO-FOSS client at the Ronald McDonald House in Portland, ME in 2007.

Allen B. Tucker, February 2019

About the Author

Allen B. Tucker is the Anne T. and Robert M. Bass Professor Emeritus in the Department of Computer Science at Bowdoin College. He held similar positions at Colgate and Georgetown Universities. He is currently a professional software developer and President of the Non-Profit FOSS Institute (NPFI), a 501(c)(3) organization that supports the development of free open source software for non-profits by students and professionals.

Allen earned a BA in mathematics from Wesleyan University and an MS and PhD in computer science from Northwestern University. He is the author or coauthor of several books and articles in the areas of programming languages, software design, natural language processing, and computer science education. He co-authored the 1986 Liberal Arts Model Curriculum in Computer Science, served as Editor-in-Chief of the *Handbook of Computer Science*, and co-authored the textbooks *Programming Languages* and *Software Development*. He also served as Fulbright Lecturer at the Ternopil Academy of National Economy in Ukraine, a visiting Erskine Lecturer at the University of Canterbury in New Zealand, a Visiting Lecturer at ESIGELEC in France, and a Visiting Professor at Whitman College.

Allen has been a member of NSF's CISE Advisory Board, the Association for Computing Machinery (ACM), the IEEE Computer Society, Computer Professionals for Social Responsibility, and the Liberal Arts Computer Science (LACS) Consortium. In 1991, he received the ACM Outstanding Contribution Award and shared the IEEE Meritorious Service Award. He is also a Fellow of the ACM and a recipient of the ACM SIGCSE Award for Outstanding Contributions to Computer Science Education.

From 2008 to 2012, Allen served on the Advisory Board for the NSF CPATH grant that supported Trinity, Wesleyan, and Connecticut College's Humanitarian Free and Open Source Software (HFOSS) initiative. That experience inspired him to begin engaging his own Bowdoin students in HFOSS and developing a curricular model called CO-FOSS (client-oriented FOSS) with his colleague Ralph Morelli at Trinity College.

From 2008 to 2015, he taught several software-development courses at Bowdoin and Whitman Colleges using the CO-FOSS model with different student teams. As a byproduct of this work, he developed strong working relationships with non-profits such as the Ronald McDonald Houses in Maine and Rhode Island, and food distribution organizations in South Carolina and Washington.

In 2013, with the belief that the CO-FOSS model would be viable in a large number of undergraduate settings, Allen co-founded NPFI. NPFI's mission is to spread the development and deployment of open source CO-FOSS products, teaching methods, grants, and other resources to other computing faculty, so that they can engage their own students with real-world HFOSS development for many more non-profit organizations in the future.

CHAPTER 1

The Journey

"Change your opinions, keep your principles;
Change your leaves, keep intact your roots."
—*Victor Hugo*

This chapter provides an overview of software — its nature, its development models, its licensing alternatives, its architectures, and its maturity. Thus it offers a useful perspective within which the development of a new software product can be viewed.

CO-FOSS is a model for developing new software. It is a particularly valuable model, both for learning about the software process and for developing an actual software product for a real client. To provide a broader context, Section 1.2 discusses three different software development models: the serial model, the agile model, and the CO-FOSS model.

Fundamental to CO-FOSS development is the free and open source license that accompanies the software itself. Without such freedom, CO-FOSS development would not be possible. This idea is discussed more carefully in Section 1.3.

A key characteristic of any software product is its underlying architecture, or organization. A coherent architecture is always an essential component of all but the most simple software products. Section 1.4 introduces the client-server family of software architectures that underly the organization of many CO-FOSS products.

Different software products also vary in their maturity. The idea of CO-FOSS applies mainly to the development of new software, often from pre-existing components. However, most software is more mature, having evolved through various levels of maturity over its lifespan. Section 1.5 looks at this larger temporal context in which CO-FOSS development lies.

1.1 SOFTWARE

Simplistically, "software" can be viewed as all the programming in a computer that is not hardware. But the very idea of "software" is a complex one. Even

the software on a single computer exists at two distinct levels – the operating system/network level (think Linux, Windows, MacOS, or Apache Server) and the application level (think Microsoft Office, OpenOffice, the Chrome browser, or the Google Maps application).

Professional software developers have skills that reflect the level and type of software that they develop. For example, Linux and network software developers work at the operating system/network level. Their skills allow them to work with such tools and techniques as C programming and process synchronization.

Other professional software developers work at the application level, such as Web programming or database design, which requires a different set of skills. The application level alone spans a wide range of distinct areas, each of which has its own community of developers. Here's just one taxonomy of software application areas that appears in Wikipedia:

> **Information systems software** supports corporate payroll, accounting, and inventory management, and individual word processing, spreadsheet, and visual presentation needs.
>
> **Entertainment software** includes video games, mobile games, and social networks.
>
> **Educational software** includes course management, survey management, and language learning support.
>
> **Enterprise infrastructure software** includes project management, database systems, document management, and content managed websites.
>
> **Simulation software** simulates social networks, battlefield scenarios, airline flight control, and vehicle driving control.
>
> **Media development software** includes computer graphics and animation, graphic art, image galleries, audio and video editing, and digital music generation.
>
> **Product engineering software** includes compilers, interpreters, virtual machines, computer aided design tools, integrated development environments (IDEs), version control systems (VCS), and debuggers.

How big is the software industry? The number of professionals in this industry is large and growing. A recent study estimated that there were 21 million software developers worldwide in 2016. Of those, nearly 4 million worked in the United States, and they comprised 2.5% of the total US workforce. At the same time, demand for software professionals greatly exceeds supply, creating a favorable job market for new developers who are completing computer science, IT, and computer engineering degree programs.

1.2 SOFTWARE DEVELOPMENT MODELS

Software is also complex in the sense that a software product can be developed using different methodologies, or "development models." On the one hand, it can be developed serially, starting from a fixed set of requirements, proceeding to a design specification, followed by writing the code and finally testing the code. On the other hand, it can be developed from the "bottom up," starting with a small prototype and incrementally adding new requirements and functionality with each iteration.

Additionally, some software can be developed as a generic product for a large (real or imagined) market, while other software can be developed as a customized product for a single client. The former approach is potentially more profitable, while the latter approach is useful for an organization that has unique software needs that are unmet by commercially-available software.

Finally, software can be developed from scratch (sometimes called a "greenfield" project), or it can be developed incrementally using pieces of code borrowed from other software with similar features (a "brownfield project").

This section briefly addresses three different software development models, their constraints, and their tradeoffs.

1.2.1 Serial Development

The serial approach to developing software originated as the so-called "waterfall model," and it was the predominant approach to developing software throughout the 1970s and 1980s. It is based on the assumption that a software product's functional requirements can be fully specified at the outset, and that subsequent stages in the development process can be carried out more-or-less serially. These stages are called "requirements analysis," "design," "coding," "testing," and "delivery."

Each stage in this process is viewed as a single discrete event. One stage typically does not begin until the previous stage is completed. Typically, the client is involved in the beginning and ending stages, but not in the crucial middle stages. This is illustrated in Figure 1.1.

If the requirements can be fully specified at the outset, the serial model can work. For example, an embedded software module that measures and reports the altitude of an airplane in real time can be designed and implemented using this model.

However, this serial approach to software development has had a poor record of success in completing software products for customers. For example, the 2015 Chaos Report [19] surveyed 50,000 software projects around the world to learn how well they met the following three criteria:

1. completed on time,
2. completed on budget, and
3. completed with all features implemented.

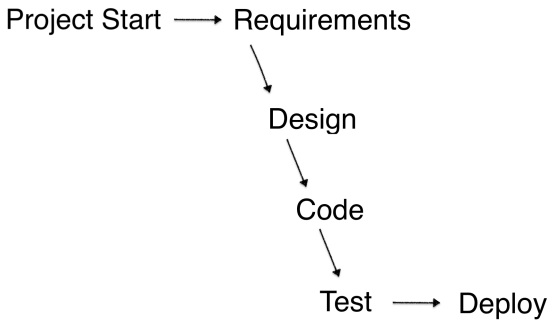

FIGURE 1.1 The serial (waterfall) software development model.

The Chaos Report found that only 11% of all projects using the traditional serial model met all three criteria, while 60% were "challenged" (that is, they were completed but did not meet all three criteria), and the remaining 29% failed (that is, they were never completed).

So in many situations, the serial development model does not work well. Its main problem lies in the assumption that the requirements of a software product can be fully specified at the outset, and that those requirements will not vary throughout the development process. In reality, various outside factors (such as changing user needs or the emergence of a new computing platform) can alter the requirements. For example, the 2015 Chaos Report [19] confirmed that not incorporating end users' feedback throughout the development process was a frequent cause for software project failure.

1.2.2 Agile Development

Since the 1990s, and in response to these problems, software development methodologies have been gradually evolving away from the serial model. Newer methodologies known as "rapid application development," "dynamic systems development," "scrum," "extreme programming," and "feature-driven development" have been shown to be more effective in settings where changing user requirements or computing platforms had become the norm. These newer methodologies all led to the 2001 publication of the *Manifesto for Agile Software Development* [5], which crystallized them into a coherent statement of principles and a development model.

In recent years, the agile model and its variants have yielded significant improvements over the traditional serial model. For example, the same 2015 Chaos Report [19] found that 39% of all projects that used the agile model met all three of the above criteria for success, while 52% were challenged and only 9% failed.

The main reasons for its improved success rate are explained by the nature of the agile process itself. In an agile project, the software product starts with

a minimal set of requirements and iterates several times through a 6-stage development cycle, as pictured in Figure 1.2. The process is fluid, in the sense that each cycle improves the requirements and develops new code in response to client feedback from reviewing the results of the previous cycle.

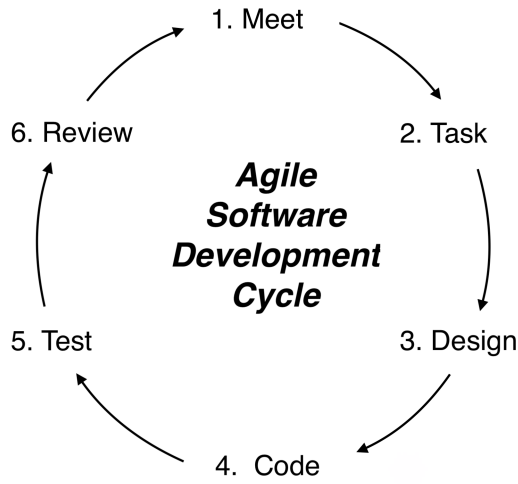

FIGURE 1.2 An agile software development cycle.

Let's look at some of the details in the agile cycle. In stage 1, the developers **Meet** with the client and discuss the client **Review** of the partially-completed software from stage 6 of the previous cycle. In stage 2, the developers and client assume new **Task**s for making progress by adding new functionality and incorporating the client's feedback from stage 1. Developers then independently complete their respective **Design**, **Code**, and **Test** stages, thus preparing the next version of the partially-completed software for client **Review**.

1.2.3 CO-FOSS Development

The CO-FOSS model for developing software is a hybrid of the serial and agile models for software development. It borrows pieces from both, as summarized in Figure 1.3.

To enable students to develop a useful piece of software for a single client in one semester, the software **Design** is organized by the instructor and the client before the semester begins, which is reminiscent of the serial model. This activity is described in detail in Chapters 2 and 3.

Then the software **Development** is completed by students and the client through a series of meet-task-code-test-review cycles. Each 1-2-week cycle is repeated 5 or 6 times throughout the semester, each repetition achieving a pre-determined milestone that ensures successful project completion.

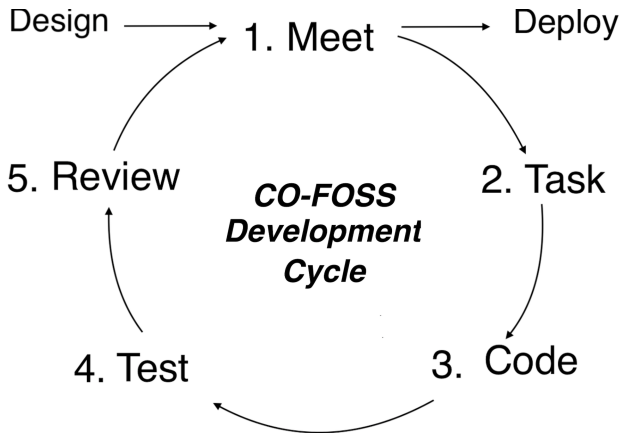

FIGURE 1.3 The CO-FOSS software development model.

The **Code** stage relies on the fact that developers work with free and open source software (FOSS). In the FOSS world, mature well-tested code can be freely downloaded for reuse in any application with a similar functional need. Thus, developers need to read and work with code written by others. Real software is less often developed from scratch by a single individual. Instead, it is usually developed incrementally by a team, each member adding and refining parts of an existing "code base" written by others.

The **Test** stage in each iteration of the cycle provides a new opportunity for *debugging* and *refactoring* the code base in preparation for client **Review** and the next iteration.

> *Debugging* means finding and correcting errors in the program. Bugs, or instances of incorrect behavior, result from programming errors. Such errors can often be notoriously difficult to find and correct, even when working with a small code base. So as an aid to finding bugs we use an aggressive strategy called "unit testing," where individual units (classes and modules) of code are individually tested at each repetition of a CO-FOSS cycle.

> *Refactoring* a program means reading the code, finding instances of poor programming practice (from either a readability or an efficiency standpoint), and reorganizing the code so that it performs the same functions in a more readable and/or efficient way.

Coding, testing, debugging, and refactoring are discussed in detail in Chapters 5, 6, and 7.

Clients further test and evaluate the software during the **Review** stage of each cycle. They play a key part in debugging, since they are the ones who most often identify bugs and provide feedback to developers during each cycle, ensuring that the final product meets their particular expectations. A more

careful treatment of the client review process and its close relationship to the Test stage appears in Chapters 5, 6, and 7.

Finally, **Deployment** of the CO-FOSS product takes place at the end of the semester, and is coordinated between the instructor, the client, and a professional software developer. This is described in detail in Chapter 9.

1.2.4 Software Customization: A Continuum

A final consideration for software developers and clients involves the alternatives that are available in selecting/developing a software product to help improve a particular mission-critical activity within the organization. These choices form a continuum – from developing a completely customized software product to obtaining a completely off-the-shelf product, with many other choices in between. Let's take a look at the trade-offs among three key choices in this continuum:

Custom software,

Off-the-shelf software, and

Custom software with off-the-shelf components.

Custom Software

Custom software is just what its name suggests. The developer designs and implements a unique piece of software that can improve a client's mission-critical activity. The software is tailored to match all that activity's needs, processes, and security requirements. Importantly, the client's staff can assimilate that software easily because it uses existing organizational vernacular that the staff already knows.

Custom software may be *open source* or *proprietary* (see Section 1.3), but the client must rely on the developer to keep it up to date with changing organizational needs. So a strong working relationship between the developer and client is essential for custom software to remain effective. Custom open source software is ideally *client-centered*, allowing the client to be involved continuously in the development process.

Custom software is not without its downsides. First, its original development cost can be higher than the alternatives, if there are any. Second, asking the developer to add new features as requirements change may also be billable. Third, custom software has no peer user community outside the client's organization to provide advice on usability issues, though this downside is somewhat mitigated by the developer's ongoing availability.

All the software projects discussed in this book have developed custom open source products. Each one is fitted to satisfy the requirements of a single customer. For example, *Homebase* was originally developed in 2008 for the Ronald McDonald House in Portland, ME. Enhancements were made by different student teams in 2012 and 2013. A single developer returned in 2015 to add

more features to *Homebase*. These results would not have occurred if *Homebase* were not *open source* and developed using a *client-centered* approach.

When weighing whether to use custom software, an organization should be sure that there is no satisfactory off-the-shelf product available that can satisfy its requirements at an affordable cost. It should also find a developer that can produce that custom software and provide ongoing support, all at an affordable cost. For example, all the software products discussed in this book were developed and are supported at no cost to their clients. However, since each of these products was developed using the CO-FOSS model, it did require client participation (averaging about 2 person-hours per week) throughout its development process.

Off-the-Shelf Software

Off-the-shelf software is a (proprietary or open source) product developed for a large number of customers. Examples include Microsoft Word, Apache OpenOffice, Google Sheets, and various smartphone- and tablet-based computer games. Off-the-shelf software is aimed at addressing a specific shared need of a mass market audience, such as the need to play a game of Sudoku on a smartphone while waiting for an airplane.

The per-user cost of off-the-shelf software can vary greatly; some products are free, others are costly, and still others are available as both free "introductory" versions and paid "full" versions. The full versions of off-the-shelf software usually come with a preponderance of features, most of which are not needed by the average user. These features are there in order to satisfy the one-size-fits-all requirement. However, their presence can make the software more difficult to learn and use.

Off-the-shelf software can be deployed quickly, usually with a simple download and install step. Another advantage of popular off-the-shelf software products is that they have large, often international, communities of users and forums that provide self-help support. So the user doesn't need to hire a developer to fix a bug or customize the software to fulfill a specific need.

On the downside, off-the-shelf software typically will not match all of an organization's needs, either lacking needed features or providing superfluous features. If customization is even possible, that typically comes at an additional cost. Routine upgrades may also come with additional costs.

Finally, off-the-shelf software can be obsolete or slow to evolve with the industry to which it is targeted. Moreover, its vernacular is invariably out of sync with the user organization's vernacular, requiring users to assimilate a new vocabulary before becoming comfortable with the software. Off-the-shelf software may also require technologies that do not conform to the organization's current computing platform. Moreover, it often comes with the subtle inability for an organization to change to a different vendor in the future when a better alternative emerges (this is sometimes called "vendor lock-in").

Custom Software with Off-the-Shelf Components

There is a middle ground between custom software and off-the-shelf software, which is becoming an increasingly popular solution for organizations. The idea of "custom software with off-the-shelf components" is that an organization finds software that matches most of its specific needs but requires a few additional functions in order to match the rest. Typically, this approach uses open source software at its core, though some of the add-on components can be proprietary as well.

A good example is `Wordpress`, which is free open source software out of the box for building websites. A Wordpress website can be customized by adding "plugins" which are modules that provide specialized functionality so that the website can provide specific functionality that matches an organization's peculiar requirements. The Wordpress plugin library is huge, and it covers a wide range of functionalities, such as membership management, on-line application form processing, and on-line product catalogs for e-commerce.

The advantages of this approach to software development are mainly that it leverages pre-existing libraries of reliable modules to help reduce up-front costs, especially those associated with writing and testing new code. Other advantages are derived from its basic open source nature: the organization owns the software and its attendant database (avoiding vendor lock-in), the software can be continuously updated to meet changing needs, and there are no licensing fees.

The disadvantages of this approach are higher upfront costs vs. off-the-shelf software and the requirement for an ongoing relationship with a developer to make changes and upgrades (which may be billable). Like custom software, this approach has no attendant user community to provide self-help (though the relationship with the developer compensates for this).

1.3 SOFTWARE LICENSING

A software product can be licensed in one of two general ways, *proprietary* or *open source*. The differences between these two types of licenses are significant, especially in regard to the software development process and environment in which the software is created and maintained.

1.3.1 Proprietary Licensing

Proprietary software is that which is licensed and sold as a binary executable program to individual and corporate customers. The source code is the private property of the developer and is kept hidden from the customer. A proprietary software license typically limits the number of computers on which a user can install the software – installing the software on more than one computer costs more money. So a proprietary license prevents the user from copying the software, modifying it, or sharing it freely with associates and friends.

From the 1970s to the mid-1980s, nearly all software was developed and sold with a proprietary license. Proprietary software is developed and maintained by an in-house programming staff of a large organization or by a vendor targeting a specific market. All developers of a proprietary software product must sign a non-disclosure agreement (known informally as an NDA) which binds them to secrecy about the product's source code and architecture.

For example, Word was developed by Microsoft's own programmers to meet the needs of the word processing market. Today it can be bought by a single user either stand-alone (for $110) or as part of Microsoft's "Office 365" bundle, a cloud-based subscription service that includes Word, Excel, PowerPoint, OneNote, Outlook, Publisher, Access, OneDrive, and Skype (for a $70 yearly subscription). The license for a single-user version of Word is a 30-page document "Microsoft Software License Terms," which spells out that the user has the right to install and use a single copy of the software on a single computer, but cannot copy it to a second computer or pass it to a friend.

Google Docs is a proprietary word processor that runs on a web server and is a free alternative to Microsoft Word. While Google Docs is less feature-rich than Word, many users prefer that because of its intuitive functionality and its interoperability with other aspects of cloud computing. Most importantly, Google Docs' cloud-based functionality supports smooth simultaneous editing of a shared document by several persons. Microsoft's cloud-based version of Word, when it is bundled within Office 365, also supports this kind of collaboration.

1.3.2 Open Source Licensing

Free and open source software (FOSS) is software whose source code and binary executable code are freely available for download by any individual or organization. Most significantly, "freely" means that downloaders are free to use the software on any computer, to modify the source code and binary, and to share the modified software with associates and friends. Because of this freedom, FOSS is accessible in markets where proprietary software has no interest and little leverage—non-profit organizations, developing countries, and individuals and businesses who are either unwilling or unable to pay the cost of purchasing proprietary software.

Most proprietary software has a FOSS alternative. For example, a FOSS alternative to Microsoft Word is called "Writer" and is part of the "OpenOffice" bundle, maintained and distributed by the Apache Foundation. OpenOffice allows any individual or organization to freely download and use it on any number of computers. It runs on Windows, Linux, and Macintosh platforms. OpenOffice is distributed under an open source license called the "Apache License Version 2.0," which describes the rights of clients to download and freely use, copy, modify, and redistribute this software. The Android operating system also carries the Apache license [24].

Besides the Apache License, three slightly different types of licenses are used for FOSS products:

> `The MIT License` [27] was developed by the Open Source Initiative to provide a totally unrestricted vehicle for reworking and redistributing the source code.
>
> `The GNU General Public License` [14], or GPL for short, was developed by the GNU Foundation to provide a vehicle for reworking and redistributing the source code, but with the caveat that any redistribution must be GPL-licensed FOSS as well. This caveat effectively keeps all derivatives of the product in the FOSS domain for other developers to freely use and refine.
>
> `The Mozilla Public License`, or MPL for short, was developed by the Mozilla Foundation for its Firefox browser and is used by many other software products today.

Many popular FOSS products (Linux and Wordpress, for example) are licensed under the GPL, preventing them from ever being commercialized or embedded inside a proprietary product. Version 2 of the GPL was released in 1991. The GPL has been repeatedly upheld in courts around the world as an enforceable license [2]. Since 1991, a variety of FOSS licenses have evolved alongside the GPL to satisfy different needs within the open source community. GPL version 3 (GPLv3) was released in 2007 to address a wide range of issues, especially its compatibility with these other FOSS licenses.

Unlike the GPL, neither the Apache License nor the MIT license protects a software product from having one of its derivative products converted into a proprietary product and sold for profit. For example, Apple's MacOS operating system is proprietary software derived from the FOSS product BSD Unix, which carries an MIT-like (permissive) license.

The LGPL and MPL represent a middle ground between the permissive MIT license and the protective GPL license. That is, while they protect the FOSS software and its derivatives from becoming fully proprietary, they allow the software to be embedded in a larger proprietary product.

Today, there are dozens of different FOSS licenses. The Free Software Foundation's own list of other licenses cites rulings on which ones are compatible with the GPL as well as guidance on how to define customized FOSS licenses [15]. The Open Source Institute maintains a similar list as part of its effort to define *open source software* [27].

One of the most difficult questions for FOSS developers is how the various licenses relate to each other. Figure 1.4 [44] provides an overview of the more widely used licenses and their inter-relationships. Each box represents a particular kind of license.

A license is more or less *protective* depending on how strongly it protects the freedoms listed in the FOSS definition, particularly the freedom to redistribute derivatives of the software. The left-to-right arrangement of the

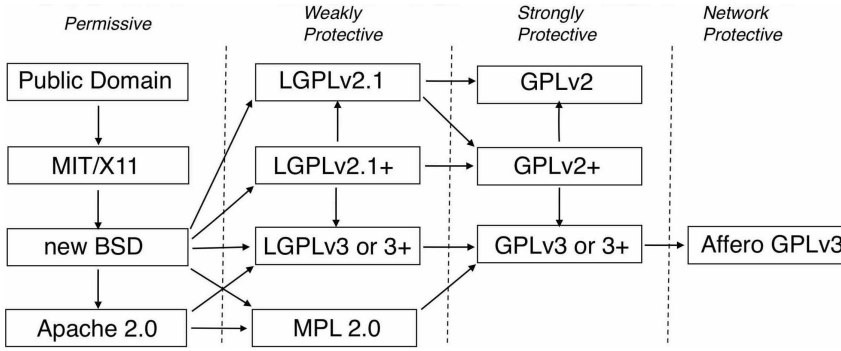

FIGURE 1.4 Relationships among common FOSS licenses.

licenses in Figure 1.4 shows how they progress from least to most protective in this sense. Here are the main distinctions among the columns in Figure 1.4:

Permissive Software in the *public domain* is, strictly speaking, unlicensed and therefore completely unprotected. Thus, someone can take a piece of public domain software and re-distribute it under another open source or proprietary license. The other three licenses in the *permissive* column also allow derivative products to become *proprietary*. For example, software with an Apache license can be turned into a proprietary product.

Weakly protective *Weakly protective licenses* are often used for source code libraries or modules. They protect the software from becoming proprietary but allow it to be used as part of a larger proprietary package. The Lesser General Public License (LGPL) and the Mozilla Public License (MPL 2.0) are the most widely used licenses of this type.

Strongly protective Licenses in the *strongly protective* column in Figure 1.4 require that derivative works must also be licensed, as a whole, under the GPL. This effectively prevents derivatives from becoming proprietary software.

Network protective The Affero GPLv3 expands the reach of the GPL so that users of a Web application can receive its source code. This also applies to network-interactive software, including programs like game servers.

The arrows in this figure represent *compatibility*, indicating where two different FOSS-licensed products can be merged into one and share a common license. To determine this sort of *compatibility* between two licenses, we trace the arrows from the two products' licenses in Figure 1.4 to a common license. For example, two software products distributed under an Apache 2.0 license and an MPL 2.0 license, respectively, can be combined into a new product and distributed under a GPLv3 license.

Licensing a complex FOSS product is sometimes a complex matter. For example, one of the most challenging issues Mozilla faced was to respect the prior licensing of Firefox's many embedded third-party modules. Agreements with the owners of these modules had to be worked out so that their code could be shipped either as open source or as binary code. The alternative would be to remove these modules from the code base altogether.

Finally, we note that all software licenses, whether they be proprietary or open source, carry some sort of warning notice that the software is provided in "as-is" condition, thus attempting to free the developer from being sued should the software cause harm or inconvenience to a client who downloads and uses it. There are many exceptions to the force of this disclaimer, such as a 2017 lawsuit (see https://npfi.org/iphone-lawsuit/) filed by iPhone users against Apple for intentionally slowing their iPhone software as it got older.

1.3.3 FOSS Origins and Impact

The free software movement was started in the early 1980s by Richard Stallman [38] and his colleagues. Stallman was a programmer at MIT's Artificial Intelligence lab and learned to program as part of the hacker culture that was thriving in much of the programming community during the 1960s and 1970s.

Having grown frustrated with the directions that the proprietary software industry was taking, Stallman started the GNU (read "GNU is Not Unix") project in 1983. This project was an effort to build an entirely free and open operating system [36]. It is clear from Stallman's original announcement about GNU that his motivations were ethical and humanitarian [18]:

> I consider that the golden rule requires that if I like a program I must share it with other people who like it. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. ... I'm looking for people for whom knowing they are helping humanity is as important as money.

In 1985 Stallman founded the Free Software Foundation to help support this new movement. He developed the definition of free software along with the concept of *copyleft*, which uses software licensing to protect the freedom of software users and developers to share their work [5]. Under a copyleft license, *free software* guarantees users the freedom to:

1. Run the software for any purpose,
2. Study and modify the software (which requires access to the source code),
3. Distribute copies of the software to help their neighbors, and/or
4. Improve the software and release those improvements to the public so that the whole community benefits [5].

Notice that these four freedoms imply "open source" as well, especially considering items 2 and 4. So using the term *free software* as defined here is equivalent to using the term *free and open source software*, or FOSS.

Despite the ambiguity of the English word "free," Stallman's definition of *free software* has nothing to do with the *price* of the software; in his own words, it means "free as in 'free speech' not as in 'free beer.'" As a byproduct, however, most software that is licensed under this definition is also distributed free of charge.

In 1989, to help protect programs developed as part of the GNU project, Stallman created the GNU General Public License [14]. The GPL is widely regarded as the strongest copyleft license, since it requires that all derivative works be made available under the four freedoms listed in the above definition of copyleft licensing.

By 1991, Stallman and his collaborators had developed an entire UNIX-based operating system, minus the kernel program. It was in this context that Linus Torvalds, working with a broad international community of programmers, developed the Linux kernel program [3]. Linux became licensed under the GPL and became the core of the GNU/Linux operating system [42]. GNU/Linux, or Linux as it is popularly called, is one of the best and most widely known examples of FOSS.

Following the dramatic success of Linux, the Open Source Initiative (OSI) was founded with the purpose of making the FOSS development process acceptable to the software industry itself [29]. In his formulation of the *open source definition* Bruce Perens and other founders hewed closely to Stallman's principles, preserving the basic freedoms that Stallman articulated. Despite this effort, for many the OSI provided a means to distance the movement from what they saw as Stallman's anti-business stance. As a result the OSI has focused more on the practical benefits of the FOSS development model.

As open source gained popularity within the software industry, a schism developed between free software and open source proponents. Perens eventually resigned from OSI [28], saying:

> Most hackers know that Free Software and Open Source are just two words for the same thing. Unfortunately, though, Open Source has de-emphasized the importance of the [four] freedoms involved in Free Software. It's time for us to fix that. We must make it clear to the world that those freedoms are still important, and that software such as Linux would not be around without them.

However, despite the efforts of Perens and others to emphasize the moral dimension, the gap between the two branches of FOSS continued to grow. Stallman himself has continued to emphasize the moral motivation behind the free software movement and has repeatedly emphasized the fact that it is the commitment to software freedom, not the temporary practical advantages, that make the FOSS movement viable [37]. In July 2009 Stallman was

still encouraging the FOSS community to place its emphasis on software freedom [39]:

> As the advocates of open source draw new users into our community, we free software activists must work even more to bring the issue of freedom to those new users' attention. We have to say, "It's free software and it gives you freedom!" more and louder than ever. Every time you say *free software*, rather than *open source*, you help our campaign.

In recent years, the FOSS movement has been highly successful and has grown to encompass a significant share of the software market. Two important events came together to contribute to this success, the emergence of the Red Hat business model and the transformation of the Netscape browser into Mozilla Firefox.

The Red Hat Business Model Red Hat Corporation was the first to show that FOSS development can be sustained by an effective business and economic model [45]. In 1993, prior to Linux's surge in popularity, Red Hat's founder, Robert Young, was running a software distribution company specializing in Unix applications. As sales of Linux distributions began to pick up, he and Marc Ewing founded Red Hat Software, Inc. in January 1995.

Red Hat's business model is to work with Linux development teams from around the world to put together the hundreds of modules that make up a Linux (or, more accurately, a GNU/Linux) distribution. Rather than selling a license for the software, as a proprietary software vendor would do, Red Hat sells service. In the 1990s, selling service, rather than branding the software as *intellectual property* and selling it, was a revolutionary concept. The Red Hat model thus provides convenience, quality, security, and service to its customers. By 2017, Red Hat Linux had gained a 67% share of the Linux market [34].

Following in Red Hat's footsteps, many other companies have discovered that rather than owning a proprietary software product, a successful business can be built around the concept of supporting and servicing a FOSS product. This fact contradicts the programmers-need-to-eat skepticism that had greeted the GNU Manifesto when it appeared in 1983[1]. That is, the GNU Manifesto defended open source as an idea that is compatible with that of financial viability in the software industry, and the Red Hat model independently verified that idea.

From Netscape to Firefox The creation of the Mozilla community was another watershed event in the history of the open source movement.

[1] Stallman originally wrote the GNU Manifesto [36] to help gain financial support for the development of the GNU operating system.

Unlike its successful FOSS predecessors (e.g., Linux and Apache) that mainly benefit professional programmers, the Firefox browser became the first FOSS product to be successfully distributed to all computer users. Here's how Firefox came into being.

In 1994, Netscape began providing unrestricted distributions of its Navigator browser. In January 1998 Netscape announced that, in addition to freely distributing its browser, it would also freely distribute the source code for its browser software, known as *Mozilla* [20]. Thus, Netscape became the first large corporation to open-source its proprietary software in the interest of widening corporate development of open source environments. This event forever changed the way software is distributed on the Internet.

Mozilla's current open source bowser, called *Firefox*, has become a major combatant in the so-called "Browser Wars," alongside Google's Chrome, Microsoft's Internet Explorer, and Apple's Safari. Of these, only Chrome and Firefox are open source browsers and they combined to command 72% of the desktop browser market in 2017. The proprietary software alternatives, Internet Explorer and Safari, command only 19% of the same market [33].

Today, the Mozilla Foundation (mozilla.org), originally formed to manage the Mozilla development effort, has evolved to become a model open source community. Mozilla has only about 300 paid employees. Another 1,500 or so volunteer programmers from a broad international community contribute to its most recent software releases. In addition to programmers and developers, the Mozilla community includes tens of thousands of testers and users, who work to promote the browser and have helped to translate it into more than 70 languages worldwide (see [16] for more details).

As Red Hat, Mozilla, and many other open source projects have demonstrated, the FOSS development model is compatible with the idea of commercial success in the software business. Today, many major software companies, including IBM, Google, Hewlett-Packard, and others, support open source development in various and substantial ways. Companies that rely on the success of systems such as Linux and Apache assign members of their own software development staffs to work, more or less full time, as contributors to these projects.

FOSS Worldwide

At this writing, the FOSS movement has spread far beyond its origins in GNU, Linux, and Mozilla. Many new FOSS communities have emerged to develop important consumer-related software products. Because of its accessibility, affordability, transparency, and association with freedom for the user, FOSS

has become a major force in the software industry. Here are three notable examples:

GIMP The GNU Image Manipulation Program (GIMP) provides a software suite for photographic and other image manipulation. It is a free alternative to the proprietary Adobe Photoshop software.

OpenOffice OpenOffice is an office productivity suite that includes word processing, spreadsheets, and presentation modules. As such, it is a free alternative to the proprietary Microsoft Office software, which includes Word, Excel, and PowerPoint.

Wordpress Wordpress is an open source platform for developing content management systems (CMS) in websites. As of 2017, Wordpress had a commanding lead in CMS market share over Drupal and Joomla, which are also open source platforms.

Businesses, governments, academic institutions, and non-profit organizations throughout the world are increasingly turning to FOSS for their software needs. Here are a few examples:

> The 5th consecutive International Conference on Open Innovation 2.0 took place in Romania in June 2017. It is attended by innovation experts, policy-makers, academic scholars, practitioners and individuals who are engaged in various aspects of open source development. The 2017 conference site has a strong IT and innovation ecosystem with 11 Universities and several innovation and technology parks [11].
>
> The Brazilian government was one of the first to experiment with FOSS, beginning shortly after the election of Luiz Inacio Lula da Silva in 2002. Following Brazil's leadership, the entire Latin American Region started initiatives to promote FOSS usage and development, including many grass roots efforts. In April of each year since 2005, free software festivals are held in 200 cities and 18 Latin American countries [12].
>
> The French Gendarmerie is reported to have saved an estimated 50 million Euros since 2004 in moving from Microsoft to the Ubuntu/Linux desktop [22].
>
> In 2009, the Amsterdam city government made OpenOffice and Firefox their default systems on all its desktops [21].
>
> Also in 2009, the United Kingdom government announced an effort to avoid vendor lock-in by considering FOSS alternatives equally when deciding IT procurements [7].
>
> Throughout the last several years, U.S. government organizations have made major commitments to FOSS, including the Library of Congress,

the U.S. Postal Service, the U.S. Census Bureau, the Department of Defense, the FBI, and many state governments (see [32] p. 182 and [30]).

In addition, some of the most dramatic FOSS progress has occurred in developing nations, where governments have seen FOSS as a way to save money, avoid the vendor lock-in problem, and bridge the technology gap.

The emergence of FOSS is fueled by many forces, including the world's need for affordable computing, the effectiveness of agile and related software development methodologies (see Section 1.2), and the increasing worldwide sense of public ownership of the Internet and its resources. A 2015 survey estimated that 78% of all companies were using open source software, up from about 42% five years earlier [40]. According to that survey, the two main reasons cited by companies for preferring open source over proprietary software are:

1. Open source delivers better security than proprietary software.
2. Open source scales better and is easier to deploy than proprietary software.[2]

Considering its current momentum, popularity, and openness to self-forming development communities, the free and open source software movement promises to remain a healthy and prominent part of the software industry for the foreseeable future.

Terminology: OSS, FOSS, FLOSS, H/FOSS, and CO-FOSS

Overall, the terms OSS and FOSS cover the broad scope of open source software. These two have somewhat different licensing variations, from more permissive (OSS) to less permissive (FOSS) restrictions on reuse. But three other nearly-equivalent terms – FLOSS, H/FOSS, and CO-FOSS – have also come into use. Let's quickly sort them out.

FLOSS was coined in 2001 as an acronym for "free/libre and open-source software" [13]. Proponents of this term point out that parts can be translated into other languages, for example the "F" representing free (English) or frei (German), and the "L" representing libre (Spanish or French), livre (Portuguese), or libero (Italian), and so on. So FLOSS is essentially equivalent to FOSS.

A significant sub-concept within the FOSS umbrella is called "Humanitarian FOSS" (H/FOSS for short), which is open source software designed for use by global relief organizations, non-profit organizations, and society at large.

[2] In this book, you will explore some of the security challenges and solutions associated with FOSS development. You will also gain hands-on experience with the complete process of open source development and deployment. So in the end, your own experiences will enable you to corroborate these two particular survey results.

Proprietary software developers largely ignore the particular needs of these organizations, since they usually lack technology budgets. Excellent examples of H/FOSS development can be found at http://hfoss.org/. [31].

CO-FOSS is that subset of H/FOSS in which the client is a single organization and the software is customized to fit the needs of that client [25]. Whether or not the software has broader uses beyond that single client is not an immediate concern to a CO-FOSS project. In fact, any temptation to prematurely broaden the reach of a CO-FOSS project beyond the specific needs of its client tends to work against the goal of project success. This broadening is sometimes called "mission creep."

1.4 SOFTWARE ARCHITECTURES

Software is the code that resides in a computer and enables a person to use it in a way that facilitates their work or enhances their lifestyle.

The computer may be a laptop or desktop device, a smartphone or tablet, or a server running a complex collection of applications for a large organization.

1.4.1 Software Frameworks

Stand-alone computing is characterized by the picture shown in Figure 1.5. That is, a single computer runs the software for the user working in isolation from any outside services. An example is a person preparing a document using word processing software on a single computer and then printing it on a printer directly connected to that computer. No network services are required to complete this task.

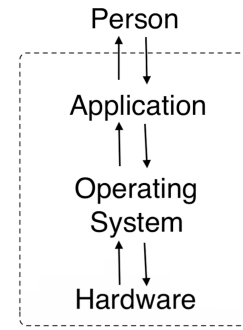

FIGURE 1.5 Stand-Alone Computing.

However, today's computer users generally accomplish their computing tasks by acquiring software services through an Internet connection, provided either by a local wi-fi signal or by a cell tower. When computers connect directly to software in this way, they are using a "client-server" framework, as shown in Figure 1.6. Here, each application is split into a so-called "client side" and a "server side," which interact with each other via the Internet. The server side of an application usually includes interaction with an application-dependent database, as suggested in Figure 1.6.

Examples of client-server computing include apps embedded within a Web browser for purchasing goods at an on-line store or paying bills from a bank account. The browser and the application client sit on the person's computer or smartphone. The on-line store or bank account's server resides on the other side of the connection. The person typically gains secure access to the full application by providing a Username and a Password. SSL encryption is a

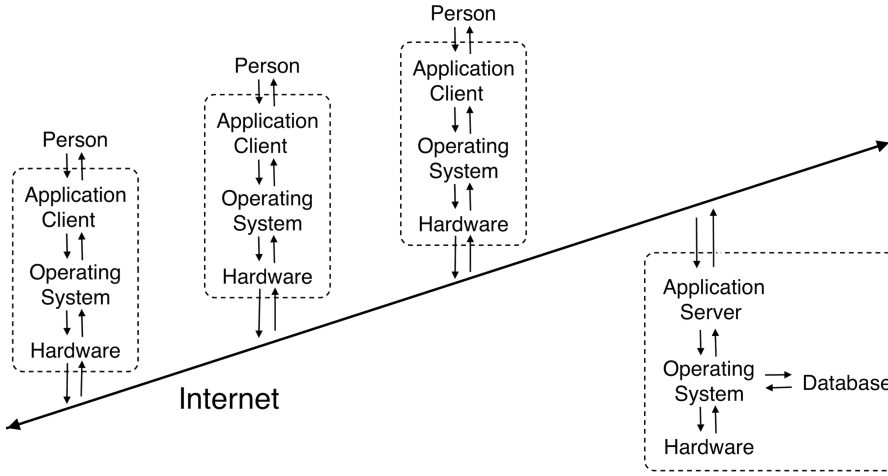

FIGURE 1.6 Client-Server Framework.

common tool for connecting the application's client with its server, ensuring that the information traveling on the Internet between the two remains secure.

When several clients can each connect to a *network* of remote servers hosted on the Internet, this is called "cloud computing." The metaphor here is that, for a user, the individual servers providing different services are invisible, as if obscured by a cloud. Thus, cloud computing is an extension of the notion of client-server computing – it just adds more interrelated servers to the mix, as shown in Figure 1.7.

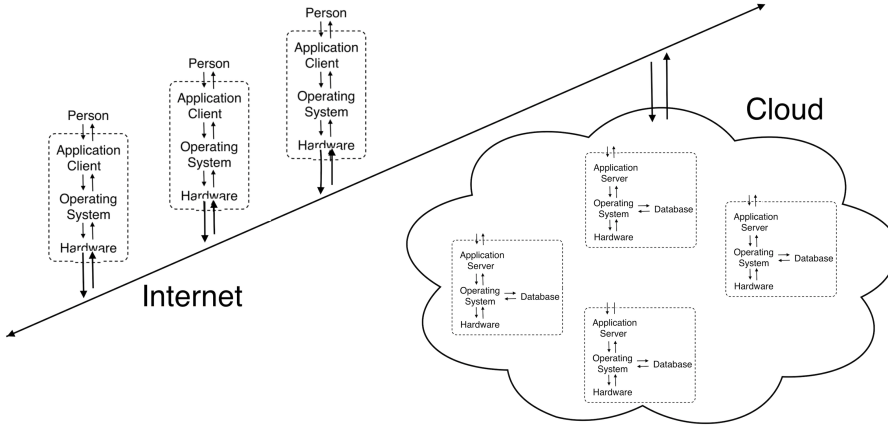

FIGURE 1.7 Cloud Computing Framework.

An example of cloud computing occurs when a user connects to a collection of related application servers through a single secure login. Once logged in, the

user can use several different services at once, such as Google Drive, Google Contacts, GMail, Google Calendar, and Google Meet, while engaging in a videoconference with co-workers and team members.

1.4.2 Web Servers and Bundles

A *web server* is the hardware-software configuration that provides web services on the Internet. A *bundle* or *stack* includes the web server, a host operating system, a database system, and one or more programming (scripting) languages. Each of the servers in Figures 1.6 and 1.7 depicts such a bundle. Two popular web servers/bundles are the Apache Web Server and Microsoft IIS.

The Apache web server is currently the world's most widely-used web server (44% of all active sites in January 2018).[26] One of its bundles is called the "LAMP stack" and it contains an Apache server, a MySQL database, and PHP language scripts that run on a Linux operating system. This is a popular combination for developing software embedded in a web site, partially because it is open source and partially because it runs equivalently on Windows (called WAMP) and MacOS (called MAMP), in addition to Linux (LAMP).

Another Apache-based open source bundle is called AMPPS, which adds Perl and Python to the above collection of programming tools for web apps. AMPPS also runs equivalently on Windows, MacOS, and Linux servers.[35] You can find more detailed information on the different Apache bundling alternatives by visiting `the Apache site` itself.

The NGINX bundle, like Apache, is open source and runs on a variety of platforms. However, NGINX also runs faster than Apache. These two factors are mainly responsible for NGINX market share growing steadily (21% of all active sites in January 2018), compared with Apache.[6]

Microsoft's IIS web server has a smaller market share (7% of all active sites in January 2018). Its application bundle includes the ASP .NET scripting language and Microsoft's MSSQL database. One reason for its limited impact is that IIS is proprietary software and runs only on Windows machines.

At another level is the more modern notion of a "web framework," which differs from a bundle because it hides from the programmer much of the interface between the client, the server, and the database. Two prominent web frameworks in use today are called "Django" and "Rails:"

`Django` is an open source web framework with the Python programming language at its core.

`Rails` is an open source web framework based on the Ruby programming language.

Choosing from among the many bundles and frameworks available for developing a new web-based CO-FOSS product is governed by several factors. Prominent factors include the nature of the application, the availability of open source code for reuse in the new application, and the experience/preferences of the instructor and the student developers themselves.

To facilitate code reuse and to capitalize on prior student preparation, all our CO-FOSS projects are client-server Web applications. They all use an Apache web server bundle with PHP, JavaScript, and MySQL. So all the examples in this book use this bundle. If we were to develop a new CO-FOSS product with different code-reuse constraints, we would consider using a different web framework such as Django or Rails. In this case, our students would need a different book to supplement this one as a source of examples.

1.5 NEW VS MATURE OPEN SOURCE PROJECTS

While the project and course design discussed in this book can yield both a functioning software product and a successful hands-on learning experience for students, it is important to distinguish this experience from the experience of contributing to a more mature FOSS product.

Mature FOSS products, such as `Eclipse` or `Wordpress`, are far more complex, contain millions of lines of code, and are designed to serve many clients, not just one. Such software is thus not customized (though pluggable architecture encourages customization in a different way), and its development team is larger and more fluid (team members come and leave) than the small team that develops a new CO-FOSS product.

In a mature project, team members have different *levels* of membership, depending on their prior experience and familiarity with the software itself. Cockburn (e.g., [9], p. 9) equates the three levels of team participation—novice, apprentice, and expert—with the three levels of mastery in *Shuhari*, a Japanese martial art concept.[3]

Achieving a particular level of contribution—novice, apprentice, or expert—is defined by a meritocracy, in which merit is measured by the quality and quantity of an individual's prior contributions to the code base. What constitutes merit can include any of the following:

- Code contributions
- Infrastructure support contributions
- Mentorship contributions
- Documentation contributions
- Testing and bug reports

[3]In Shuhari, "shu" means *follow tradition*, "ha" means *break with tradition*, and "ri" means *leave tradition behind*. For more information, see `https://en.wikipedia.org/wiki/Shuhari/`.

With this overview, we can see that students in a 1-semester software projects course will likely gain little hands-on experience as developers by engaging a mature FOSS software project as novices. That is, the complexity of mastering that product to a level where they can become a code contributor is far greater than what most students can expect to achieve in a 1-semester course. Nevertheless, it is useful to explore these mature projects in a bit more detail, so as to inform students interested in entering the software field about the projects and professional communities they can expect to encounter.

1.5.1 Maturity Assessment

Models have been proposed for assessing the maturity of an open source project.[4] Maturity models provide clear quantifiable measures for evaluating a project.

Several factors are considered when evaluating the maturity of an open source project and its community. These include quality assurance, scalability, security, performance, adoption, community strength, community governance, support, and IT management.

A critical measure of an open source project's maturity is the strength of the community that surrounds its development. A strong community can provide a wealth of input from around the world. In comparison, a proprietary product can only benefit from the input of its employees.

A second measure of maturity evaluates the licensing terms and intellectual property management policies and controls in the project. As we saw in Section 1.3, several popular open source licenses have proven to be effective.

Finer-grained methodologies that can assess an open source software project are defined in products like `QSOS` (Qualification and Selection of Open Source Software) and OpenBRR (Open Business Readiness Rating, developed by `Carnegie Mellon University West` and others).

A simpler way to make a quick judgment about the maturity of an open source project is to ask whether each of its core developers has had at least 3 months experience on the project, the project has many more users than developers, and the following actions have been taken:

- the source code is in a repository for public download,
- the project has a public forum where users can post bug reports and queries about the system's adaptability to new users, and
- the development team is open to taking on new members who may volunteer.

This is different from a new CO-FOSS project, which has no immediate intention of broadening either its client base or its developer team, but may

[4] In fact, maturity models are not new in the software engineering industry. The Capability Maturity Model Integration (CMMI) is one such example [1]. CMMI includes best practices for planning, engineering, and managing product development and maintenance.

become more mature in the future. That is, after project completion it may be desirable to consider broadening the developer team or reaching out to other clients who have needs for a similar product. An example of this type of project is the *FarmData* project, which was originally developed by students at Dickinson College for a single client but is now more mature and has a larger user and developer community.

A newly-completed CO-FOSS project that promises to have broader impact may transition to a more mature phase which has been called the *democratic meritocracy* phase. Democratic meritocracy is an ideal form of governance for a young FOSS project, in the sense that all the project's participants are representatives from the meritocracy of contributors. Typically, the sponsors of these types of projects are non-profit foundations whose boards of directors are also selected by the membership. An excellent example of a democratic meritocracy is the Debian community, which has now grown to include thousands of voluntary developers and a very representative process of voting and selecting project leaders.

In the next section, we discuss the long-term transition of a young CO-FOSS project into a more mature project. We will return to that discussion in Chapter 9, which considers the initial steps for making that transition after the project has been completed.

1.5.2 Incubation

The formation of a vibrant community of users and developers marks a critical stage in a CO-FOSS project's transition from origination to maturity. This stage is sometimes called *incubation*, and its purpose is to establish a self-sustaining open-source project with a long lifespan. Both the Eclipse Foundation and the Apache Software Foundation have created *incubators* that invite young open source projects to join.

Two key activities govern how successfully an open source project can pass through its incubation phase and become healthy and sustainable for the long run: building a vibrant community and establishing a viable bug tracking process. At its beginning, a project has only a single (lead) developer, a sponsoring client, and a single user. As the code base evolves, a core group of developers can emerge alongside a handful of "bleeding edge" new users.

How can a CO-FOSS project transform its fragile community into one that has a significant number of developers and users, who are actively reporting issues, and whose developers are contributing bug fixes and new features as the product evolves? Three inter-related groups are vital to this transition: users, contributors, and committers.

Users know and use the software actively. They provide feedback to developers (contributors and committers) when they find bugs or other difficulties when using the software. They also suggest new features that could improve the software's usability or applicability.

Contributors are users who also contribute bug fixes and minor features to the software, but don't have the right to alter the code base itself. Contributions can also be in the form of documentation, administrative support, and testing.

Committers are developers who review contributions and install them in the code base. In this activity, the committer ensures that the code base keeps its integrity—i.e., that the new features are correctly implemented and that the bugs are actually fixed.

Attracting new contributors and committers to the project requires active recruiting, not just passive "openness" for outsiders to join. For example, a certain amount of professional and social networking must be directly associated with the project. The lead developers especially must make reasonable efforts to recruit promising new contributors.

An active Web presence, including easily accessible developer and user forums and project wiki, are valuable catalysts that encourage the successful incubation of a new FOSS project. The establishment of effective on-line forums allows developers and users to discuss specific issues related to the usability of the software itself.

These forums provide an immediate avenue of expression through which a user can report a bug or other technical issue related to using the software. They also provide timely information to developers about the status of all active bugs and other new features that are being considered for the next release of the software.

Finally, these forums provide documentary evidence that can be used when a contributor applies "promotion" to committer status. That person's contributions can be retrieved from the forum's discussion threads and then used by project leaders to evaluate that application.

Community

Abstractly, Jensen [23] characterizes FOSS project organization as a socio-technical interaction network, or *STIN* for short. STINs are always in flux; they are self-organizing networks of activities, people, and tools, and often all these parts are geographically distributed around the world.

More concretely, mature FOSS projects tend to have three main organizational distinctions from proprietary projects. These are:

1. **Self-organizing** Mature FOSS projects allow participants to find their own level and project activity with which to become engaged, based on their interests and skills. Proprietary project leaders assign each participant to a project activity.

2. **Egalitarian** Mature FOSS projects openly invite contributions from everyone. Proprietary projects are hierarchically organized and closed in this regard.

3. **Meritocratic** Mature FOSS projects organize their work around public discussions, and decisions about future directions are based on merit. Proprietary projects organize around the results of private discussions among project leaders, and decisions about future directions are highly influenced by cost and profit.

A FOSS community itself is quite fluid—most users of the software are, in fact, passive users. Jensen [23] (rather harshly) calls these users "free-riders," since they give nothing back in return for the privilege of using the software at no cost.

Users who do provide feedback to the developers do so in an entirely voluntary spirit. Feedback typically occurs through the software's user forum, which is prominently accessible at the project's Web site. Some users may go a step further by providing bug fixes or suggestions for new features in the form of code patches. Engaging in this activity self-promotes the user to *contributor* status. Whether or not a contributor's suggestions are accepted and become part of the code base, however, is decided by a committer.

Promotion to *committer* status is done on the merits of a person's collected contributions to the project over time. In this sense, the contributions become a portfolio of work that can be evaluated to assess the merits of that person's case for assuming the responsibilities of a committer. Who decides on the promotion of a contributor to committer status? This is often done by a core project leadership group. In Apache, for example, the *Project Management Committee* (*PMC* for short), is the group of committers that oversees the project's organization, including making promotions.

While users and contributors are most likely volunteers, many committers become paid employees of the project. What particular skills are required to attain committer status? Generally, an applicant's portfolio contains two types of contributions: those that illustrate technical competence and those that exhibit social skills.

Finally, research [23] has shown that successful FOSS projects must maintain a critical mass of contributors and committers, in relation to "free-riding" users, in order to remain vibrant over the long run. Too many free-riders can kill the project.

Policy, procedural, and technical decision-making in an open source project aims to be fully transparent to all community members. When a complex policy, procedural, or technical issue arises, a member of the PMC typically posts the issue on a public discussion forum, where it is debated and either ratified or rejected by consensus or majority vote. Here, *consensus* means that at least two other developers support a particular solution and no other developers post strong disagreements.

Sometimes, of course, achieving consensus on a contentious issue is not possible. Often conflicts arise during discussions about community infrastructure, technical direction, expectations about developer roles, or interrelationships among roles. These kinds of *conflict*s can be resolved by a process involving a small PMC made up of prominent members of the community. This group has

the job of ensuring fairness throughout the community by solving persistent disputes.

In addition to maintaining the code base, the participants in a mature FOSS project play roles that accomplish several other important project tasks. Here is a summary of these roles and their respective activities:

The **project leader** maintains the project's release plan and current status, and moderates the developer forum.

The **expert user** maintains the software's actors, use cases, requirements, and user roles.

The **lead developer** maintains the software architecture.

Other **developers** maintain the user interface design, domain classes, database design, code base, unit test suite, build package, and build schedule.

Testers manage bug reports and the user forum.

Writers maintain on-line help text.

Bug Marshalls oversee opened bugs and pass them on to developers.

Release Managers overlook the packaging and releasing of new versions of the software for general public download and use.

We also note that it is not unusual for an individual to play two or more of these roles simultaneously, depending on his/her particular interests and the size of the developer and user community.

Bug Tracking

As the code base becomes complex and its community grows, an open source software project should establish a viable process for identifying and tracking the status of bugs that are reported by users and developers. Bug tracking is an important process, especially in a development culture where Linus' Law, "given enough eyeballs, all bugs are shallow," is held in such high esteem.

Bug tracking is a formalized process that governs how bugs are identified and how their resolution is managed by the developers who work with the code base. A particularly interesting bug tracking process is the one established by the Mozilla Foundation, based on its open source bug-management tool called `Bugzilla`.

The "life cycle of a bug" refers to a bug's progress through a series of discrete states. Such a life cycle can be described in the form of a state diagram, as shown in Figure 1.8 for Bugzilla. There, we see that a bug can be in any of five states: "unconfirmed," "confirmed," "in progress," "resolved," and "verified." A bug can be introduced by any user. The management of a bug

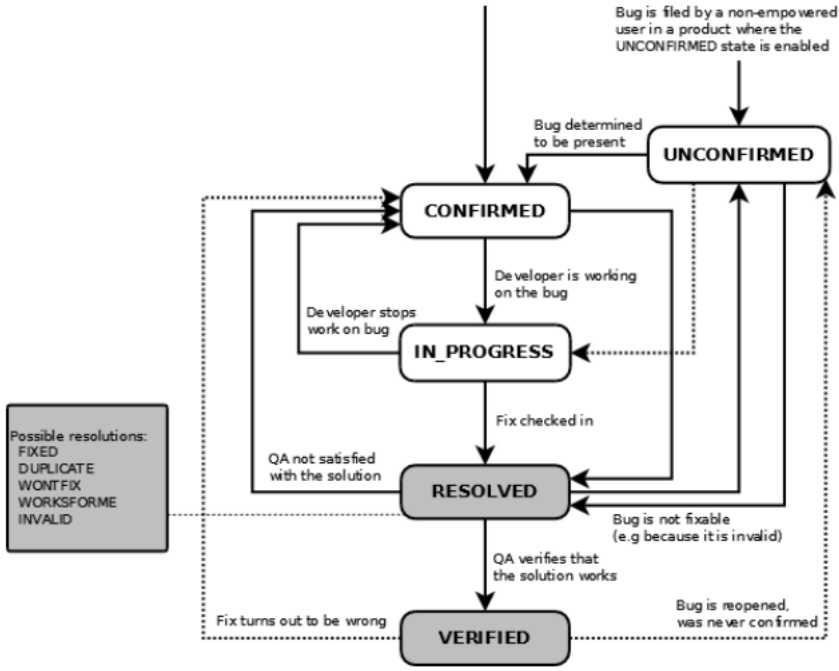

FIGURE 1.8 Life cycle of a bug, from Bugzilla documentation, p 9.

is done by developers with commit privileges, and the whole process can be quite complex.

For example, the Eclipse community uses Bugzilla (see https://npfi.org/bugzilla/). A quick look at the Bugzilla User Guide shows that the process of bug management is not trivial. The process works only if the project has both an active user community for detecting and reporting bugs, and an active developer/committer community for fixing, resolving, and verifying the fixes.

For projects with large user and developer communities, this summary provides insight into how to implement Linus' Law in a practical setting. It also shows that the process of fixing a bug and verifying that fix relies strongly on both the openness of the source code and the atmosphere of trust that exists among users and developers.

1.6 INTO THE WEEDS

This section provides guidance on how to best navigate the rest of this book, depending on whether you are an instructor, a student, a client, or a professional software developer.

1.6.1 To the Instructor

You are about to organize a software development course that has ambitious goals and outcomes for your students.

As in a traditional course, you can expect that your students will learn basic principles of software development, including the stages in the development cycle – requirements, design, coding, testing, deployment, and maintenance. You can also expect that they will master the steps in a client-centered development cycle – meet, plan, code, test, and evaluate – as they develop their CO-FOSS product with the help of a real client.

Two key ideas will help ensure the success of your course: 1) the idea of open source software and 2) the idea of client-centered development. An important goal of your course is to convince students that software development can be a successful and productive enterprise, and that each one of them can play an essential role to achieve that success.

To accomplish this goal, you will need to introduce students to specific software tools to facilitate their work – a system stack, a "sandbox server," an integrated development environment (IDE), a software repository with version control, unit testing, face-to-face collaboration, and open source licenses. Working as a team, you, your students, and your client will be engaging in a client-centered process that will help ensure successful project completion at the end of the course.

For the projects discussed in this book, our students used a system stack with an Apache server, MySQL, PHP, and JavaScript running on Linux, Mac OSX and Windows machines. In one project, students also used Android development tools to develop an app for an Android tablet using Java. They used Mercurial version control with Google Code repositories. Because Google Code is no longer supported, these projects have migrated over to GitHub repositories and they use the Git version control system. For a peek at the current versions of these projects and their code bases, see https://github.com/megandalster/.

For coding, testing, and committing code to the repositories, our students used an Eclipse IDE with either PHP/JavaScript or the Android Java programming language. For unit testing their PHP code, they used SimpleTest. At this time, however, PHPUnit would be the preferred unit testing tool, and Android Studio would be the preferred IDE for developing an Android Java project.

The tools you select for your project will depend upon several factors: your own experience, your client, your application, other open source code bases that you can reuse, and your students' skills coming into the course. In our experience, students were intermediate or advanced CS majors, with moderate-to-excellent programming skills, Java or Python language skills, modest IDE experience, and data structures familiarity. They had little or no prior experience with large software projects, databases, version control, or team programming.

Finding a client and a software project that you and your students can complete in this course requires particular care and attention. While this work goes beyond an instructor's normal teaching load, it is key to successfully organizing and teaching the software development course in this way.[5] For more detailed guidance on finding a client and a project, see Chapter 2.

It is no accident that the 3-month project implementation stage is the length of a 1-semester course. This requires that you design the software project carefully so that your students can complete it by the end of the course. We think that student success in this regard is the most important outcome. Failure to complete the project would not only be a disservice to the client; it would also leave your students with a negative view of software development as a profession. For more detailed guidance on organizing your project and your course around it, see Chapter 3.

To support your teaching the course itself, this book provides detailed guidance on the development process, including links to code bases, "sandbox" databases, assignments, and mini-lectures drawn from our own experiences. In particular:

Chapter 4 introduces students to the fundamentals of the client-centered process, open source licensing, team roles, using the code repository, and communicating with team members.

Chapter 5 covers principles of programming languages, IDEs, and coding the domain classes. It also introduces the idea of test case design, unit testing, code synchronization with a "sandbox" server, issue tracking, and client review.

Chapter 6 covers the principles and practice of database design, tables, queries, CRUD functions, testing, security, and client review.

Chapter 7 introduces design and development of the user interface, including the model-view-controller pattern, usability testing, and client review.

Chapter 8 offers suggestions that will help your students finalize their project and prepare the software for productive use. We offer advice about technical writing, user documentation, developer documentation, and user training.

Chapter 9 addresses issues that you will confront after the semester is finished. It discusses finalizing your project's public repository with the key artifacts that your students have developed – code base, issue tracking, Wiki pages, "sandbox" database, requirements document, and developer documentation. They also provide guidance on finding a developer and passing these artifacts to that developer for deployment and ongoing support.

[5] Because this preparation normally takes up to 2 months prior to the beginning of the course, you may want to seek outside support, either as release time from your home institution or as a grant from an outside source like `NPFI`.

1.6.2 To the Student

You are about to begin a course where you will learn the modern principles and practices of software development. As a central activity in this course, you will help create a real software product that fulfills an important need for a real client. (You may view this as a service learning opportunity embedded within a regular academic course.)

Your success in this course will be determined by many factors, including the quality of your contributions to the software itself (programming, testing, and documentation) and your contributions to the team effort (sharing code and collaborating).

In this course you can expect to use what you have learned about programming and data structures in earlier courses. But you can also expect to gain new understanding about the principles and practice of software development, including the following:

Chapter 4 introduces you to the fundamentals of the client-centered process, open source licensing, team roles, using the code repository, and communicating with team members.

Chapter 5 covers principles of programming languages, IDEs, and coding the domain classes. It also introduces the idea of test case design, unit testing, code synchronization with a "sandbox" server, issue tracking, and client review.

Chapter 6 covers the principles and practices of database design, tables, queries, CRUD functions, testing, security, and client review.

Chapter 7 introduces design and development of the user interface, including the model-view-controller pattern usability testing, and client review.

Chapter 8 offers suggestions that will help you finalize your project and prepare the software for productive use. We offer advice about technical writing, user documentation, developer documentation, and user training.

Unlike other courses, this one is driven by three major elements:

1. teamwork
2. a real-world software product
3. self-education

First, you will work with a team to develop a new software product. The team will include a few other classmates, your instructor as "benign dictator," and a client as the evaluator and recipient of your completed software. So you will learn to work with a team and communicate with teammates and the client as your project develops.

Second, the software product you will be developing is real; you may complete it by completing a series of assignments (called "milestones") with your teammates. Your team will be working with a real client who expects to receive a viable product at the end of the semester. Rather than a final exam or homework grades, you will be evaluated on the basis of one outcome – did your and your team's work result in a viable software product that the client can begin using to help improve his/her work experience?

Third, to achieve a successful result, you will be expected to work like a software professional. That is, you should plan to educate yourself about a variety of topics in programming, database development, and user interface programming. The instructor will not spoon-feed any of this to you. In a broader sense, this course provides you with an opportunity to really embrace the idea of yourself as a "lifelong learner," regardless of your future profession.

If you do enter the software field after graduation, you will quickly learn that it evolves rapidly—new languages, new software methodologies, and (especially) new applications will appear throughout your professional lifetime. To stay at the forefront of the field, you will learn most of what you need to learn on-the-job, and much of this knowledge does not even exist today! So life-long learning is a key element of survival in the software field.

1.6.3 To the Client

You are about to take part in a project that will develop new software for your organization. When completed, this software will help improve one of your organization's mission-critical activities. Your role in this project is to participate in regular meetings with student developers and provide them with critical feedback on the quality of their progress at each stage of the project. Your time commitment for this work may be 1-2 hours per week.

Before the project begins, you will work with the instructor to help him/her understand your organization's software need and compile a "requirements document" that the developers will use as guidance for developing the software to fulfill that need. For more concrete guidance about what to expect from this initial step, please take a look at Chapter 3.

Soon after the project begins, and every 1-2 weeks thereafter, the developers will provide you with a new working prototype of your software. Each time you exercise it, you will see new features that you can test, evaluate, and make suggestions for improvements. Later in the project, as you encounter issues with the software, you will be able to post them on the project's "issues board" so that the developers can address them in a timely manner. For more guidance on what to expect during this central stage of the project, please take a look at the "Client Review" sections of Chapters 5 through 7.

Especially important to the success of this software is your feedback on the quality and ease of use for the features introduced in Chapter 7. Toward the end of the project, your constructive suggestions on the "user help" pages and other documentation (see Chapters 7) will also be very important.

1.6.4 To the Developer

You are about to receive a newly-developed software product to install on the client's server or web site. You should expect to receive the source code, database, and all documentation from the instructor as a complete package.

You should test the software for robustness and work with the instructor to fix any new issues you discover before installing it on the client's server. After installing the software, you should expect to provide ongoing support as new issues arise, especially during the first few weeks of active use by the client. For more guidance on interfacing with the instructor and the client and deploying the product, please take a look at Chapter 9.

In some CO-FOSS projects, a professional developer like yourself has been involved much earlier in the project, assisting the instructor and client with project requirements and/or meeting regularly with the student team to provide professional advice on various technical challenges of the project as they occur. For example, this approach was taken successfully in 2016 by an instructor at Green River College whose students developed software with continuing support and mentoring from a professional developer (for more information, see https://npfi.org/the-2016-npfi-grant-award/).

You may see other benefits from your joining the project earlier than the time when the instructor hands you the final product for deployment on the client's server. For example, you may want to brush up your own skills with client-centered development using platforms and tools you haven't used before. Or else you may want to engage actively with the class as an indirect recruiting tool for your own company. Or you may simply want to give back to your community in a way that utilizes your technical skills more directly than what you do in your "day job."

Whatever your motivation, you should talk with the instructor about engaging more actively in the project and work out an arrangement that works for both of you. Once you make such an arrangement, you can use the rest of this book as a resource guide for working alongside the student developers, selecting from among Chapters 4 through 7 the ones that are appropriate to your needs.

Finally, we should mention that the Non-Profit FOSS Institute (NPFI) exists solely to provide *pro bono* support for instructors, clients, and developers who would like to start a new CO-FOSS project or get involved with a current one. Many aspects of NPFI support are mentioned throughout this book. For information about a particular current or future project that interests you, please feel free to contact NPFI at https://npfi.org/contact/.

1.7 SUMMARY

This chapter introduces the larger ideas behind software development, focussing on a model for open source development which we call CO-FOSS. This model is particularly suitable for inclusion in a course where students

gain real-world experience developing a new software product for a local client. Software architectures, frameworks, and licensing are presented with a particular focus on their influence on open source development.

The distinctions between a new CO-FOSS product and the mature FOSS products that students will encounter if they enter the software profession after graduation are also presented. After all, most mature FOSS products are the result of incubation from earlier CO-FOSS projects.

This chapter concludes with some advice to the instructor, the student, the client, and the professional developer on how to best use the material in the remainder of the book.

1.8 MILESTONE 1

1. If you are an instructor, what are the risks and rewards for your launching a new CO-FOSS project with a student team?

2. If you are a student, what did you learn that you did not already know about the software world or the process of software development?

3. If you are a non-profit representative, what critical operations in your organization would be well-served by the addition of new customized free and open source software?

4. If you are a professional software developer, how would engaging in a new CO-FOSS project improve your resume or your personal well-being?

Bibliography

[1] http://en.wikipedia.org/wiki/capability˙maturity˙model˙integration.

[2] http://en.wikipedia.org/wiki/gnu˙general˙public˙license.

[3] http://en.wikipedia.org/wiki/linux˙kernel.

[4] https://en.wikipedia.org/wiki/unified˙modeling˙language.

[5] http://www.gnu.org/philosophy/free-sw.html.

[6] http://www.hostingadvice.com/how-to/nginx-vs-apache/.

[7] BBC. UK government backs open source. *Online*, February 2009.

[8] Grant Braught, John McCormick, James Bowring, Quinn Burke, Barbara Cutler, David Goldschmidt, Mukkai Krishnamoorthy, Wesley Turner, Steven Huss-Lederman, Bonnie MacKellar, and Allen Tucker. A multi-institutional perspective on h/foss projects in the computing curriculum. *ACM Transactions on Computing Education*, 18(2):1–31, July 2018.

[9] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2005.

[10] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[11] European Commission. Open innovation 2.0 conference, June.

[12] Festival Latinoamericano de Instalacion de Software Libre. https://flisol.info/flisol2017.

[13] FLOSS Definition. https://en.wikipedia.org/wiki/alternative˙terms˙for˙free˙software#floss.

[14] Free Software Foundation. https://en.wikipedia.org/wiki/gnu˙general˙public˙license.

[15] Free Software Foundation. http://www.fsf.org/licensing/licenses/.

[16] Mozilla Foundation. http://www.mozilla.org/about/.

[17] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.

[18] GNU. http://www.gnu.org/gnu/initial-announcement.html.

[19] Standish Group. https://www.infoq.com/articles/standish-chaos-2015.

[20] Jim Hamerly, Tom Paquin, and Susan Walton. Freeing the source: The story of mozilla. *Open Sources: Voices from the Open Source*, pages 197–206, 1999.

[21] Gijs Hillenius. Amsterdam to make openoffice and firefox default on city desktops. *Online*, April 2009.

[22] Gijs Hillenius. Fr: Gendarmerie saves millions with open desktop and web applications. *Online*, 2009.

[23] Chris Jensen and Walt Scacchi. Governance in open source software development projects. In Pär gerfalk, Cornelia Boldyreff, Jesus M. Gonzalez-Barahona, Gregory R. Madey, and John Noll, editors, *Open Source Software: New Horizons*, volume 319, pages 130–142, Heidelberg, May 2010. Springer.

[24] Android License. https://source.android.com/source/licenses.

[25] Bonnie MacKellar, Mihaela Sabin, and Allen Tucker. *Bridging the academia-industry gap in software engineering: A client-oriented open source software projects course*, pages 373–394. IGI Global, 2014.

[26] Netcraft. https://news.netcraft.com/archives/2018/01/19/january-2018-web-server-survey.html.

[27] OSI. http://www.opensource.org/licenses.

[28] Bruce Perens. http://slashdot.org/articles/99/02/18/0927202.shtml.

[29] Bruce Perens. Open sources: Voices from the open source revolution. *The Open Source Initiative*, pages 171–188, 1999.

[30] Federal Source Code Policy. https://sourcecode.cio.gov/#fn17.

[31] The Humanitarian FOSS Project. http://hfoss.org/.

[32] Peter H. Salus. *The Daemon, the Gnu, and the Penguin: How free and open source software is changing the world.* Reed Media Services, 2008.

[33] Desktop Browser Market Share. https://www.netmarketshare.com/.

[34] Red Hat Linux Market Share. https://www.gartner.com/doc/reprints?ct=150106&id=1-26vhvsw&st=sb.

[35] Sourceforge. https://sourceforge.net/projects/ampps/.

[36] Richard Stallman. http://www.gnu.org/gnu/manifesto.html.

[37] Richard Stallman. http://www.gnu.org/philosophy/use-free-software.html.

[38] Richard Stallman. *The GNU Operating System and the Free Software Movement*. O'Reilly, 1999.

[39] Richard Stallman. Why 'open source' misses the point of free software. *Communications of the Association for Computing Machinery*, 52(6):31–33, June 2009.

[40] Zdnet Survey. http://www.zdnet.com/article/its-an-open-source-world-78-percent-of-companies-run-open-source-software/.

[41] Jenifer Tidwell. *Designing Interfaces 2e: Patterns for Effective Interaction Design*. O'Reilly, 2010.

[42] Linus Torvalds. *The Linux Edge*. O'Reilly, 1999.

[43] Allen Tucker, Ralph Morelli, and Chamindra de Silva. *Software Development: An Open Source Approach*. CRC Press, Boca Raton, Florida, 2011.

[44] David A. Wheeler. http://www.dwheeler.com/essays/floss-license-slide.html.

[45] Robert Young. Giving it away: How red hat software stumbled across a new economic model and helped improve an industry. *Open Sources: Voices from the Open Source Revolution*, pages 113–126, 1999.