Chapter 2

Working with a Project Team

"It takes a whole village to raise a child."
—*African proverb*

A free and open source software (FOSS) project is distinguished by a development methodology that combines agile techniques with high levels of interaction among developers and users.

This chapter introduces the key ideas behind FOSS development, including agile techniques, the use of frameworks, code reading, documentation, and teamwork.

Two types of FOSS projects, which we call "client-oriented" and "community-oriented" projects, are characterized in this chapter. Participating in a client-oriented project requires a personal level of communication with team members and users. Participating in a community-oriented project requires the use of a different set of communication channels and conventions.

By completing this chapter, readers should be prepared to begin working on a client-oriented FOSS project team or contributing to an ongoing community-oriented FOSS project. The remainder of this book provides guidance for continuing that effort and making a real contribution to an ongoing FOSS project.

2.1 Key FOSS Activities

A number of key activities play important roles in any effective FOSS development effort, whether it be client-oriented or community-oriented. These strategies are summarized in this section and they are illustrated often in later chapters.

2.1.1 Agile Development

The software development world has recently embraced a powerful paradigm called *agile development*. Agile development emerged in response to the many

failures of the traditional software process that were noted in Chapter 1.

In a traditional development process, each stage – requirements gathering, design, coding, testing, and delivery – 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 of the process, but not in the crucial middle stages. This is illustrated in Figure 2.1.

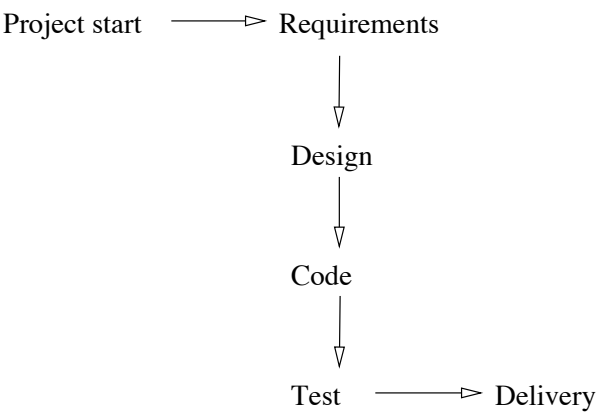

FIGURE 2.1: The traditional software development process.

The agile process is more fluid, in the sense that each stage has a smaller time scale and all stages repeat in a continuing cycle until the project is complete. The agile development cycle requires that the user be engaged continuously, thus allowing new features to be added from user-provided scenarios and test cases throughout the development period. Thus, the software product evolves from the bottom up, rather than from the top down. This process is pictured in Figure 2.2.

Since users are involved throughout the agile development process, they play a critical role at each repetition of the design, coding, testing, and review stages. Because these stages are repeated again and again, each iteration provides a new opportunity for *debugging* and *refactoring* the code base in preparation for adding new functionality in 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.

Users play a key part in debugging, since they are the ones who most often identify bugs, both during and after the development process has been completed. Continuous communication between users and developers is essential for debugging to be effective. FOSS development, which is an agile process,

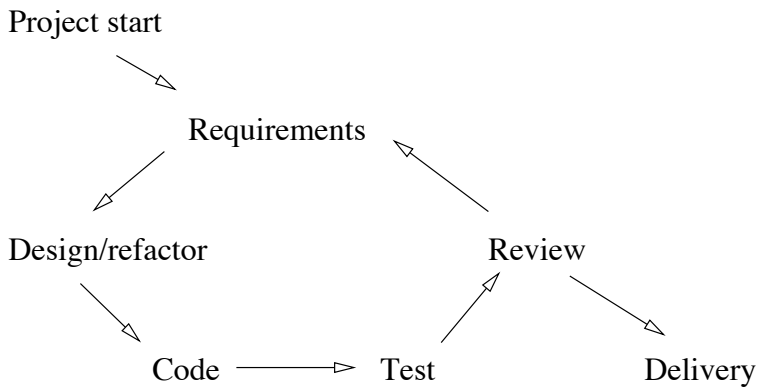

FIGURE 2.2: The agile software development process.

is especially effective in this regard, since it relies on continuous interaction between users and developers.

Refactoring a program means to read the code, find instances of poor programming practice (from either a readability or an efficiency standpoint), and reorganize (usually simplify) the code so that it performs the same functions in a more readable and/or efficient way. No new functionality is added during refactoring, only an improvement in the quality of the code.

The ability to read code with a critical eye, especially code not written by oneself, is a fundamental skill in software development. Software is seldom written by a single person from scratch, contrary to what might have been inferred in introductory programming courses. Instead, software is usually developed incrementally by many developers, each one adding new code to an existing "code base," thus adding a new feature to its overall functionality.

2.1.2 Using Patterns

The overall architecture of a software system refers to the organization of its code base in a way that best reflects the system's functionality, supports its systematic development and deployment, and allows effective distribution of programming tasks among the team's developers.

A *design pattern* is an abstraction from a concrete programming form that keeps reappearing in different coding contexts. Once such an abstraction is identified, it is described and given a name by which programmers can refer to and reuse it.

Examples of design patterns include: Strategy, Visitor, Observer, Mediator, Factory and Prototype (for object creation), and Iterator (for traversals). For more discussions about these and other software design patterns, see **en.wikipedia.org/wiki/Design_pattern**.

An *architectural pattern* moves the idea of a design pattern to the architectural level of a software artifact. At this level, we have patterns for implementing computer-user interaction, such as Client-Server and Model-View-Controller (MVC), as well as others like Multi-Tier, Pipe-and-Filter, and Peer-to-Peer. See http://en.wikipedia.org/wiki/Architectural_pattern) for more discussion of archtectural patterns. The Client-Server, Multi-Tier, and MVC architectural patterns are more fully discussed and illustrated in Chapters 4 and 8.[1]

Once a system's architecture is determined, the code base can be organized appropriately. For example, software organized using a client-server or multi-tier pattern often has the following natural directory structure for its code base.

Domain modules/classes These define the key elements of the application. The domain modules define the name space upon which all other modules are derived. They reflect terminology familiar to users as they exercise the software. Names must be chosen carefully, so as to promote clarity of communication among developers and users. Development of the domain modules in a software system is the subject of Chapter 6.

Database modules/classes These define the tables and variables that comprise the persistent data in the system. The database resides on the server in a client-server software system. Each table is related to one or more core classes/modules. Development of the database modules is the subject of Chapter 7.

User interface modules These implement the system-user interactions that take place on the client side of the software system. The user interface implements the functionality given by the use cases in the requirements document. Development of the user interface is the subject of Chapter 8.

Unit test modules Each of these modules is developed in conjunction with a core, database, or user interface module. Together, the unit test modules comprise a "test suite" for the software system. Whenever any module in the system is changed (refactored, added, or expanded), the test suite must be run to be sure that the system's functionality is not compromised by that change. Unit testing strategies are revisited in each of Chapters 6, 7, and 8.

[1] By contrast, a *software framework* is an organizational generalization for a specific type of software application. It is language- and system-specific, and its code inevitably contains instances of various design and architectural patterns. For example, Web application frameworks are implemented in PHP, Java, Ruby, and C++. Well-known PHP frameworks are called CakePHP and the Zend Framework. Well-known Java frameworks are called Apache Struts and Spring. For more discussion of software frameworks, see http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks.

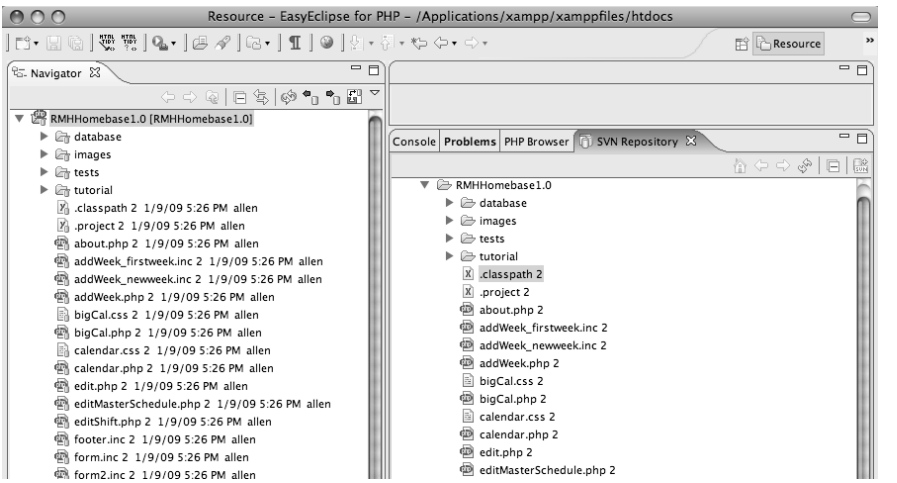

FIGURE 2.3: Using the Eclipse environment with Subversion.

On-line help tutorials These modules are typically developed last, and in conjunction with user training that accompanies delivery of the software. Their structure tends to mirror that of the user interface modules, so that there will be one help screen for each distinct use case that users will be exercising with the system. Writing on-line help modules is covered carefully in Chapter 9.

Organization of individual modules within the code base is initially done by the lead developers. Refinements to this code base are made by individual developers on the project team. To facilitate collaboration, and to ensure the integrity of the code base when several developers are making changes to it simultaneously, a "version control system" is used to synchronize their work.

As shown in Figure 2.3, the code base for *RMH Homebase* follows this general organizational scheme. Figure 2.3 also shows how several developers can synchronize their work using a version control system (SVN in this case). More discussion of version control systems appears in Chapter 3.

2.1.3 Reading and Writing Code

Program reading and writing is, of course, the central activity within software development, since the program is the software. The ability to read and write program code in the language(s) of the application is a key requirement for all the developers in a software project. A corollary requirement is that developers must be able to learn new language features and skills as needed to support the timely implementation of new software elements.

Many fine programming languages are in use for developing contemporary software. This book's examples are shown in PHP and MySQL, although

```
<div id="container">
    <?PHP include('header.php');?>
    <div id="content">
<p>
   <strong>Personnel Input Form</strong><br />
   Here you can enter new personnel into the database.</p>
        <?PHP include('validate.php');?>
        <?PHP
            //Check if they have submitted the form
            if(!array_key_exists('_submit_check', $_POST)) {
                include('form.inc');
            }
            else{
                //in this case, the form has been submitted
                $errors = validate_form(); //step one is validation.
                // errors is an array of problems with their submission
                if($errors){
                    //if any errors exist, display them and give them the form to fill out again
                    show_errors($errors);
                    include('form.inc');
                }
                //otherwise this was a successful form submission
                else {
                    process_form();
                }
            }

        ?>
        <?PHP include('footer.inc');?>
    </div>
</div>
```

FIGURE 2.4: Example code from *RMH Homebase*.

readers who are familiar with Java or C++ should have no trouble understanding their meaning. Other projects and tutorials appearing at the book's Web site myopensoftware.org/textbook use different languages, such as Java, so that readers working with different languages in their projects should look there for additional support.

Reading code is an especially important skill. The code you read in published programming textbooks is, unfortunately, not typical of the code you find in most software applications.

In general, code published in textbooks tends to be more uniform and readable than code found in real applications. By contrast, code that appears in software applications often reflects more than one programming style and may contain elements that are inefficient, unnecessarily verbose, or just plain difficult to read.

For example, Figure 2.4 shows some example PHP code from the *RMH Homebase* project, and Figure 2.5 shows what it produces in a Web browser. PHP code can be embedded inside the HTML of a Web page by using the HTML tags <?PHP and ?>. Whenever such a page is rendered, the PHP code is executed. In this example, the embedded PHP code calls functions that display the header (which includes the menu bar), the applicant's information form, and a footer (not shown in Figure 2.5).

Writing good code requires using conventionally accepted coding and documentation practices. This is especially important when the software is being developed by a team. Unfortunately, much of the code that underlies actual software products does not reflect the use of good practices. Thus, some of

FIGURE 2.5: Output of the example code in Figure 2.4.

a programmer's work includes rewriting poorly written code to make it more readable and receptive to the addition of new features.

Below is a list of widely used coding standards for common program elements. These standards are illustrated here in PHP, though similar standards exist for Java, C++, or any other contemporary programming language.

- Naming and spelling—Class, function, variable and constant names should be descriptive English words. Class names should be capitalized; if they consist of more than one word, each word should be capitalized—e.g., Person, SubCallList.

  - Multiple-word function and variable names should separate their adjacent words by an underscore—e.g., `$primary_phone`. Alternatively, these names can be written with each non-first word capitalized—e.g., `$primaryPhone`.
  - Global variable names should be written in all-caps and begin with an underscore—e.g., `_MY_GLOBAL`.
  - Constant names should be written in all-caps—e.g., `MY_CONSTANT`.

- Line length—A line generally contains no more than one statement or expression. A statement that cannot fit on a single line is broken into two or more lines that are indented from the original.

```
/**
 * fill a vacancy in this shift with a new person
 * @return false if this shift has no vacancy
 */
function fill_vacancy($who) {
    if ($this->vacancies > 0) {
        $this->persons[] = $who;
        $this->vacancies=$this->vacancies-1;
        return true;
    }
    return false;
}
```

FIGURE 2.6: Documentation with indented blocks and control structures.

- Indentation—Use consistent indentation for control structures and function bodies. Use the same number of indentation spaces (usually four) consistently throughout the program—e.g., see Figure 2.6.

The use of coding standards such as these is sometimes met with resistance by programmers. They argue that pressures to complete projects on time and on budget prevent them from the luxury of writing clear and well-documented code all the time.

This argument breaks down when the software being developed is subject to later revisions and extensions, especially by other programmers who need to read the code as they revise and extend it. This is especially true in the open source development world, where the source code typically passes through several sets of programmers' eyes as it evolves.

2.1.4 Documentation

As a general rule, effective software development requires that the code not only be well written but also be well documented.

Software is well documented if a programmer unfamiliar with the code can read it alongside its requirements statement and gain a reasonable understanding of how it works. Minimally, this means that the code should contain a documentary comment at the beginning of each class and non-trivial method, as well as a comment describing the purpose of each instance variable in a class. Additionally, each complex function may contain additional documentation to help clarify its tricky parts.

When reading a code base for the first time, a new developer may find a shortage (sometimes a complete absence) of commentary documentation. If the code is well written, the reader may still be able to deduce much of its functionality from the code itself. In fact, it is a good exercise for a developer new to a code base to add commentary documentation in places where it is

```
/**
 * class Shift characterizes a time interval for scheduling
 * @version May 1, 2008
 * @author Alex and Malcom
 */
 class Shift {
    private $mm_dd_yy; // String: "mm-dd-yy".
    private $name;     // String: '9-12', '12-3', '3-6',
                       // '6-9', '10-1', '1-4', '12-2', '2-5'
    private $vacancies; // no. of vacancies in this shift
    private $persons;  // array of person ids filling slots,
                       // followed by their name,
                       // e.g. "Malcom1234567890+Malcom+Palmer"
    private $sub_call_list; // "yes" or "no" if SCL exists
    private $day;      // string name of month "Monday"...
    private $id;       // "mm-dd-yy-ss-ss" is a unique key
    private $notes;    // notes written by the manager
    ...
```

FIGURE 2.7: Inserting comments in the code.

lacking. That is, it improves one's own understanding and it contributes to the project by improving future readers' understanding of the code.

A good way to begin refactoring an under-documented code base is to add documentation in places where it is lacking. This helps not only to improve the quality of the code for future readers, but also to reveal bad smells and improve the code's receptiveness to the addition of new features.

Documentation standards exist for most current programming languages, including PHP and Java. These latter are supported by the JavaDoc and PHPDoc tools, respectively. They implement a standard layout for the code and its comments, and they automatically generate a separate set of documentation for the code once it is fully commented.

For example, consider the Shift class in the *RMH Homebase* application discussed above. Its documentation contains a stylized comment of the form:

```
/**
 *
 */
```

at the head of each class and each non-trivial method, as well as an in-line comment alongside each of its instance variables. This is shown in Figure 2.7.

Notice that this documentation contains so-called *tags*, such as `@author` and `@version`. When used, each of these tags specifies a particular aspect of the code, such as its author, the date it was created or last updated, and the value returned by a method.

TABLE 2.1: Some Important PHPDoc Tags and Their Meanings

Tag	Meaning
@author	name of the author of the class or module
@version	date the class or module was created or last updated
@package	name of the package to which the class or module belongs
@return	type and value returned by the function or method
@param	name and purpose of a parameter to a function or method

A short list of the important PHPDoc tags with a brief description of their meanings is shown in Table 2.1.[2] Other languages, such as Java, have similar tagging conventions. It is important for developers using those languages to learn those tagging conventions before writing any documentation.

Once a code base is completely documented, a run of PHPDoc (or JavaDoc, as the case may be) will generate a complete set of documentation for all its classes and modules. For example, Figure 2.8 shows the documentation that is generated for the `Shift` class shown in Figure 2.7.

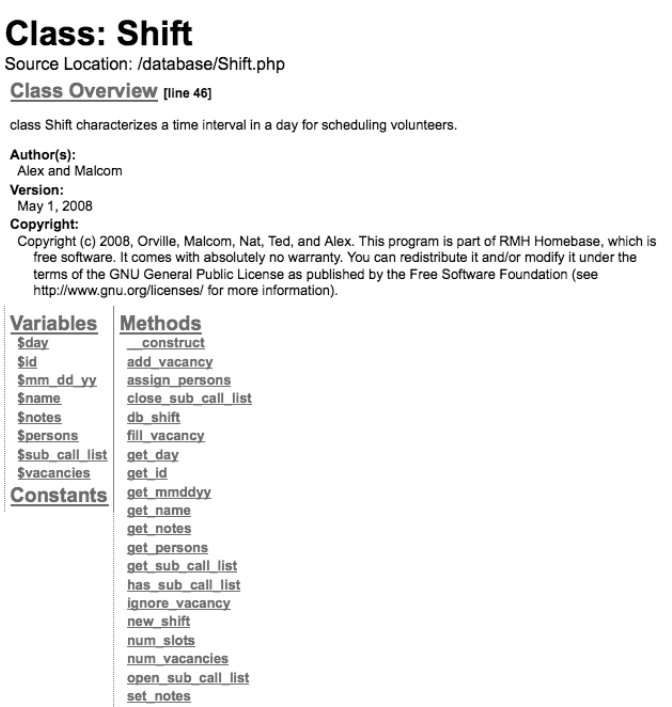

FIGURE 2.8: PHP documentation generated for the Shift class.

[2] A more complete list can be found at http://phpdoc.org.

2.1.5 On-Line Help

Documentation serves the needs of developers who are interested in understanding the functionality of a software system. However, it does not serve the needs of users who want to learn the system. For this purpose, separate elements are needed, which often take the form of on-line help tutorials, or equivalently a printable manual that teaches the user how to perform each use case supported by the software.

So, as a separate step from documentation, open source software development requires that on-line help tutorials be developed and integrated within the software itself. These tutorials are written in a language and style familiar to the user, and they should use terminology that is common to the user's domain of activity.

Each help tutorial should correspond to a single user activity, called a "use case,"[3] which typically appears as a group of interactive forms. Thus, the word "Help" in the navigation bar should take the user to that particular tutorial corresponding to the form with which the user is currently working. The tutorial itself should describe a short sequence of steps, with illustrative examples, that shows how to accomplish an instance of the use case that the form implements.

For example, the *RMH Homebase* software has nine distinct use cases (see Appendix A). One of these is called "Change a Calendar" which allows the user to find and fill a vacancy for a shift on a particular day of the week. The corresponding form for that use case is shown in Figure 2.9.

If the user needs assistance, the help tutorial for filling a vacancy can be selected on the menu bar, and the user receives the step-by-step instructions shown in Figure 2.10.

The help tutorial opens in a separate window from the user's current form, so that he/she can work with the form while simultaneously reading the screen.

2.2 Client-Oriented vs Community-Oriented Projects

Prior to the emergence of FOSS development, large-scale proprietary software projects were characterized by a tightly controlled, top-down, hierarchical development process. The newer FOSS development process is characterized by a loosely controlled, bottom-up, distributed community that is highly cooperative and democratic.

This atmosphere of openness and collaboration cannot be emulated in the proprietary development world, since that world demands complete secrecy

[3] See Chapter 5 and Appendix A for more discussion and examples of use cases.

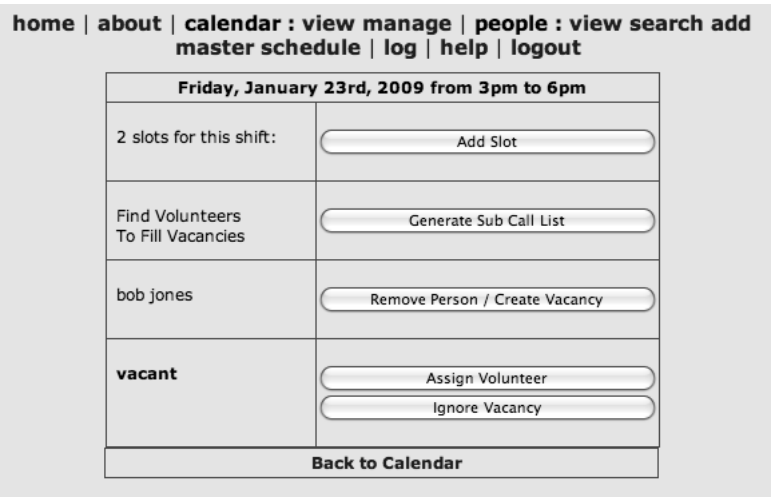

FIGURE 2.9: Form for filling a vacancy on a shift.

and control over its development processes and products. For example, proprietary software developers are required by their employers to sign a "non-disclosure agreement," or NDA, which restricts them from talking about or sharing any of the source code they read or write with anyone outside the company that owns the software.

NDAs thus legally prevent proprietary software developers from discussing any programming issues in public forums or over e-mail, even if such discussions would benefit their work. For this reason, many feel that NDAs lead to reduced programmer productivity and lower quality in proprietary software projects. NDAs, of course, are not applicable to FOSS developers.

As for FOSS, projects tend to have one of two different but complementary development models, which can be called "community-oriented" and "client-oriented."[4]

Client-Oriented Projects The client-oriented development model applies to a new FOSS project that has only a few developers and a single client. The client expresses a need for software that could replace an existing system (often a manual system), and the developers set about designing and implementing software which fulfills that need. Typically, the development period is a few months and the software has only a few thousand lines of code. A good example of a client-oriented FOSS project is *RMH Homebase*, which was introduced in an earlier section.

[4]The "community-oriented" model actually has several sub-classes, which distinguish single- from multiple-vendor support, single from multiple licensing, etc. These finer variations are discussed in more detail in Chapter 10.

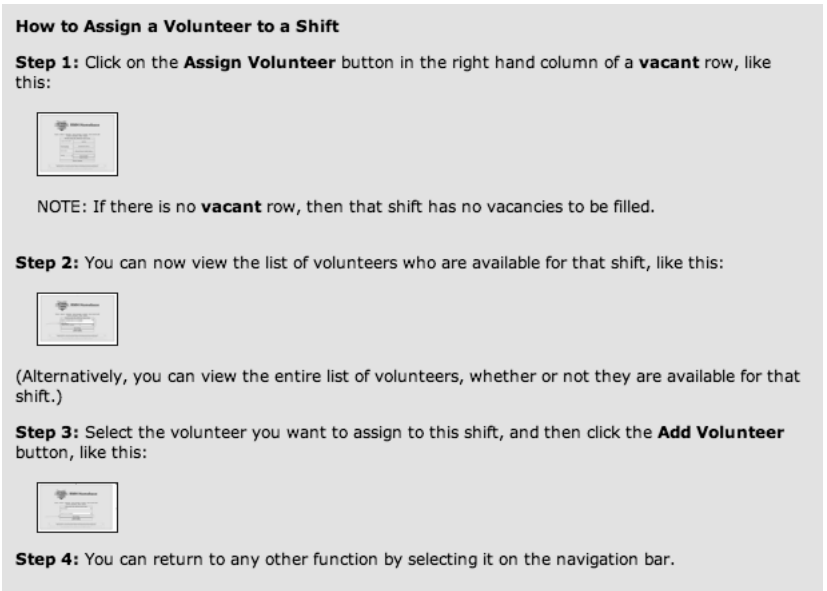

FIGURE 2.10: Help screen for filling a vacancy.

Community-Oriented Projects The community-oriented model applies to a more mature FOSS project, having a large group of developers, covering a wide geographic spectrum, and serving a diverse group of users who share a common software need. The code base for such a project is large, on the order of tens or hundreds of thousands of lines of code. A good example of a community-oriented project is *Sahana*, which was also introduced in an earlier section.

These two genres have a lot in common. Both aim to develop free and open source software using an agile collaborative process, yielding an artifact that can be widely reused, adapted, and shared. Variants of the GNU General Public License provide assurance that such software cannot be privatized. The notion of team-based agile development with a lot of user interaction is central to both genres.

While they share a common philosophy, community-oriented and client-oriented FOSS projects have some important structural and methodological distinctions. These distinctions arise because of differences in their maturity, code base size, user community size, team size, and geographical distribution. These differences are summarized in Table 2.2.

In the course of this study, readers will be encouraged to participate in either a client-oriented or a community-oriented FOSS development project. Because these alternatives have somewhat different collaboration strategies and goals, they are discussed in more detail in the next two sections.

TABLE 2.2: Client-Oriented vs. Community-Oriented Projects

	Community-oriented	**Client-oriented**
Maturity	an existing product to which new features can be added E.g., the current version of *Sahana*	new product, no useful existing product E.g., the original version of *Sahana* (2005)
Size	large code base (tens of thousands or more lines of code) E.g., *Sahana*	small code base (a few thousand lines of code) E.g., *RMH Homebase*
Users	large and diverse group of users with similar needs E.g., *Firefox*	few users, single client E.g., *RMH Homebase*
Team size	large team (hundreds of developers)	small team (5–10 developers)
Geography	many locations worldwide	single location

2.2.1 Project Evolution

Many community-oriented FOSS projects begin as client-oriented projects. Others begin as proprietary software and are later converted to open source. Still others start when a lone developer "scratches an itch" and turns a good idea into a viable code base.[5]

If the original product is well-conceived, well-programmed and available for download from a public repository like Sourceforge, it has the potential to be adapted, refined, or expanded for wider use as a community-oriented FOSS project.

Whatever the starting point, the transition of a software product into a community-oriented FOSS project usually occurs in response to requests from new users to expand and generalize that project's functionality, alongside commitments from new developers to help implement this new functionality. Communication and new energy for that project occurs by word of mouth through the project's public discussion forums. The community thus grows organically, based on the perceived quality and usefulness of the product for a larger group of users.[6]

The Linux project provides an excellent example of the transition of a FOSS

[5] As an interesting "reverse transition," we note that sometimes a new client-oriented project can emerge out of a larger community-oriented project. For example, a customization of the *Sahana* code base is currently being developed to support the special needs of the New York City Office of Emergency Management.

[6] It is important to note that many (most) small FOSS projects never evolve beyond their original stage of development. Projects fail to mature for various reasons, including the absence of a broader audience beyond the single user, a lack of interest by new developers, or some other reason related to broader technology market dynamics.

product from a lone-developer project into a community-oriented project. The Linux project started almost by accident in August 1991 with the following Usenet post by Linus Torvalds, a graduate student at the time [Tor91]:

> Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) ... I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Torvalds' request was answered by a widespread response [wp-e]. In 1992 Torvalds released the Linux kernel under the GNU General Public License (GPL). By 1993 more than 100 programmers were contributing to kernel development. Today, thousands of programmers around the world contribute to the Linux project. Many contributors are individual volunteers, but today many large corporations also support Linux development, including IBM and Red Hat.

The FOSS development model is also characterized by transparency, openness, and collaborative decision making. These characteristics can be seen in how some of the most successful projects describe themselves. For example, the Mozilla project, which created *Firefox*, describes itself as follows [Foue]:

> The common thread that runs throughout Mozilla is our belief that, as the most significant social and technological development of our time, the Internet is a public resource that must remain open and accessible to all. With this in mind, our efforts are ultimately driven by our mission of encouraging choice, innovation and opportunity online.
>
> To achieve these goals, we use a highly transparent, collaborative process that brings together thousands of dedicated volunteers and corporate contributors from around the world with a small staff of employees to coordinate the creation of products like the Firefox Web browser.

Similarly, the Apache Software Foundation, the organizational home of numerous FOSS projects, describes itself as "not simply a group of projects, but rather a community of developers and users [Foua]."

Thus, the FOSS development model, as described by Eric Raymond in his popular book *The Cathedral and the Bazaar* [Ray00], has the following central characteristics:

- The source code, both development versions and stable versions, is always publicly available.
- Stable versions of the software, containing the latest bug fixes, are released frequently.

- Software development requires close relationships and interactions with user groups.
- Technical aspects of the project—features, bugs, algorithms—are discussed publicly by the community.
- Although there is some centralized control, individual users and developers have significant ability to influence the project.
- The community is merit based. Prestige and reputation within the community are based on the frequency and quality of one's contributions.

Of course, the highly distributed FOSS model would not be possible without the interconnectivity and modern communication tools provided by the Internet.

2.2.2 Similarities and Differences

While many elements of a community-oriented project are quite different from those of a client-oriented project, the two genres share many methodological similarities. Here are the similarities.

Agile Methods Despite the differences in scale and scope between a large FOSS project and a small one, both types of projects generally use an *agile* development methodology, as described above.

User Role Both client-oriented and community-oriented projects actively engage users in the development process. However, users in community-oriented projects often play the role of beta tester or bug finder as well. For example, the Mozilla project has more than 10,000 users, many of whom serve as beta testers and provide feedback to developers.

Project Leadership Different leadership models have emerged in different FOSS communities. Some, like Linux in its early days, used the "benevolent dictator" model, with Linus Torvalds as the dictator. Others have a "core team" that shares decision making over what makes it into the code base. Still others are more democratic, in the sense that all developers are empowered to suggest new design directions and goals.

Team Roles Developers play different roles—designer, tester, coder, refactorer, documenter—at different times during the project's evolution. Different contributors tend to self-select into specific tasks based on their interests and abilities. Typically, a community-oriented project has a lot of work going on in parallel.

Code Synchronization Repository The *code synchronization repository* (see Chapter 3 for more discussion) is a common tool used by both community-oriented and client-oriented project developers. Many large

projects use their Sourceforge site for this purpose, while others, such as *Sahana*, use their own site.

Staging and Live Server The use of a *staging server* (see Chapter 3 for more discussion) in a large project is the same as in a small project. Establishing the software on a live server is the responsibility of the user, since different users have different preferences for integrating the software into their operations.

Team Communication All kinds of communication methods and tools are used in a community-oriented project, including email, list servers, Internet relay channels (IRCs), blogs, and video-conferencing. Most of the same procedures apply here as in client-oriented projects; an agreed-upon agenda for each conference, a central location where to-do lists, discussion threads, and other documents are maintained, and so forth (see Chapter 3 for more details).

Programming Activities Programming languages and tools used are similar between small and large projects. However, since large projects tend to have a longer life, they are more likely to have more than one language in their code base. The choice of an integrated development environment, or IDE, is also independent of the size of a project. While Eclipse is popular among all sizes of projects, different developers in large projects are more likely to use different IDEs (see Chapter 3 for more discussion).

Understanding a community-oriented project with an existing code base and large developer community presents its own set of challenges not found in a client-oriented project. The following differences should be kept in mind.

- Some developers in a community-oriented project may have been engaged in the project from the beginning, while others may be just joining the project. In a large project, team members come and go, and they may play different roles over the life of the project.
- A community-oriented project usually has a large number of users, none of whom has the individual capability or authority to shape the project or its software. This user community exerts collective influence over the software's features and development priorities.
- A community-oriented project might have had an identifiable beginning, but it probably does not have a clear end. It is ongoing in the sense that new code is constantly being contributed as features are added or changed to suit the changing needs of its user community. The project evolves over time as if it were a living organism.
- A community-oriented project's development effort is driven by requirements generated in a distributed and bottom-up fashion by individual

users. They are typically adopted using some kind of consensus mechanism among the project's leaders.

- The project's leaders include a group of developers who have "commit privileges," meaning that they can commit actual changes and enhancements to the code base. Others who contribute changes (bug fixes, etc.) must submit their contributions to these developers before they are accepted and committed to the code base.

Community-oriented FOSS projects are usually open and welcoming toward newcomers. The best advice for a newcomer is to take some time to understand how the community accomplishes effective work before attempting to make contributions upon first contact.

2.3 Working on a Client-Oriented Project

A software development team is a group of persons who work together on a project to create a specific software artifact. Throughout the life of the project, individual team members play roles for which they are particularly well suited. Within a particular role, different tasks emerge as the project evolves. Each task is assumed by one or more team members, and remains active until it is completed.

This section describes the general organization of a software team for a client-oriented FOSS project, paying special attention to the roles people play, the tasks they assume, and the tools they use for collaboration, interaction, and code sharing.

2.3.1 Members, Roles, and Tasks

The members of a software team can play any of the following key roles during the life of the project.

- analyst
- developer
- user
- IT technician
- team leader
- observer

The *analyst* role is filled by a person who understands the user's domain, elicits requirements, defines use cases, evaluates risks and alternatives, and sketches the initial design of the software to be developed. The analyst's tasks may also include cost estimation and project scheduling.

The *developer* role is filled by a person who writes test cases from requirements, and reads, writes, tests, debugs, and refactors program code. Developers also refine requirements and use cases, write documentation and on-line help pages, and solicit feedback from users at each stage in the project. In short, a developer is a programmer who can also read, write, and communicate effectively with users.

The *user* role is filled by persons knowledgeable about the application's domain. A user's tasks are to review use cases, provide data for test cases, review the output of code written by developers, provide feedback on the quality of that output, and find bugs in the software after each iteration has been deployed.

The *IT technician* role is filled by persons who configure and maintain the technical environment used by the team. Their tasks are to set up and maintain the code repository, the staging server, the videoconferencing system, the team discussion site, and other software tools needed by the team to develop the software effectively.

The *team leader* role is filled by a person who oversees the development process. The team leader's tasks include setting weekly agendas and milestones, assigning tasks to team members, coordinating overall system architecture, teaching other team members about new techniques, leading regular team discussions, and helping resolve design issues.

The *observer* role is filled by a person interested in watching the project develop and/or whose project management expertise may provide occasional high-level advice on particular design decisions. Overall, the observer role is a fairly passive one.

It is important to note that the same person may play different roles and assume different tasks as different needs arise during the life of a software project. For example, the team leader may also play a developer role when introducing a new concept or technique to other developers.

Moreover, the same person may play two or more roles simultaneously. For example, a developer may also temporarily serve as an IT support person when setting up a master copy of the code base, called a *code repository*, for all the developers to use. Thus, fluidity in role-playing and task assumption is a key element of an agile open source software project.

To illustrate these ideas, the following team was formed to develop the *RMH Homebase* volunteer scheduling system introduced in Chapter 1. This list also identifies the initial role(s) that each person assumed at the beginning of the project.[7]

[7] The names used here are pseudonyms, to protect the privacy of the actual team members.

- analyst—Alex
- developers—Orville, Malcom, Ted, Nat
- users—Gina, Karl, Susan
- IT technicians—Ellis, John, Michelle
- team leader—Alex
- observers—Riccardo, Truman

Throughout the three-month life of this project, roles changed. At the beginning, Alex was an analyst while he gathered requirements, wrote use cases, and sketched the initial design. Early in the project, Orville temporarily became an IT support person when he set up the project's code repository.

To illustrate the team dynamics of task assumption, here are some of the discussions following the first videoconference for the *RMH Homebase* project:

> **Subject: Videochat, 1/30**
> **From: Alex**
> **Date: Tue, 29 Jan 2008 at 7:03pm**
> **Category: Agendas & Meeting Notes**
>
> *Here's a sketch of an agenda for tomorrow's videoconference:*
>
> 1. *Discuss each of our new PHP classes.*
> 2. *Discuss PHP tutorials that will be most useful to us.*
> 3. *Determine standards for coding, testing, and documenting classes as they develop.*
> 4. *Determine what each of us can reasonably accomplish for next week.*
> 5. *Prepare press releases for the project.*
> 6. *Identify any relationship of the project with other volunteer scheduling systems (e.g., VMOSS).*
>
> *Feel free to add anything to this list that I've overlooked.*
>
> *"See" you at 1:00 tomorrow. I'll try to connect with Orville using my iChat and AIM id = alex. If we get stuck, my phone here is 207-729-1234.*
>
> **Orville Wed, 30 Jan 2008 at 2:07pm**
> *We had a videochat today. Present were: Alex, Malcom, Nat, Orville, Ted.*
>
> *We covered our agenda and worked out videochatting bugs. Worked well overall for a first time. Please reply in the comments with the class you will personally work on for next week.*

Malcom Wed, 30 Jan 2008 at 2:08pm
I'm working on Calendar, and researching existing PHP Calendar classes.

Orville Wed, 30 Jan 2008 at 2:08pm
I'll break people.php into separate class files, then work on Person (which needs to be done first since the others extend it). Then I'll work on Applicant and Manager. Also, once Alex decides on a structure for our packaging, I'll implement it in the repository.

Nat Wed, 30 Jan 2008 at 2:55pm
I am currently working on SubCallList. I'll post again and take up one (or more) of Orville's classes that extend Person.

As you can see, Alex initiated the discussion by posting the agenda for the videoconference. The developers Orville, Malcom, and Nat had a short follow-up exchange, in which they summarized the videoconference and assigned themselves to the tasks of developing the core classes Calendar, Person, and SubCallList during the upcoming week.[8]

While roles and tasks are often pre-assigned by the team leader, team members may self-select into and out of tasks as the project evolves, depending on personal preferences, skills, and other commitments.

This fluidity is particularly important in the open source development world, where team members are not always working full time on any one project. They usually have commitments to other work, for instance other course work (if they are full-time students) or other professional activities (if they are users or developers in the corporate world).

2.3.2 Team Dynamics

With the tools in place and the programming activities identified, having a coherent and collaborative team dynamic through the life of the project is a key driver for the success of the project. How and when do collaborations occur, and what is the substance of such a collaboration?

The catalyst in team dynamics is usually the project leader. The project leader sets the tone for effective project collaboration. This tone is characterized by trust, inclusiveness, and confidence that the project will succeed and every team member's participation will be crucial to that success.

The project leader also defines a regular schedule for team videoconferences, usually on a weekly basis. An agenda is set for each videoconference, initially by the project leader and refined by suggestions from the team members.

Out of each videoconference comes a review of the prior week's accomplishments and a new set of goals for the upcoming week, accompanied by

[8] Detailed descriptions of these elements of *RMH Homebase* can be found in Appendix A.

assignments of tasks to team members. These goals and assignments are determined by consent of the team members.

2.3.3 Scheduling, Milestones, and To-Do Lists

The schedule for a software project must respect the constraints of individual team members. It must also be set so that the goals of the project are substantially met at the end of the development period. While flexibility in the accomplishment of particular tasks is important, some overall view of the main goals of the project must be apparent at the outset. That is, all team members must become confident that the project can be completed within the time frame allotted and that each one can play an important role in reaching that outcome.

To that end, a project *milestone* is an intermediate goal that must be met by a particular calendar date. Examples of goals that can be met as milestones include completion of the project's database modules, completion of the online help screens, completion of user training, and so forth. Below are the milestones that had to be completed at the end of the *RMH Homebase* project:

> Wednesday, 23 April, 2008—Complete draft of help tutorials
> Thursday, 24 April, 2008—User training session I at RMH
> Thursday, 1 May, 2008—User training session II at RMH
> Friday, 9 May, 2008—Deliver final system to RMH

A *to-do list* is a collection of smaller units of work that occur in pursuit of a milestone. Unlike a milestone, a to-do list is an assignment of particular tasks to particular individuals on the development team. Below is an example of developing a to-do list that was aimed at the accomplishment of the milestones listed above for the *RMH Homebase* project.

> **Agenda for 4/16 Videochat**
> **From: Alex**
> **Date: Mon, 14 Apr 2008 at 9:44am**
> **Category: Agendas & Meeting Notes**
>
> *Here is what I see needs to be done to finish the project, along with debugging. Feel free to add items that I have overlooked.*
>
> *A. (Nat and Alex) Write and edit the remaining tutorials. Three are now completed, edited, and submitted. Here's a summary of where they stand:*
>
> 1. *Logging on to the Web site—I don't think this one will be needed (the login page is pretty self-explanatory)*
> 2. *People: Searching—done*
> 3. *People: Editing—done*

4. *People: Adding—done, except .gif images need to be fixed*
5. *Calendar: Viewing*
6. *Calendar: Editing (includes removing persons, SubCallList editing, and adding persons to a shift)*
7. *Calendar: Managing (includes creating slots, generating weeks, and publishing)*
8. *Schedule: Viewing and editing*

B. (Ted and Nat) Organize and conduct workshop sessions at RMH for 4/24 and 5/1.

C. (Alex) Draft evaluation form for workshop participants.

D. (Orville) Write help.php.

E. (Malcom and Alex) Go through the code and add comments for phpdoc and future developers. Add the copyright notice at the top of each source file.

F. (Karl) Ensure that the RMH technical person prepares the RMH server with PHP and MySQL so that the system can be installed there. Add a button to the Web site to link to RMH Homebase.

Orville Mon, 14 Apr 2008 at 2:35pm

help.php is written and committed. Nat, take a look at editPersonHelp.inc.php in the tutorial folder. That's the format your final tutorials should follow. Then look at help.php—it just pulls the relevant tutorial (defaulting to what I call index.inc.php which is the list of all tutorials).

Note the difference in the image paths now, in the editPersonHelp.inc.php file.

To see what it will look like, log in, start editing a person, and click "help."

Also, might I suggest we always include a help footer? With links back to "help index," the URL of which would simply be help.php with no argument passed to it.

Alex Tue, 15 Apr 2008 at 7:29am

Question for Orville and Malcom: When Ted and Nat do the workshops at RMH on the 24th and 1st, they will be using the tutorials which are full of examples from our little "jones" family database. However, the live system at hfoss.bowdoin.edu has an empty database (except for admin/admin) which makes the tutorials pretty useless. Can we set up a temporary "training version" of the database with the "jones" family live, so that the tutorials and the workshop will work together?

It seems that this "training version" of the database might be useful to RMH staff in the future. Ted, I think you need to ask Gina and others at the training sessions about this question.

Orville Tue, 15 Apr 2008 at 7:41am

I just ran testDBSchedules to populate the database for the traning sessions—note that the actual shifts aren't filled yet—you need to actually start publishing some weeks.

I can show Nat or Ted how to reset the tables on-site if you guys are comfortable with unix and terminal-style OSes. If not, it's safer to not do it, because we're working on the actual server, and a mistyped command can cause a lot of damage.

Nat Fri, 18 Apr 2008 at 12:56pm

Alex, I've committed most of the calendar management help tutorials. I went with a main manage calendar tutorial which then lets you pick which specific task you are doing. This avoids redundancy since you need to get to the list of weeks page in order to edit anything.

I've completed all but the editShiftsHelp, which I've been trying to make as simple/easy to follow as possible. So feel free to critique the ones that are done; I should be committing editShiftHelp—I'm aiming for 3pm.

Notice here that Orville has taken the lead on setting up a training version of the database in preparation for the two training sessions. Meanwhile, Nat and Alex are communicating about the completion of the final help tutorials in preparation for those sessions. All of the tasks discussed in this dialogue are "to-do" items, which together lead toward achieving the help tutorial, training, and final delivery milestones.

2.4 Joining a Community-Oriented Project

Before joining a community-oriented FOSS project, it is good to begin with a clear understanding of one's personal motivation. Different individuals have different motives for joining, such as:

Academic The project fulfills a particular learning or research goal. FOSS finds a natural home in academic institutions, since all the code and design documents are transparently available for further analysis and modification. It also presents an excellent way to share an implementation of a hypothesis or the results of data gathering with other researchers who have similar goals.

Philanthropic The project fulfills one's desire to contribute to the benefit of one's neighbors. This is like the motive for an engineer joining Engineers without Borders (http://ewb-international.org/) or a doctor joining Doctors without Borders (http://doctorswithoutborders.org/).

Recognition The project fulfills one's desire to be judged by the quality of their work. FOSS development presents an excellent channel to widely disseminate one's portfolio of contributions to peers and mentors. This can contribute to one's resume, which can in turn secure access to other interesting projects in the future. Unsolicited praise from a distant mentor or colleague is often very satisfying.

Commercial The project provides an avenue through which commercial success can be obtained. Even though the product is FOSS, a developer can often use it while providing service and support for using the product. In addition, successful contributions to a project may yield job offers or consulting opportunities in similar areas that are well aligned with the skills developed while contributing to the project.

Hacker The project provides an opportunity to become energized by writing new code and developing new features and products. Many developers simply enjoy creating new artifacts, or "scratching their own itch" (so to speak) by writing code that does something novel and interesting.

User The product allows a user to contribute to software that can be used internally in their own organization. Often such a contribution ensures that your organization has a say in the product's future functionality. Sometimes this contribution involves quality assurance (testing) and/or writing user documentation that will help colleagues in the organization to effectively use the system.

The exact mixture of participants in any particular FOSS project can vary greatly. The variety and roles of individual developers, sponsors, academics, users, and domain experts are unique to each project's nature and goals. Moreover, project participants often have distinct motivations for joining the project, as explained above.

What brings everyone together in a community-oriented project is that each person contributes to the end result. The determination of future directions for the product is governed typically by a meritocracy. That is, a person's rank or title matters less in their influence on a project than does the quality of their contributions to the end result of the project.

The karma earned by contributing valuable code to a project can thus help a developer to become accepted and recognized in future FOSS projects. Contributions are measured not only by code submissions, but also by feedback provided on helping to improve the project's overall quality and usefulness.

2.4.1 Project Selection

Once the motivations for joining a FOSS project become clear, the next step is to find a project that can fulfill those motivations. And there is a wide spectrum of choices available, ranging from single-developer projects to those maintained by multi-organizational consortia.

Firefox, OpenOffice and Linux are the most visible projects in the spectrum. Persons who manage to contribute something of value to any of these projects will certainly gain recognition, based on the sheer visibility that these projects enjoy in the software world.

However, there are also many small- to medium-sized projects that have potential and are growing rapidly. By participating in one of these, you will probably find more opportunities to contribute something significant. By analogy, it is like being a little fish in a small, growing pond rather than a little fish in a large, established pond.

Places to search first are the large repositories such as http://sourceforge.net and http://launchpad.net, which are two of the most popular. Also, directories like http://freshmeat.net serve as a kind of "yellow pages" for FOSS projects that reside elsewhere on the Internet.

The http://sourceforge.net and http://launchpad.net repositories are designed to be accessible to new developers, providing certain assurances on access, downloading, and reviewing any project's code base. These repositories also have integrated mechanisms for collaboration, bug tracking, planning and release management. After gaining access to one of these projects, a developer has a personal login to access the project's tools and forums, which in turn provide strong support for making real contributions.

Although Sourceforge has over 200,000 projects registered, many of these are either dead or inactive. So selecting a project that will be both challenging and rewarding involves the following additional suggestions:

Seek a popular project. A good indicator of a project's popularity is how many downloads its software has each month and how active its user and developer forums are. The user forum enables users to get help and give feedback on the project. Since the users represent the project's target group, it is important to know the size of this group to help understand the project's current and future vitality.

Seek a project with a lot of vitality. A project's statistics often include the number of new code contributions that have been made, the number of feature requests/bugs that have been submitted, and/or the number of new releases that are created each year. All these are measures of a project's vitality and hence its receptiveness to the addition of newcomers.

Seek a project with growth potential. Some projects have a large user base and deliver a lot of value but are now mature and have reached

a plateau in terms of future growth. Thus, these projects will provide very little opportunity for a newcomer to make an impact.

Seek a project that welcomes diversity and promotes contributions. Not all open source projects operate as diverse open communities; some are run by a closely knit group of people with a lot in common. A project that instead has a lot of diversity (e.g., representing various nationalities, types of contributions and motivations) is likely to be more welcoming to newcomers.

Seek a project that will be enjoyable. Newcomers interested in content management systems, business intelligence applications, disaster management systems, or medical information systems should seek projects that are targeted to those applications. Whatever project is selected, it should energize the newcomer every time a contribution is made and the results are posted.

2.4.2 First Contact with the Project

Once a project is selected, the newcomer may take each of the following three steps, in turn, to initiate their involvement with a community-oriented FOSS project:

1. Reading and improving the documentation
2. Using communication channels
3. Working with the code

Each of these steps is enabled by interacting with the project's Web site. Here's some more information about what these steps involve.

2.4.2.1 Reading and Improving the Documentation

The first step to getting involved with a community-oriented FOSS project is to visit its Web site and learn as much as possible about the nature of the project. This step naturally evokes questions, which may be answered by digging into the project's documentation. If answers are elusive, a newcomer may post a question to the community through one of its communication channels.

Of course, the *quality* of different FOSS projects' documentation is highly variable. Some projects provide ample, well-written documents; others provide little or none; and still others provide unhelpful or poorly written documentation.[9]

[9] Programmers are notorious for focusing all their attention on the code and undervaluing the documentation that should accompany it.

Thus, one of the best ways for a newcomer to begin contributing to a FOSS project is to contribute to its documentation. For example, if a project is lacking a good user guide for a module or function, a newcomer could take notes while exercising the system, and then use the notes to write a guide that will help other users.

Similarly, if a newcomer is interested in working on a particular segment of the source code, for example fixing a bug, a good way to start is to read the code itself, along with its documentation. In places where the documentation is missing or unclear, a newcomer may rewrite it. The project's developer community is usually grateful for all contributions like this.

2.4.2.2 Using Communication Channels

Three popular mechanisms are available for users to engage with a project's development community: mailing lists, forums, and IRC channels. The first two are not time zone dependent, so the entire community has an opportunity to access and reply to one's posts. The latter, however, is a long-standing medium (IRC) and is very popular as a hangout area for open source projects.

Each project typically creates its own chat rooms and those interested can join without being invited. The IRC chat room might therefore be useful as an informal place for finding out more about community, before a newcomer starts using the mailing lists more frequently.

For example, the *Sahana* project provides separate mailing lists for developers and users (and there are others for directors and members of the project management team). These are linked to the *Sahana* Web site and include a complete archive of all the project's communications. For example, once a newcomer joins the *Sahana* user list, he/she can read through its entire history. Skimming this history is a good way to get a feel for the community and its current focus areas.

The *Sahana* project is open and welcoming to newcomers, since it explicitly invites them to "Join us and contribute." Its Web site **sahanafoundation.org** greets visitors with a concise overview of the various ways to join its community, as either a developer, a user, or a donor. As noted above, *Sahana* provides a wealth of documentation to help newcomers become familiar with the project.

To join in a project's discussion, newcomers may first need to "register" as a new community member. At that point, it is useful to join the project's mailing lists and chat rooms and try to identify some of its ongoing issues and current goals.

In this activity, it is pragmatic for newcomers to choose tasks and contributions that are within their capabilities and can help improve their overall understanding of the project. It is also good to "scratch one's itch" and focus on those aspects of the project about which one is most passionate.

2.4.2.3 Working with the Code

Finally, a newcomer will eventually want to get involved with the code base itself; adding features, improving documentation, or fixing bugs and refactoring. This activity is best initiated by reading and analyzing the code, especially parts that relate to features that allow "scratching one's itch!"

Newcomers may start this activity by becoming a user and tester for the project's code base. That is, they will download and "play" with the software until they become familiar with its overall organization and functionalities.

During this activity, they will also note any apparent bugs or missing features, as well as features that they think need to be revised. These concerns should be communicated to the project's developer community.

While contributing code is a central component of any software project, that activity is not the only one needed by a successful software project. In the case of FOSS, it is often the non-coding contributions where most of the demand lies. Here is a summary of the different types of contributions that can be made to a FOSS project:

Bug Fixing and Core Development Development fuels the growth of the project and thus is a core contribution. Bug fixing is a valuable entry point for new developers. It is a good way to demonstrate one's capability to contribute something significant to the project, since bug fixes and patches are reviewed by senior developers and are ultimately credited to the contributor if they are accepted.

Testing and Quality Assurance Every time a new release or build is made, new bugs can creep in due to newly coded features or unexpected changes in the system's dependencies. Testing is thus a critical role to ensure the quality of the product for the user base. Testers are the last line of defense on ensuring product quality. However, to play this role, a newcomer does not need to have specific technical expertise. The requirement here is for someone who has a good affinity for quality and can put themselves in the shoes of a typical user.

Documentation Some developers believe that the code is the documentation and should be more than sufficient for anyone to understand the functionality of the software. This belief is often seconded by the fact that documentation itself is an afterthought and is often not kept up-to-date with the code. Of course, every software system is designed expecting that code readers have a certain level of IT literacy. However, not all users are as familiar and literate with common coding conventions and practices. Thus, user documentation, installation guides, on-line help, and quick references are critical elements of a well-documented software artifact.

Release Management Once the core product is built, it needs to be packaged into the various computer environments that the product should

support, which typically includes Windows, Max OS, and Linux. This requires additional work to package the software in a form that is natural to the installation process of the target computer environments.

Creative Design Work Finding a talented developer who is a good creative designer is sometimes a challenge. The skill and discipline needed to develop robust code do not propagate well to the arts. A good creative designer can greatly enhance the user experience and affinity to the product, and thus can have a strong impact on the system's usability. Persons with artistic backgrounds typically excel in this form of contribution.

Infrastructure Support Sustaining the supporting IT infrastructure for a FOSS project can be demanding. It typically involves maintaining Web sites, wikis, mailing lists, IRC channels, forums, a version control system, and a bug tracker. Ensuring that these supporting tools are kept up-to-date is a lot of work, as is the key task of developing a backup and recovery mechanism should the development site fail catastrophically.

Translation As much as English is an established *lingua franca* of the international community, not all users may be able to speak it or understand it. Thus translators are needed to build language packs that translate the user interfaces and supporting documentation into appropriate non-English languages. Here lies the opportunity for diversifying the user community, since anyone who finds that their native language is not supported can contribute translations to the community. The only skills needed are the ability to read English sentences and translate them into one's native language.

2.4.3 Norms for Good Citizenship

When joining a community-oriented FOSS project, newcomers need to understand how the community ticks. Every community has a distinctive culture and norms and it is important to understand them. The most important step is to begin actively participating in discussions, so that the newcomer's name and interests become recognized by other active community members.

Despite their different cultures, most FOSS developers and users follow widely accepted commonsense norms when communicating in a mailing list, forum, or IRC session. Here is a summary of these norms:

Post appropriate material There are often multiple mailing lists for development, users, and other specialist areas per project. One should post questions and comments to the appropriate mailing list, since those supervising that list will be more likely to respond.

Be brief and to the point Shorter is always better. Most people on the list are busy people. It's much easier to hold people's attention with

three paragraphs than with three pages. Sending anything that's larger than it needs to be is an imposition on other readers on the list. If what you're saying requires a comprehensive treatment, it's probably worthwhile to write a brief summary at the start, so that subscribers don't have to spend a lot of time deciding whether what you've written really interests them.

Remove unnecessary "quoted text" Often email users allow you to quote the text of the message to which you are replying—and over the course of an animated email conversation, the back-and-forth trail of quoted text can become very long. To avoid wasting space, quoted text should be minimized.

Avoid kneejerk e-mails Try to make each e-mail constructive, with well-supported points and links to references whenever appropriate. Avoid simplistic, emotive, kneejerk reactions and other messages that you may regret later on. Kneejerk reactions usually aren't interesting reading for other community members.

Stay on topic Once a thread of discussion is started, try to stay on topic of the original discussion. If you digress the original topic might get lost in the discussion. If you need to digress to a new topic, start a new discussion thread (i.e., use a new subject line).

Avoid large attachments Not everyone on this mailing list will have broadband, so avoid sending large attachments and try your best to send links to large files instead.

Proofread your post E-mail doesn't offer the additional modes of communication of in-person communication. Body language, tone, eye contact, etc. aren't available to help you get your point across. Be sure your message is free of errors. You have no idea who is going to read your message—it's not quite the same as personal e-mail.

Don't write in CAPITALS In the e-mail world, this is equivalent to shouting. Text is a somewhat limited medium when it comes to being demonstrative. Use it conservatively.

Refrain from using obscure abbreviations Using abbreviations like *wrt* and *rtfm* is useful, but only if the person reading your message knows what they mean. In other cases, the message may look pretentious and exclusionary.

Avoid making comments that can be mistaken as sarcasm Some messages don't always come across in text for others as they might be intended. Despite their smarminess, "emoticons" or sideways smileys should be used sparingly.

Respect the diversity of the community Successful FOSS projects have contributors from all around the world and from different native languages and cultures. There are also contributors from different communities from academics, emergency management, civil society, and the software development world. Also, since English is not everyone's primary language, not everyone writes e-mail in the same style, so some tolerance for these differences is appropriate.

Avoid foul language and flaming Posting extremely foul or abusive language aimed at a fellow list member is a deal breaker. This includes obscenities, verbal harassment, or comments that would prove offensive based on race, religion, or sexual orientation.

Avoid encouraging (and using) "Read the f...ing manual" responses It is quite likely when you start on a new open source project that you will ask questions to clarify your understanding that have been asked on the same mailing list or otherwise numerous times before. Some developers become agitated having to repeat themselves, and often they place those questions and answers in a FAQ or a manual. Thus, it is best to read the documentation and try first to find the information for oneself before asking a question that's already been answered before.

2.4.4 Becoming a User First

The primary voice to be heard by a FOSS developer is the user's voice. Ultimately, the success or failure of the project depends on how widely it is adopted by users.

Unlike traditionally developed software products, FOSS projects do not typically provide a clear set of requirements specifications at the outset. Instead, that knowledge is held by the user community at large. The only way to learn about requirements is to employ common collaboration mechanisms, ask questions and learn about what the application should do from its user base.

By "becoming a user first," a newcomer may either find bugs that need fixing or suggest ideas for new features that can improve the software's usefulness. So, submitting bug reports and enhancement suggestions is a real contribution, since users are an integral part of the FOSS community and their feedback is important.

Here are a few steps that can be taken by a newcomer to become a user first:

Understand the project's goals The mission statement of the project, often found in the project's **About Us** page, will provide an initial understanding of why the project exists and what common goals drive everyone in its community.

Understand the itch Understand what is compelling people to contribute to develop something in this problem domain by identifying the key problem the system is trying to help solve.

Understand the alternatives Learn about all the alternative open source, freeware, and/or proprietary products that compete for the same target user base, and see how this system differentiates itself from these alternatives. Often the commercial products are ahead of the open source alternatives, so they are a good source for understanding the gaps.

Install the system and play the role of the user There is no better test to understand the user experience with a system than to download, install, and try it for oneself. This will also provide insight into improvements that can be made to better address the needs of the target user base.

Ask questions Asking questions on the user mailing list can clarify one's understanding of the product and confirm that potential bugs and enhancements are really valid. Even simple questions are welcome; this constitutes participation and answering questions also constitutes a contribution in its own right. For example, Launchpad actually gives credit to people who answer users' questions.

Submit bug reports Once a bug is found and it can be confirmed that the system is not performing as it should, one can submit a bug report to the project's bug tracker. A bug report needs to be written so that a developer can recreate the bug without having to ask for further clarificaition. Below is an example of a good bug report that was recently submitted to the *Sahana* project:

```
--------------

Release Version:
Sahana r603 (2010-01-24 13:02:53) - You can obtain this
from the Help->About menu
(http://haiti.sahanafoundation.org/prod/default/about)

Environment:
Firefox 3.5
Windows XP

Prerequisites:
User should be logged in to the system.

Steps:
1) Click 'Mapping'.
```

```
2) Click 'Map Service Catalogue'.
3) Click 'Features'.
4) Click on an ID in the Location list.
   (The 'Edit location' page will appear.)
5) Remove the value in the 'Latitude' field.
   (This step is optional)
6) Click on 'Conversion Tool' link.

Expected Result:

    The Converter should appear.

Actual Result:

    Clicking the link results in nothing. Converter doesn't
    appear.
```

2.5 Summary

A FOSS project is distinguished by a development methodology that combines agile techniques with high levels of interaction among developers and users. FOSS development includes not only coding and debugging, but also communicating, writing documentation, and submitting bug reports.

Two genres of FOSS projects, called "client-oriented" and "community-oriented" projects, are also distinguished. Both share common methodologies, such as agile development and user involvement.

However, the two have different development paradigms because of their different maturity levels, user base size, and diversity of their developer communities. Working on a client-oriented project requires a personal level of communication with team members and users, while joining a community-oriented project requires initiating a different collection of communication techniques and activities.

Exercises

The exercises below are aimed at facilitating the team-formation and team-joining activities for a FOSS development project that can be carried out throughout the remainder of a one-semester course. The first three exercises

are appropriate for beginning work on a client-oriented project, while the second three are suited to joining an ongoing community-oriented project.

2.1 Read the requirements document and initial code base for a client-oriented FOSS project that your team will be conducting in this course.

- **a.** Identify your teammates and the role that each one will play in the project.
- **b.** Evaluate the condition of the code base. In your view, is it well written and documented? What needs to be done to improve its readability and extendability to new functionality?
- **c.** Evaluate the requirements document. Does it clearly state the objectives of your project? Does it relate well to the code base, in the sense that its use cases relate clearly to the organization of the code base?

2.2 Participate in the first team meeting for your client-oriented project. There, an initial task assignment will be given.

- **a.** Evaluate and discuss your preparation and willingness to perform the tasks that are assigned to you. Feel free to suggest changes appropriately.
- **b.** If you have questions about the requirements document, ask users to clarify those questions.
- **c.** If you have questions about the code base, ask the project leader to clarify those questions.

2.3 Complete the initial task(s) assigned to you for your client-oriented project. In this exercise, you should be comfortable collaborating with teammates as needed. Communicate your progress and completion of your task(s) to all teammates on or before the due date for your task(s).

2.4 Visit http://sourceforge.net and find five active, vibrant FOSS projects and five moribund ones. Explain how you came to each of these assessments.

2.5 Revisit http://sourceforge.net and locate two interesting community-oriented projects.

- **a.** Compare the two with regard to their apparent vitality, receptiveness to newcomers, and compatibility with your particular programming and other development skills.
- **b.** Visit each one's IRC channel and observe the conversations that are taking place among developers and users. Do the conversations seem to be welcoming? Do the posts seem to be constructive?

c. Browse the source code for each project and make a brief assessment of how well organized and documented it is.

2.6 Select a community-oriented project and explore the current issues being discussed on its IRC channels (developer and user forums) and email archives. If necessary, take appropriate steps to join the project by obtaining a user id and password. Read the user documentation and, if possible, exercise an on-line test version of the software.