Bridging the Academia-Industry Gap in Software Engineering: A Client-Oriented Open Source Software Projects Course

Bonnie K. MacKellar
St. John's University, USA

Mihaela Sabin
University of New Hampshire, USA

Allen B. Tucker
Bowdoin College, USA

ABSTRACT

Too often, computer science programs offer a software engineering course that emphasizes concepts, principles, and practical techniques, but fails to engage students in real-world software experiences. We have developed an approach to teaching undergraduate software engineering courses that integrates client-oriented project development and open source development practice. We call this approach the *client-oriented open source software (CO-FOSS)* model. The advantages of this approach are that students are involved directly with a client, nonprofits gain a useful software application, and the project is available as open source for other students or organizations to extend and adapt. This chapter describes our motivation, elaborates on our approach, and presents our results in substantial detail. As we shall show, the process is agile and the development framework is transferrable to other one-semester software engineering courses in a wide range of institutions.

MOTIVATION

Most computer science programs offer a software engineering course and view it as a critical link in ensuring the career-readiness of computer science graduates. However, too often this course is taught in terms of abstract principles, failing to engage students in real-world software experiences. Many of the skills required in industry are best learned by hands-on practice, such as the need for effective communication among developers, or the need to interact with a non-technical client. Thus, students who have never engaged in a hands-on project in software engineering enter the workforce with gaps in their skills.

It is, however, difficult to bring a significant software development experience into the confines of a one-semester course in academia. The most common approach has been to introduce a "toy project", which is a small project designed by the instructor, and have students work in teams to complete the project by the end of the semester. The advantage of this approach is that students will ideally learn to work in teams and share responsibility for developing a codebase. The disadvantages are that the project may be oversimplified, and students gain no experience interacting with clients or with code written by others.

Another approach is to work with local companies in the private sector who sponsor *proprietary client-oriented software projects*. This has been used successfully by a number of schools, especially larger programs that already have established linkages with companies (Judith, Bair, & Börstler, 2003; Tadayon, 2004; Tan & Jones, 2008). Another setting that favors this approach is an internship course with the projects being developed onsite at local companies. The advantage is that students gain experience with real clients with high stakes in real projects. However, these projects are often standalone, one-off projects since companies may be reluctant to have students work on their internal codebase, or to develop mission critical software. This means that it may be difficult to get enough time and attention from personnel at the company while the students work on the project. Also, the project will normally become the property of the company, meaning that it cannot be freely shared with other schools trying to adopt a similar approach.

A third approach is to engage students in Free and Open Source Software (FOSS) development by having them contribute to *a large and active open source project*, such as Linux or Mozilla (Marmorstein, 2011; Ellis, Morelli, DeLanerolle, & Hislop, 2007). The advantage of this approach is that instructors and students can gain from the mentoring achieved through communication with the project's professional developers, and in some cases they contribute marginally to the "live" code base or the user documentation. The disadvantages of this approach are that most ongoing projects are large and complex, their developers may not be accessible, and given the time it takes to come up to speed in the project, students may gain little practical experience in a one-semester course.

A fourth approach, which occupies a middle ground between the proprietary client-oriented project model and the full-scale FOSS project, is to engage students in FOSS development via a relatively small project that fits in a one semester course, with a local nonprofit organization as the client. Local nonprofits are often happy to collaborate on these projects since they may have needs for mission-critical software systems that are not well met by the commercial software industry, yet they have limited technology budgets. Thus, it is relatively easy for an instructor to locate and collaborate with a local nonprofit. However, many instructors may still be unsure of how to get started or how to organize such a course.

This chapter describes our collective experience with the fourth approach, which we call *client-oriented free and open source software development* (CO-FOSS). The big advantage of treating client-oriented open source projects is the very openness of the project. An open source project developed in the context of one course for one client can be reused, extended, and adapted for new clients by subsequent iterations of the same course, or even by courses at different institutions. By providing not just the codebase but the course organization itself as an open source project, a collection of such projects can be built up to be used as models at different institutions. In addition, the tools and practices of open source projects provide a readymade infrastructure for software project courses.

Service learning projects for nonprofits have been used in a number of other software engineering courses and have been reported in these papers (Olsen, 2008; Poger & Bailie, 2006; Venkatagiri, 2006); however, these projects have not taken advantage of the reusable and extendable capabilities of the CO-FOSS approach. An example of a service learning approach that has been turned into a model for other universities is EPICS (Coyle, Jamieson & Oakes, 2005). This was developed for engineering schools and thus is not specific to computer science. In the EPICS model, interdisciplinary teams of eight to twenty students are assembled to work with community organizations on engineering problems. The teams are vertically integrated, consisting of freshmen through seniors. This model has been expanded to a number of universities and an implementation of it in a software engineering context was reported in (Linos, Herman & Lally, 2003). EPICS is similar to our model in that it provides a structure for community-oriented service learning projects, providing advice on team formation, identifying clients, structuring communication, and so on. However, it differs from our approach because it does not leverage the power of open source development, meaning that it is more similar to the proprietary client-oriented model in many ways. In addition, the vertically integrated teams are a challenge to integrate into many computer science programs.

However, the CO-FOSS approach has some challenges. It may be difficult for an instructor who has never done this type of project before to develop the course, to manage the project, and to attend to the post-course activities that accompany placing the new software into production. Therefore, we will present a framework for developing this type of course. In the CO-FOSS approach, one of the most important goals is that the students be able to actually complete a working prototype within the boundaries of a semester. To this end, the approach is highly structured, with weekly goals for completion. The instructor (rather than the students) develops the requirements. Since CO-FOSS projects are open source, architectures and code can be reused and leveraged into new projects. And because open source projects are visible to the world, students are motivated to achieve a higher level of quality. The openness of the course artifacts also means that instructors in other institutions can easily adapt existing projects to meet their clients' needs. Allen Tucker first developed this framework as part of the Humanitarian FOSS project (Morelli, de Lanerolle, & Tucker, 2012) and has taught numerous iterations of a software engineering course this way. Bonnie MacKellar and Mihaela Sabin adapted this framework to courses at their universities by taking into account students' characteristics that are specific to their institutions (MacKellar, Sabin, and Tucker, 2013). The result is a model for teaching software engineering that brings real world experience to a wide variety of institutions. In this chapter, we will present this framework, our experiences with adaptation, and a set of guidelines and best practices that instructors can use to integrate CO-FOSS development into their courses.

INTRODUCTION TO THE CO-FOSS MODEL

The CO-FOSS model has two major elements: 1) a *process* and 2) a *product*. The process is agile, participatory, and open. The product is real, functional, and useful to the mission-critical operations of the nonprofit client. This section elaborates each of these elements in turn.

The *process* of CO-FOSS development that we advocate is carried out by a team of developers, which necessarily includes a client representative who understands the manual activities that the software will replace. Because the process is agile, the client has frequent (weekly) opportunities to interact with each element of the software as it comes on-line and to provide feedback to the developers on which features work well and which do not. In turn, the developers can take that feedback into account as they refine and extend the software during the following week.

The team leader is typically the course instructor, who evaluates the work of the student developers each week and takes client feedback into account while preparing the following week's assignment. All of this activity is facilitated by a weekly meeting, either in-person or through an interactive video conference, where the developers demonstrate and explain their work to the client, the client provides feedback, and the instructor takes notes that inform the project's next steps.

The *product* of a CO-FOSS project begins with a complete requirements document and an initial codebase. The instructor must complete the requirements step before the beginning of the semester. This is a departure from the usual software engineering course, where students often work on requirements as well as software development. An alternative is a two-semester software development course, in which requirements elicitation and engineering concepts and practices are taught in the first semester, prior to designing and implementing the software in the second. The initial codebase may be the result of a prior similar open source development project, or it may consist only of an initial set of domain classes springing out of the requirements document. The final product is a real and viable software artifact, which fully implements the requirements that had been laid out at the beginning of the project, and which may be extended and adapted by other students for different clients in the future.

Our experience with nonprofits as clients for open source projects like this suggests that many nonprofits exhibit similar software needs that are not particularly well met by commercially available software at a price that the nonprofits can afford. For example, many nonprofits use numerous volunteers to help realize their mission. These nonprofits need software that assists them with scheduling volunteers into calendar shifts using the idea of a master schedule. Volunteers typically like to use simple repeating patterns (like "every other Thursday in the 12-3 shift" or "the first Monday of each month from 8-12") but commercially-available calendaring tools do not support this sort of master scheduling in an easily customizable way. Thus, *Homebase* was initially developed in 2008 to support the volunteer scheduling needs of the Ronald McDonald House in Portland, ME. Since then, the *Homebase* design and code has been reused and adapted for other clients that have similar volunteer scheduling needs: the Ronald McDonald House in Wilmington, DE, the Second Helping food rescue organization in South Carolina, and the Mid-Coast Hunger Prevention Program in Brunswick, ME. Because it is open source, *Homebase* can evolve over time and can be easily adapted and reused for new projects with similar needs. We have found that reusing the software architecture and underlying code for a successful project can greatly simplify an instructor's task of developing a new project for a different client.

Moving to a more detailed level of granularity, we find that the process of developing and delivering a one-semester CO-FOSS course can be divided into an eight-step framework:

1. Pre-course activity
2. Curriculum design: syllabus and milestones
3. Structuring client communication
4. Team formation and task assignment
5. Developer communication and code sharing
6. Writing user and developer documentation
7. Evaluation of team members' contributions
8. Post-course activity

The following section explains each step in this framework.

STEPS TO DEVELOPING AND DELIVERING A CO-FOSS COURSE

Pre-course Activity

Client Sponsorship

Preparing a software development course so that students can have a real-life collaborative experience with a client and develop a useful software product during the course of a single semester requires significant effort on the part of the instructor. First, the instructor must find a willing nonprofit client and identify a specific software project. Professional or personal associations with people who are familiar with the day-to-day operation of the nonprofit (such as the executive director, operations manager, or a particularly active board member) can be very helpful in this effort. If the college has a service learning office, personnel there will often have contacts with local nonprofits. The local reputation of the nonprofit in the community can also provide leads. If the instructor has already worked with a CO-FOSS project for another client or wants to work with an existing CO-FOSS project from another school, it is best to locate nonprofits with similar needs or in a similar sector. The existence of a similar successful project for a previous client often heightens interest within a targeted nonprofit client. Once a client is located, the instructor must identify a specific employee who is responsible for the operations which the software can enhance, excited about helping with the development of such software, and able to dedicate time (a few hours before the semester begins and an hour a week throughout the semester itself) to work with the instructor and the student development team as they design and develop the software itself.

It is very important that the project be designed to ensure success. That is, the most important goal of the course should be that the students actually complete a working prototype for the software within the 13-week boundaries of a semester. This goal usually means that the instructor and the client representative make some hard choices about what will and will not be included in the final product at the end of the semester before the project begins. These decisions should inform the next step of the pre-course activity, that is, requirements elicitation and documentation.

Requirements Document

The instructor must work with the client to elicit requirements and tailor a project that both serves the needs of the nonprofit and can be completed within the context of a normal semester course by a team of students. The result of this activity will be a requirements document that spells out in substantial detail the functional, technical, and user interface requirements of the software, as well as the eventual use to which it will be put when it is completed. The requirements document must provide enough detailed domain-specific information so that students can identify initial development tasks almost from the get-go. This information should include, for example, initial domain classes for which representative attributes have been identified; a selection of programming and database languages appropriate to the project; and an overall architecture for the software so that students will clearly understand where each of their modules will fit within the larger product. Links to examples of our requirements documents can be found in the Resources section at the end of the chapter. Typically, a layered architecture that separates the user interface from persistent data representation and manipulation provides a good starting point for many web-based applications.

While the idea of providing the requirements document to the students may seem surprising since many traditional software engineering courses involve the students in writing requirements as well as developing the software, we find that undergraduate students typically do not have the experience necessary for writing adequate requirements for a client. They either become bogged down in the task, leaving too little time for development, or they produce a requirements document that ends up having little to do with the final system. This has also been noted by Cheng and Lin (2010), who have developed a guided approach to the software engineering class project that is similar to ours, but without the client and open source focus. In their approach, the instructor also develops specifications and an overall design.

On the other hand, the presentation of a requirements document at the beginning of the course does not limit student-client interaction or student participation in requirements development. In the first case, students in this course interact weekly with the client to gain clarity on the details of requirements so that they can develop appropriate code for the weekly assignment. In the second case, the fact that the whole process is agile ensures that requirements evolve alongside the coding. So students are in fact first-class participants in the refinement of requirements, even though they do not develop the initial version of the requirements document.

Curriculum Design: Syllabus and Milestones

The syllabus can be structured in a way similar to that of other computer science courses. That is, each week of the semester, students are expected to complete a specific set of learning objectives and demonstrate their learning by completing an assignment that reflects those objectives. But there are two main differences: 1) the learning objectives and assignment are drawn directly from the requirements document, and 2) students normally must collaborate and share a common code base to complete the assignment.

The first week or two in the semester are dedicated to team formation and setting up for collaboration. Each team must initialize a shared code repository for the project using a designated version control system, such as Subversion, GIT or Mercurial. Each student must also set up his/her own computer with a development environment that supports the programming tools chosen for the project. That environment must also be interfaced with the shared code base, so that each student can integrate his/her own work with teammates' work. This start-up step is not trivial for the student team, since students are not generally familiar with code sharing from earlier courses. Depending on the particular version control system and development environment chosen, sufficient tutorial materials should be made available to students so that they can independently set up their own computers to work effectively in the course. Links to supporting materials for Mercurial and Eclipse, for instance, can be found in the Resources section.

During each subsequent week in the course, each student team member develops, unit tests, and commits a new piece of the software to the repository. These weekly assignments are determined by the instructor, who takes into account client feedback from the previous week's assignment. The layered architecture of the software design facilitates task sequencing. That is, when the domain model, user interface, and database controller layers are distinguished, they can be helpful in organizing the weekly assignments in a course. In this case, it is often preferable to design and unit test the domain classes in weeks 3-5, the database modules in weeks 6-8, and the user interface modules in weeks 9-11 of the semester. That leaves weeks 12-13 free for students to perform more detailed integration testing, develop user documentation, and address other special circumstances that inevitably arise during the semester. The final exam date for the course provides a good opportunity for the student team to deliver the completed software prototype to the client in the setting of an oral presentation. Often, other members of the client organization are invited to this presentation, such as the executive director, other staff members, or even some board members.

Structuring Client Communication

As should be evident from the above discussion, client feedback is essential as each weekly assignment is completed and demonstrated. Students need feedback to determine the extent to which their coding efforts successfully addressed the assigned task. The instructor needs feedback to help assess the project's overall progress and to determine in detail the shape of the following week's assignment.

It is not practical for the client to meet face-to-face with the student team members at each weekly class session. Moreover, it may not even be practical for all the student team members or the instructor to be physically present at each team meeting. On the other hand, "virtual attendance" at each team meeting should be required of all students and the instructor and the client representative. Virtual weekly attendance among all team members can be ensured if everyone uses a visual teleconferencing medium, such as Skype or Google Hangout, to facilitate it.

The second ingredient to ensure client feedback at weekly team meetings is access to a shared "sandbox" server that can be used to demonstrate the current state of student progress with the project. This server can be provided by either the university or an independent Internet service provider. The important point is that at each team meeting, the client can actively view and work with the partially-developed software under the verbal guidance of the students who are developing it.

The dynamics of the weekly team meeting among the students, the instructor, and the client representative are as follows. Each student on the team presents to the client a brief description of what he/she accomplished during the prior week, and shows the client how that works. The client tries it out using the "sandbox" server and provides feedback on the spot. The instructor takes notes and uses the presentation and the feedback as factors in designing the following week's assignment. Typically, a week's assignment has two parts – one part that cleans up issues identified by the client in the team meeting and the other part that makes progress developing some additional aspect of the software. At the end of the development period, a more or less complete prototype will thus emerge. Critically, the software prototype that results from this process is not the product of the student developers working in a vacuum with the original requirements document. Rather, it is the product of weekly client feedback and immediate adjustment to that feedback. This is the essence of the term "agile development," brought to life within the setting of a real software project.

Team Formation and Task Assignment

Team formation and task assignment constitute one of the key tasks when designing this type of course.
The task of team formation involves some key decisions:

1. How large will each team be?
2. What tasks will be assigned to each team?

3. Who will be assigned to each team? Who makes that decision?

Team Size

There are many factors beyond the instructor's control that impact these decisions. For example, student preparedness and the size of the client project will impact the size of the teams. The number of enrolled students, as well as the number and scope of available projects will also have an impact. However, the framework for CO-FOSS projects is flexible enough to incorporate varying team structures. For example, at Bowdoin College, there were several projects, and well-prepared students, so teams were structured by project. At St John's University, on the other hand, the instructor had already been using a way of structuring the course so that all students work on one project. This works well in that environment because the students are less well-prepared and can specialize in areas where they feel most comfortable, such as writing the help system, developing the database, or working on the user interface. At UNH Manchester, components of the project are partially developed in other courses. For example, students learn database model, design, and implementation techniques or how to integrate a web-based user interface with database services in the Database Design and Development and Advanced Web Authoring courses.

Task Assignment

The architecture of the system plays a big role in the structure of the teams and the tasks they are assigned. This is another decision point that is eased by following the practice of building upon an existing open source project from past semesters. The *Homebase* architecture consists of a database, domain objects, a help system, and user interface. Thus, it is clear that students will be allocated to each of these tasks. If smaller teams are working at multiple smaller projects, and are following the Bowdoin model of focusing on separate layers at different points in the semester, then each student will end up working on each layer. This means that all students will need to gain expertise in every aspect of the system. This works best when all students are very well prepared for the course, and have seen topics such as database design and SQL before taking this course.

An alternative structure, used at St John's and UNH Manchester is to use the entire class as a team if the project scope is large enough. In this organization, the large team is split into subgroups along system architecture lines. If students are not familiar in advance with many of the project technologies, this organization has the advantage that students do not have to learn many new technologies at once. Also, students can be grouped according to their strengths. For example, the group working on the database layer can be composed of students who have already taken the database course. The advantages of this type of team organization are detailed in (MacKellar 2011). The disadvantage is that some layers depend on other layers. This means that interfaces between layers must be carefully specified and adhered to, and stubs for testing purposes may need to be developed.

Allocating students to teams

The CO-FOSS framework does not impose many constraints on the task of assigning students to teams, so the instructor has several choices to make in forming the teams. There has been quite a bit of research, both within computer science and in other disciplines, on the best way to form teams for group projects. Richards (2009) detailed many of the considerations in a survey. For example, students may collaborate more effectively if they are allowed to choose their own teams (Grundy, 1996) but when this is permitted, the more capable students are likely to end up together in one group. If it is important to distribute students of different abilities across the groups, then the instructor should assign students to groups. Criteria such as GPA in past courses or surveys of student can be used to determine placement (McConnell, 2006). Other criteria, such as gender, ethnicity, or simple time availability may come into play as well. Even issue tickets have been used as a way to form teams (Coppit and Haddox-Schatz, 2005). If the teams will consist of students working on all aspects of their project, as detailed above, then either student-based or instructor-based assignment can be used, and any of the various considerations can be used.

If the students will be working in larger teams, and assigned to specific components in the system architecture, then the instructor should do the assignment, and take their skills and interests into consideration. At St John's, the students are surveyed at the beginning of the semester, and also submit resumes. The instructor then assigns students to task groups in a way that minimizes project risks, much as a project manager in industry does. Thus, a student who indicates experience with SQL will be assigned to work on the database layer, whereas a student who has used PHP in the past will be assigned to the user interface layer, and a student who is familiar with QA, perhaps from an internship, will be assigned to the testing group. At UNH Manchester, the instructor solicits input from students in the first class about their academic and professional experiences, self-reported computing strengths, and areas of interests in their future careers. Two other sources of information are taken into consideration: student transcripts and evaluation from faculty members who know the students from their classes. Gathering this information is possible in a small department with less than 100 majors and a climate with frequent and meaningful interactions among faculty. This method of assigning development roles to students by the instructor only emphasizes learning to work in teams on large software systems rather than learning an array of specific technologies, such as PHP and SQL. The method is also inclusive of various talents and interests and avoids having students distracted by inexperience with a specific technology.

Developer Communication and Code Sharing

One of the advantages of approaching the software engineering course as an open source project is that open source development comes with a set of standard practices and tools which tend to work very well for students. The tools themselves are open source, so they are affordable in an academic setting. Open source work practices, which evolved from the need to support a highly asynchronous, distributed group of developers, also work well for students because the practices do not require as much face-to-face development effort.

What are these tools and practices? In general, most open source projects use the following tools (Fogel, 2005):

- A project repository
- Mechanisms for team communications, usually mailing lists and real time chat channels
- A version control system
- An issue tracking system

Since there are so many open source projects, a number of standard sites and tools have appeared to meet these needs. These sites constitute a readymade toolbox for software engineering educators, greatly easing their task since they are already set up and integrated.

The project repository

Open source projects, like any other software project, need to reside somewhere. Not just the code, but design documents, installation instructions, build sequences, and records of defects must be maintained. Open source projects, however, are not usually tied to any one organization and must be accessible to developers and other contributors on a worldwide basis. Thus, quite a bit of effort has gone into building sites that allow sharing of project artifacts. These sites are referred to as project repositories, and often contain a number of related features, such as version control, bug tracking, and wikis. Examples of currently popular project repositories include SourceForge, Google Code, and GitHub. These repositories are free for open source projects. Obviously, the fact that they are free is very appealing for cash strapped universities. But they also are convenient for a number of other reasons. Since version control and bug tracking are typically integrated, the instructor is freed from needing to install and maintain complex software. Since they are designed to be shareable among distributed contributors, repositories allow students who may not be able to attend face-to-face meetings to still collaborate. And since these repositories host many projects, including some very famous ones, students can browse the repository site, see lots of interesting projects, and see that people much like themselves are contributing. This can add to their sense of connectedness with the discipline of software engineering and the open source development community.

Version control systems are critical to projects that require more than one developer to work on code in a controlled fashion. There are a number of version control systems in common use. Interestingly, most of them are open source systems. Examples include CVS, Subversion, Mercurial, and Git. A project repository will offer one or more of these systems, and the instructor, when setting up the project, must choose one. Using a project repository forces the students to use the version control system since they must interact with it to place their code into the repository.

Another typical feature of open source projects is the use of an online issue tracking (or bug tracking) system. These are used extensively in industry as well. In the open source world, these systems are usually integrated into the open source project repositories. For example, the issue tracker on Google Code integrates with Google Groups, which is an online discussion system, so that when an issue is entered, a message will automatically go out to the discussion group for the project, and when the issue is resolved, another message will be generated. This is easy for an instructor to configure, and helps ensure that everyone in the class is aware of the current status of all bugs in the software.

Online messaging systems

A characteristic of open source projects is that developers are distributed geographically, and are not likely to work on the project during set business hours. This means that tools to support asynchronous, online communication are critical to the success of the project. Very commonly, mailing lists are used. As mentioned above, support for such mailing lists may be integrated into the project repository. Mailing lists allow developers who are both geographically and temporally separated to maintain a conversation. Mailing lists are superior to regular email because they are topic-specific and they allow all of the developers to stay in the loop. Another tool that is used to support synchronous conversations is Internet Relay Chat (IRC), although many projects also use free videoconferencing tools such as Skype and Google Hangouts.

There are some significant advantages to using online messaging systems in a software engineering course. First, using these tools means that students do not have to hold as many face to face meetings, which can be a huge problem for non-traditional students with family commitments, commuter students, and even residential students who are taking a heavy course load. Students also appreciate the ability to search the conversations on the mailing list, making it less necessary to take careful meeting notes. And finally, mailing lists are very useful from the instructor's point of view, because he or she can monitor the conversations, getting a better idea of where the students are having trouble, which groups are not communicating very well, and what the various project statuses are. It is even possible to mine the conversations to create more careful analyses of student conversations (MacKellar, 2013).

Writing Developer and User Documentation

There are two types of writing that software developers usually engage in: writing documents aimed at other developers, and writing documents aimed at users. Software engineering courses usually try to have students engage in both types of writing. A very common set of documents produced during a traditional course would include a requirements document, a system design, a user manual or help system, and code level documentation. This follows the needs of the standard waterfall method, but in a one-semester software engineering course, this also can be very rushed. It is not clear how well students learn to write a requirements document in the two or so weeks that are typically devoted to the process in a traditional one-semester course.

In the CO-FOSS model, the requirements are already developed before the course starts, and if the course project is an iteration of earlier work, the architecture is also already defined. Thus, the bulk of the writing in the course is concentrated on two sets of artifacts: technical documentation aimed at other developers, and user documentation. The technical documentation consists of system design documents in the form of use cases and class designs, as well as comments in the code itself. The user documentation consists of an online help system.

Generally, open source projects have had a reputation of being poorly documented both in terms of developer-oriented documentation and user-oriented help systems and manuals (Madsen & Nürnberg, 2005; Meneely, Williams & Gehringer, 2008). This is not a desirable outcome for a software engineering course or for a client-oriented project, particularly when the clients are small nonprofits where there may not be a lot of technical expertise. This is an area where the instructor's guidance is vital to establishing effective standards for documentation. The client's feedback is also very important to help students develop a good help system, since most students have never written about or even thought about their software from a client perspective before.

Because of the open source nature of the CO-FOSS model, all artifacts are publicly available. This means that other students as well as the client can access, comment on, and even improve the documentation as needed. It also encourages the students writing the documents to concentrate on quality, since they know that other people will see their work. The code repository consists of an introductory page explaining the point of the project. The repository can also have a project wiki, which contains information of use to developers and users, such as how to install the software, and possibly instructions on writing new code modules. The use cases and class design are posted on the code repository, either in the wiki or as a downloadable document. Most open source repositories also contain an issue tracker and a discussion forum, and even those can be seen as forms of documentation.

Since students have usually not done a lot of technical writing before this course, they require a lot of guidance, structure, and examples when completing this part of the project. Students can be directed to other CO-FOSS student projects to find examples of good writing. The instructor can discuss these projects with students and point out ways in which the writing is effective or not effective. Even more powerfully, as a given client oriented open source project evolves over time, students will be able to work with existing help systems and design documents, and to use these as templates for their current project. Once an instructor has established a consistent framework for the open source project, it can be reused again and again, and students will be able to see a wide variety of examples, discovering what works and what does not work. For an example of an on-line help system developed by students, see rmh.myopensoftware.org and login as Admin1112345678 (same password) and hit the **help** tab.

Evaluation of Team Members' Contributions

Evaluating student contributions in team projects is one of the thorniest issues encountered by instructors of project-based courses. There is a large body of literature, and many competing ideas, on how to do this task effectively. The most common approaches are 1) individual grades based on peer evaluations 2) individual grades based on instructor evaluations 3) a group grade that is assigned to all group members based on project success 4) mixed approaches incorporating the previous three approaches in various ways.

One of the problems that makes it difficult to evaluate group projects is that individual contributions need to be measured to avoid the “free-loader” effect: that is, students who fail to make any meaningful contribution towards the project but who end up passing because they share a group grade. Research has found that students prefer individual grades in order to prevent the free-loader effect (Farrell, Ravalli, and Farrell, 2012). Thus, it is common to assess students with a mixture of approaches; assigning a group grade as well as an individual grade, weighting each in some fashion. Farrell finds that the group grade is often determined objectively (how many project objectives were achieved while the individual grade is often determined subjectively (how much of a contribution did a particular student make?).

At Bowdoin College, the project grade measured the degree to which the team completed all the project milestones. Individual grades were assigned in relation to this project grade in order to recognize differences among different students’ contributions. At St John’s University, students received individualized grades that took into account the overall group success, the students contribution to the group effort as measured by proportion of code written and activity at meetings and on the message board, and the quality of the student’s code. At UNH Manchester students also received individualized grades that measured the degree to which assigned tasks were successfully completed. These tasks included: work on the software system artifacts (whether code or documentation), documenting the development process (team meeting agendas and minutes, participation on the team’s reporting on status of the team’s artifacts, and writing self-evaluations to reflect on progress and challenges with each student’s individual contribution.

Post-course Activity

A significant challenge to CO-FOSS projects is to ensure that the resulting software artifact is delivered and supported in a timely way to the nonprofit organization that helped develop it. This is a challenge because, just as in the case of pre-course CO-FOSS activities, this activity falls outside the normal expectations of a university faculty member.

Our experience in meeting this challenge is varied. At Bowdoin, for example, the instructor has dedicated his own time to ensuring that the software is delivered and properly supported through the first several months of its use. The quality of the software artifacts developed so far has been so high (in the case of Homebase and Homeplate, for example – see discussion in the next section) that the need for ongoing software support, once it is put into use at the nonprofit, has been minimal. At St John’s and UNH Manchester, instructors have also devoted significant time to bringing the system up to production standards.

In the long run, the need for ongoing support for a successful CO-FOSS project can be met by the establishment of a partnership between the non-profit and a local software firm that has the capability and interest to provide that support on a cost-effective basis. For each of the Homebase and Homeplate projects, the non-profit has partnered with a local software firm that provides support at a reasonable cost.

We recognize that sustainability is a major challenge for any community service project involving student developers, simply because at the end of the semester both the students and the instructor typically move on to other courses and priorities. On the other hand, it is mandatory to the integrity of the CO-FOSS model that a support structure be put into place that facilitates the creation of client-software firm partnerships once a semester project is completed and the software is ready for deployment.

To this end, the authors are participating in the establishment of a new organization called the Non-Profit FOSS Institute (NPFI). The purpose of the NPFI is to facilitate the planning, creation, execution, and ongoing support for new CO-FOSS projects like the ones discussed in this chapter. Its 13-person Advisory Board represents all three types of participants in such projects – non-profits, instructors, and software firms. The Board members all have significant experience with, and enthusiasm for, developing

CO-FOSS at a national level.

Key to the success of the NPFI is to facilitate and support the formation of "triads," each triad having an instructor, a non-profit client, and a local software firm. Facilitation includes supplying support materials – code bases, requirements document examples and templates, and other teaching materials that will help an instructor get started with a new CO-FOSS course. Because NPFI is in an early stage of development, we cannot say much more about it at this writing. We can say that in the spring of 2014, the Board plans to launch the http://npfi.org web site, which will provide many more details about the mission and organization of the NPFI. Readers who are interested in becoming associated with NPFI in the future are encouraged to visit this web site and become a member. NPFI membership will be open to all instructors, non-profits, software firms and others who embrace the CO-FOSS model.

CASE STUDIES: THE EXPERIENCE AT THREE SCHOOLS

The three authors have significant experience in adapting the CO-FOSS framework to different institutional and nonprofit environments. Overall, we have found this framework to be robust in the sense that it is adaptable to a variety of academic and nonprofit settings. However, an instructor wishing to use this framework in his/her own course must take into account local differences in student population, institutional support, and nonprofit availability and willingness to participate in such a project. The case studies in this section detail how the three authors customized the model to meet their local needs.

Case study 1: *Homebase*

Developing the framework with a client-server application at a small private college

Bowdoin College is a selective residential liberal arts college with a fully-developed Computer Science Department, averaging about 10 majors per class. Students are enrolled full-time at the College, and computer science majors typically take four courses per semester. Many majors also choose to double major with mathematics, economics, or one of the sciences. At the time students enroll in the software development course, they are typically juniors and have taken a significant number of computer science courses, including data structures and algorithms, and have done a good deal of programming in different languages.

In Spring, 2008 Allen Tucker developed and taught the software course using the CO-FOSS approach for the first time. Inspired by the H-FOSS model (Morelli, DeLanerolle, and Tucker, 2012), this course aimed to develop on-line volunteer scheduling software for the Ronald McDonald House in Portland, Maine. Four brave students enrolled in and completed the course, all seniors (three CS majors and one economics/math major who had a lot of programming experience). The software that they completed was dubbed "Homebase" by the RMH staff at some point during the semester.

Prior to the beginning of the semester, the instructor met with RMH staff to gather information that would contribute to a requirements document for the *Homebase* project. The requirements document included a description of the then-current manual volunteer calendar scheduling process, a description of the methodology and tools that would be used to develop the software, a few screen-shot sketches of the desired user interface, and a time-line of milestones that had to be met to complete the project by the end of the semester.

The goal was to develop a complete working prototype for *Homebase*, including on-line user help and a week of training. The development team included the four students who would do the major programming, the instructor who would manage the project by giving weekly assignments and overseeing the sandbox server, and a client representative who would test the partially-completed software and provide weekly feedback to the students. Throughout the semester, each milestone was adjusted as a result of the weekly team meeting and feedback from the client. By the end of the semester, the client knew exactly what she was receiving as a software tool, making this a truly agile process.

The software architecture for *Homebase* may be its most important characteristic as a model for other CO-FOSS projects that use a similar teaching strategy. The client-server architecture can be viewed as having layers – the domain classes, the database modules, the user interface modules, and the user help modules. Confined by the limits of a single semester, the course can naturally flow by developing these four layers in order as a series of 3-week chunks, beginning with students developing and unit-testing the domain classes identified in the design document. This initial chunk breeds a vocabulary of terms that can be shared unambiguously between the students and the client – in the case of *Homebase*, terms like "Shift", "Week", and "Volunteer Availability" take on specific meanings that facilitate communication throughout the remainder of the development process.

The outcome of this project was a fully functional online volunteer scheduling module that integrated with the RMH website and replaced the manual scheduling system during the summer of 2008. The instructor spent significant effort during that summer making the software "bullet-proof" so that it ran reliably and correctly on a 24/7 basis. Volunteers and House staff uniformly praised the software for its ease of use, security, and accessibility from anywhere there was a Web browser. The students completed this course knowing that they had not only experienced a real-world software development task but also made a significant service learning contribution. Since its completion in 2008, *Homebase* has been used as a starting point for similar software development activities, both at Bowdoin and at other universities and has been updated and expanded by two more teams of Bowdoin students in Spring 2012 and Fall 2013. A link to the current version of *Homebase* and its requirements document is listed in the Resources section at the end of this chapter.

Case study 2: *Homeplate*

Adapting the framework for a mobile computing application.

The CO-FOSS framework discussed above was used again at Bowdoin College in 2011 to develop a room scheduling module called *Homeroom*, and again in 2012 to develop a module called *Homeplate* for volunteers to record pickups and drop-offs for a food rescue and distribution organization called Second Helpings, in South Carolina. The *Homeplate* project was significant because it used the CO-FOSS framework for developing the domain, database, and user interface layers for an easy-to-use Web-based client-server application. But it was also significant because it provided a server-side platform with which an independent Android application could later be developed and deployed.

The server side of *Homeplate* was developed by a team of three Bowdoin students during the Spring 2012 semester, and deployed soon after the end of the semester. The *Homeplate* software architecture was essentially the same as that of *Homebase*, and we reused several key modules of *Homebase* in this new project. During the summer, one student worked with the instructor and the client to develop the Android tablet app that could be carried on the Second Helpings trucks themselves. This app facilitated volunteers' recording of food weights at the time the food was being picked up and dropped off from the trucks. To attempt this Android app as part of the semester project alongside the *Homeplate* software would have created both conceptual and practical overload. In general, it is always important for an instructor launching a CO-FOSS project to assess what can be accomplished in a semester, and trim the project appropriately in a way that ensures student success with a high probability.

The Android app runs on a tablet and sends and receives scheduling and weights data from the *Homeplate* server via FTP when the tablet is in a free wi-fi zone. This strategy avoids requiring Second Helpings to purchase expensive data plans to accompany their tablets, which would be a deal-breaker for most charitable nonprofits. The Android app was developed on a Java-like platform, which is provided freely through developer.android.com. Lots of tutorials about Android development are freely available on the Web, so that the student, the instructor, and the client representative were able to fully develop and deploy this enhancement during an 8-week period in the summer of 2012.

The Android app was deployed by Second Helpings on all 5 of their trucks in September of 2012, and has been running successfully ever since then. Volunteers (most of whom are retirees) remark that this app is extremely intuitive and easy to use. The success of *Homeplate* and its accompanying Android app has recently been reported in local a local newspaper article (Bredeson 2013) which highlights the essential role that students played in their success. Instructors interested in more details about *Homeplate* design, development, and source code downloads can access the *Homeplate* link in the Resources section.

Case study 3: *RMHRoomReservationMaker*

Adapting the framework at an urban university with students of diverse backgrounds and abilities.

St John's University (SJU) is a large, urban university whose students are ethnically diverse; a large percentage are Pell-eligible and first in their family to attend college. Many students work off campus or have significant family responsibilities. Software engineering is a required course in the computer science major. Students enroll in the software engineering course with various backgrounds – some have done little programming and others know quite a bit. In order to meet the needs of this diverse group of students, we usually center the software engineering course around one larger project, on which the entire class collaborates. This organization, which is described in more detail in (MacKellar, 2011), is based on the idea that students work in smaller groups organized around a project role, such as testing, development of the database, or help system development. This is a common organization in real world software projects.

In previous years, the project for the course was always a "toy project", designed strictly for the class and without a real world client. We wanted to improve the course by working with a real world client and by bringing more open source process into the course. We had a client in mind, the Ronald McDonald House of Manhattan, where there were several potential projects available. We were aware of the CO-FOSS through the Humanitarian Free and Open Source project. One of the CO-FOSS projects at Bowdoin College, *Homeroom*, was very similar to a project request from RMH Manhattan. The framework seemed to be a good approach; the challenge for us was adapting it to work with our students and with our whole-class project approach.

The first phase, the pre-course activities, was heavily facilitated by working within the CO-FOSS framework and making use of the architecture and various components of *Homeroom*, a pre-existing open source project that had been developed at Bowdoin College. There were some major differences in requirements between the two projects, so our project became one of extending the *Homeroom* code rather than simply adapting it. Since we had never worked with a real client for our software engineering course before, we used the *Homeroom* project as a model in many ways. One important way in which *Homeroom* served as a model was simply as a guide for sizing our project. Choosing a project of an appropriate size and scope is critical to success with client-facing projects, so being able to compare our potential project to an already successful one was critical.

We also used the organization and resources from the Bowdoin College projects quite extensively. We used the extended project description in (Morelli, Tucker, and DeSilva, 2011), materials on the project website, and discussions with Dr. Tucker to become familiar with the structure and the decision points.

We followed the organization as much as possible since it provided a successful structure.

However, there were significant areas where adaptations had to be made. This became most apparent during the curriculum design phase. Our student body is quite different from the students at Bowdoin College. The class was larger, with 25 students. Few of them had ever written a program longer than a hundred lines, and none had ever collaborated with another student on a software project. The *Homeroom* software was written in PHP, a programming language with which most of the students had little experience. A central concern in this course was to fill in their missing skills at the same time they were developing a system design for the project. To teach students the skills they would need for the project, the first half of the semester provided in-class labs on the basics of PHP, version control, and databases. Thus, students were not able to begin project implementation until about halfway into the course.

The project involved the room scheduling process at the Ronald McDonald House of Manhattan. As mentioned before, significant differences in requirements meant that we could not simply modify *Homeroom*. In particular, RMH-Manhattan needed two interfaces, a more complex workflow for processing room reservations that involved an approval process and a lot of automated email, and an audit system that tracked all changes made to reservations. Therefore, we reused the overall architecture and the lowest level layers of *Homeroom* – the database and domain object layers – but had to do an entirely new business logic layer. The project ended up spanning two semesters. In the first iteration of the course, the students completed a skeletal prototype. During the second iteration, a new group of students worked to close out issue tickets, complete functionality, and bring the system to a point where it might be deployed.

The differing approach, both in terms of the whole-class project approach and the necessity to spend time teaching students the needed skills, meant that the syllabus structure and task assignment phases had to be modified. As mentioned earlier, students spent the first half of the semester learning the technology and tools. This was done via a series of labs, including code reading exercises working with the *Homeroom* codebase from Bowdoin. Students began development at midsemester. The project work was organized into 2 multiweek chunks, similar to the organization of the *Homebase* project. Since all students in the course were working on the same project, they were organized into teams based on functionality. Students were chosen for teams by the instructor, who used the criterion of "least risk' in making the assignments, based largely on information on resumes submitted by the students in the first week.

Since the students only had half the semester to devote to project work, the project required two iterations of the course to be finished. In the second iteration of the course, the same overall schedule was used, with students learning the tools and the codebase in the first half of the semester. In the second half, the students worked on the project in feature oriented teams again, with tickets from the issue tracker assigned to the various groups.

The Google Code open source software repository was used to host the project. This repository has version control built into it, as well as an issue tracker which we used extensively. Because the students were mainly commuter students who do not spend much time on campus, a project specific discussion board was set up, which was used extensively by all of the students. Most communication relevant to the project happened either in class or on the discussion board. All tickets from the issue tracker were automatically forwarded to the discussion board so that students would all be aware of any bugs or problems. All code commits were also forwarded to a second discussion board which all students subscribed to.

Although finishing the project required two course iterations rather than one, this was not unexpected due to the fact that the students had weaker skills and less time to spend on the project than was the case at

Bowdoin College. The open source codebase for *Homebase* and *Homeroom* served as both a model and as a set of classes and modules that could be adapted to the differing requirements of RMH-Manhattan. Even more importantly, the CO-FOSS course organization and materials were critical to our success; it is not likely we would even have attempted a project of this scale with a real client without a "recipe" which could be adapted to our needs.

Case study 4: *DONATE*

Adapting the framework for a course at an urban, commuter college with transfer students constrained financially and by work and family commitments.

Affordability has been a compelling reason for adopting FOSS in the computing curricula and for equipping the computing labs with support infrastructure in the Computing Technology program at University of New Hampshire at Manchester (UNH Manchester). Using FOSS systems and services, however, is just the first step in taking advantage of how FOSS development principles and practices can impact students learning. In this case study we describe the experience with adapting the CO-FOSS framework for an upper-level elective course in the B.S. Computer Information Systems (BS CIS) major at UNH Manchester. The major requirements are structured into a two-layer core (eight courses), integrative and professional experience (four courses, including internship and capstone project courses), a self-designed concentration in an application domain (four courses), and three computing electives.

An overarching challenge for teaching CO-FOSS development in the BS CIS program is a collection of hard constraints placed on students by their work, family obligations, and other commitments. When time on campus is reduced to class meetings only, students are forced to do project work once a week, typically the night before the class meeting, with almost no time to coordinate their work with other team members. These constraints are compounded by the students' uneven academic preparedness. A large majority of students (70%) have more than 50 credits transferred from local two-year colleges, where (1) projects were individual endeavors; 2) student exposure to FOSS principles and practices was limited or non-existent, and 3) prior programming experience did not include algorithm design and using abstraction to tackle more complex problems.

In Spring, 2012, the *Homebase* approach was used in the Web Application Development course, an upper-level elective course. Eleven students enrolled in the course: four undergraduate CIS majors, two graduate IT majors, and five continuing education students. The continuing education students were engineers from a local company in Manchester, who were interested in open source technologies and gaining experience developing web-based services for their company's in-house applications. The client was YWCA of Manchester, with whom the UNH-Manchester program had collaborated since Fall 2008 on various projects and student service learning activities. The project addressed the client's need for an information system that tracks donations from individuals and organizations. Like the other nonprofits mentioned in this paper, YWCA cannot afford to buy software and pay developers or consultants. Open source is the only feasible approach that they have for developing a donation tracking system.

At the beginning of the semester, the project's prototype had a code base with the same layered architecture as the *Homebase* software. The project was a combined result from two other courses with student projects of smaller and dedicated scope – one database course and one web authoring course, both at sophomore level. The prototype's back-end had a functional MySQL database, although incomplete, with a well-designed schema, sample data, and scripts to install, populate, and query the database. The front-end had a single use case implemented, i.e., viewing donors and searching donors by name. The prototype was staged on a virtual machine that runs on UNH-Manchester's server. The code base is hosted by Google Code.

In the first half of the semester students learned how to develop an open source client-based project through a variety of activities: 1) experimenting with the *Homebase* and *Homeroom* projects in the textbook (Tucker, Morelli, & de Silva, 2011) ; 2) meeting with the YWCA business director to get clarifications on the system requirements; 3) learning from three experienced FOSS developers who joined the class via Skype; 4) receiving feedback and having their work reviewed by the software engineers who were auditing the class; and 5) doing assignments that provided practice with: PHP and SQL, model-view-controller architectural pattern, techniques for specifying requirements and design decisions, XAMPP run-time environment configuration, and a comprehensive development toolkit (Eclipse PHP, Xdebug, Doxygen, SimpleTest, and Balsamiq). In addition, a project forum, wiki, and hosted project version control and issue tracking supported teamwork and collaboration. The code base became the "common denominator" for all design and implementation decisions. That is why the assignments were grounded in the code base and allowed students to learn first-hand about the developer roles needed for the project. By the time the six week-long project sprints occurred in the second half of the semester, students understood well the roles they would assume to maximize their contributions to the project.

At UNH-Manchester, the *Homebase* approach yielded several good outcomes. First, having continuing education engineers as observers allowed creation of the role of configuration manager, who was the most active and involved student during class meetings. These observers also participated in client meetings, reviewed and gave feedback on use cases, database and UI design, and code quality. Students also learned from one of the invited speakers about the Asana service for project management activities, which they ended up using for issue tracking.

CONCLUSIONS

Trying to incorporate real world software practices and project experience into the typical one semester undergraduate software engineering course is always challenging. The CO-FOSS approach, which occupies a middle ground between proprietary software development for individual customers and working within a large, existing FOSS project, offers a number of advantages. Projects developed in conjunction with a nonprofit client can be tailored to be student friendly and fit within the constraints of the academic semester. At the same time, following FOSS practices means that the architecture, codebase, and documentation can be reused and adapted for future projects. As shown in this chapter, a CO-FOSS project does not have to remain resident within one college but can be adapted by other schools in true open source fashion. Our future directions involve efforts to bring more schools to this approach, creating ways to support maintenance of these projects, and investigating ways to foster communication among students working on similar projects at different schools.

In conclusion, this chapter illustrates how a CO-FOSS project course can be designed and delivered. Such a course embodies a novel curriculum design that teaches communication skills, teamwork, and writing skills. It prepares students by engaging them in real-world problems, introducing them to fresh technologies (such as agile programming, layered architecture, and mobile computing), and leveraging all the advantages of open source development and tool use. We hope that our work will inspire others to rethink their courses in these ways, so that their students will become better prepared to enter the software industry with both feet on the ground.

REFERENCES

Bredeson, A. (2013). Second Helpings volunteers get much-needed technological boost from app built by students. *The Island Packet*, June 2013. Retrieved July 24, 2013, from http://www.islandpacket.com/2013/06/14/2541912/second-helpings-volunteers-get.html.

Cheng, Y., & Lin, J. (2010). A constrained and guided approach for managing software engineering course projects. *IEEE Transactions on Education*, 53(3), 430 – 436.

Coppit, D., & Haddox-Schatz, J. (2005). Large team projects in software engineering courses. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (pp. 137–141), ACM Compute Society.

Coyle, Edward J., Leah H. Jamieson, and William C. Oakes. 2005. "EPICS: Engineering Projects in Community Service." *International Journal of Engineering Education*, 21 (1).

Ellis, H. J. C., Morelli, R. a., Lanerolle, T. R. De, & Hislop, G. W. (2007). Holistic Software Engineering Education Based on a Humanitarian Open Source Project. *20th Conference on Software Engineering Education & Training (CSEET'07)* (pp. 327–335). IEEE Press.

Farrell, V., Ravalli, G., & Farrell, G. (2012). Capstone project: fair, just and accountable assessment. *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (pp. 168–173), ACM.

Fogel, K. (2005), *Producing Open Source Software*. O'Reilly, Sebastopol, CA.

Grundy, J. (1996). A comparative analysis of design principles for project-based IT courses. *Proceedings of the Second Australasian Conference on Computer Science Educatio*n. (pp. 170–177), Australian Computer Society.

Hauge, O., Ayala, C., & Conradi, R. (2010). Adoption of open source software in software-intensive organizations – A systematic literature review. *Information and Software Technology*. 52(11) 1133–1154.

Judith, W., Bair, B., & Börstler, J. (2003). Client sponsored projects in software engineering courses. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003)* (pp. 401–402). ACM.

Linos, P. K., Herman, S., & Lally, J. (2003). A service-learning program for computer science and software engineering. In *Proceedings of the 8th annual conference on Innovation and Technology in Computer Science Education* (pp. 30–34). ACM.

Mackellar, B. K. (2011). A software engineering course with a large-scale project and diverse roles for students. *Journal of Computing Sciences in Colleges*, 26(6), 93–100.

MacKellar, B. K., Sabin, M., & Tucker, A. (2013). Scaling a framework for client-driven open source software projects: a report from three schools. *Journal of Computing Sciences in Colleges*, 28(6), 140–147.

MacKellar, B. (2013). Analyzing coordination among students in a software engineering project course. *26th IEEE Conference on Software Engineering Education and Training, San Francisco, CA; 05/2013* (pp. 279–283), IEEE Press.

Marmorstein, R. (2011). Open source contribution as an effective software engineering class project. *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education - ITiCSE '11*(pp. 268-272). ACM.

Madsen, F., & Nürnberg, P. (2005). Calliope: supporting high-level documentation of open-source projects. *Proceedings of the 2005 Symposia on Metainformatics* (p. 10). ACM.

McConnell, J. J. (2006) Active and cooperative learning. *ACM SIGCSE Bulletin*, 38(2), 24.

Meneely, A., Williams, L., & Gehringer, E. (2008). ROSE: a repository of education-friendly open-source projects. *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, (pp. 7–11), ACM.

Morelli, R., de Lanerolle, T., & Tucker, A. (2012). The HFOSS Project: Engaging Students in Service Learning through Building Software. In Nejmeh, B. (Ed.), *Service Learning in the Computer and Information Sciences*, (Chapter 5), Wiley and IEEE Press, New York, NY.

Olsen, A. L. (2008). A service learning project for a software engineering course. *Journal of Computing Sciences in Colleges*, *24*(2), 130–136.

Poger, S., & Bailie, F. (2006). Student perspectives on a real world project. *Journal of Computing Sciences in Colleges*, *21*(6), 69–75.

Richards, D. (2009). Designing Project-Based Courses with a Focus on Group Formation and Assessment. *Transactions on Computer Education*, 9(1), 1–40.

Tadayon, N. (2004). Software engineering based on the team software process with a real world project. *Journal of Computing Sciences in Colleges*, *19*(4) (pp. 133–142).

Tan, J., & Jones, M. (2008). A case study of classroom experience with client-based team projects. *Journal of Computing Sciences in Colleges*, *23*(5), 150–159.

Tucker, A., Morelli, R., & de Silva, C. (2011). *Software Development: An Open Source Approach*. Boca Raton, FL, CRC Press.

Venkatagiri, S. (2006). Engineering the software requirements of nonprofits: a service-learning approach. *28th IEEE International Conference on Software Engineering*, (pp. 643–648), IEEE Press.

CO-FOSS RESOURCES

Examples of the *Homebase* and *Homeplate* requirements documents:
http://code.google.com/p/rmh-homebase
http://code.google.com/p/sh-homeplate

The current version of *Homebase* and its requirements document
http://code.google.com/p/rmh-homebase

The *Homeplate* project
http://code.google.com/p/sh-homeplate.
Student oriented Mercurial and Eclipse tutorials
http://myopensoftware.org/content/supporting-materials

Example syllabi for CO-FOSS project courses
http://myopensoftware.org/content/extended-course-syllabus.

The *RMH-RoomReservationMaker* project
https://code.google.com/p/rmh-roomreservation-maker/

Examples of currently popular project repositories:
SourceForge http://sourceforge.net/
Google Code https://code.google.com/
GitHub https://github.com/.

Examples of currently popular version control systems
CVS http://savannah.nongnu.org/projects/cvs
Subversion http://subversion.apache.org/
Mercurial http://mercurial.selenic.com/
Git http://git-scm.com/

ADDITIONAL READING SECTION

Course Organization and Project Planning

Bernhart, M., Grechenig, T., Hetzl, J., & Zuser, W. (2006). Dimensions of software engineering course design. *Proceedings of the 28th International Conference on Software Engineering* (pp. 667–672), ACM.

Filho, W. P. (2006). A software process for time-constrained course projects. *Proceedings of the 28th International Conference on Software Engineering* (pp. 707–710). Shanghai, China: ACM.

Gehrke, M., Giese, H., Nickel, U. A., Niere, J., Tichy, M., Wadsack, J. P., & Zündorf, A. (2002). Reporting about industrial strength software engineering courses for undergraduates. *Proceedings of the 24th International Conference on Software Engineering* (pp. 395–405). Orlando, Florida: ACM.

Agile Software Process

Larman, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. (2004). Prentice Hall PTR. NJ: Upper Saddle Rivier.

Reichlmayr, T. (2003). The agile approach in an undergraduate software engineering course project. *IEEE **Fro**ntiers in Education Conference* (pp. 13–18), IEEE Press.

Sanders, D. (2007). Using Scrum to manage student projects. *Journal of Computing Sciences in Colleges*, *23*(1), 79–79.

Sherrell, L. B., & Robertson, J. J. (2006). Pair programming and agile software development: experiences in a college setting. *Journal of Computing Sciences in Colleges*, *22*(2), 145–153.

Team Formation and Task Assignment

Anewalt, K. (2009). Dynamic group management in a software projects course. *Journal of Computing Sciences in Colleges*, *25*(2), 146–151.

Beck, J., Almstrum, V. L., Ellis, H. J. C., & Towhidnejad, M. (2009). Best practices in software engineering project class management, *ACM SIGCSE Bulletin*, (pp. 201–202), ACM.

Ikonen, M., & Kurhila, J. (2009). Discovering high-impact success factors in capstone software projects. *Proceedings of the 10th ACM Conference on Information Technology Education* (pp. 235–244). Fairfax, Virginia, USA: ACM.

Nita-Rotaru, C., Dark, M., & Popescu, V. (2007). A Multi-Expertise Application-Driven Class. *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (pp. 119–123). ACM.

Stein, M. V. (2002). Using large vs. small group projects in capstone and software engineering courses. *Journal of Computing Sciences in Colleges*, *17*(4), 1–6.

Way, T. (2005). A company-based framework for a software engineering course. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (pp. 132–136). ACM.

Developer Communication and Code Sharing

Beck, J. (2005). Using the CVS version management system in a software engineering course. *Journal of Computing Sciences in Colleges*, *20*(6), 57–65.

Hart, D. (2009). A survey of source code management tools for programming courses. *Journal of Computing Sciences in Colleges*, *24*(6), 113–114.

Lancor, L. (2008). Collaboration tools in a one-semester software engineering course: what worked? what didn't? *Journal of Computing Sciences in Colleges*, *23*(5), 160–168.

Liu, C. (2005). Using Issue Tracking Tools to Facilitate Student Learning of Communication Skills in Software Engineering Courses. *18th IEEE Conference on Software Engineering Education & Training (CSEET'05)* (pp. 61–68). IEEE Press.

Tan, J., & Jones, M. (2008). An evaluation of tools supporting enhanced student collaboration. *2008 38th Annual Frontiers in Education Conference* (pp. 7–12). IEEE Press.

Evaluation of Team Members' Contributions

Ellis, H. J. C., & Mitchell, R. (2004). Self-grading in a project-based software engineering course. *17th Conference on Software Engineering Education and Training, 2004. Proceedings.* (pp. 138–143). IEEE Press.

Hayes, J. H., Lethbridge, T. C., & Port, D. (2003). Evaluating individual contribution toward group software engineering projects*, Proceedings of the 25th International Conference on Software Engineering* (pp. 622–627). Portland, Oregon: IEEE Press.

Wilkins, D. E., & Lawhead, P. B. (2000). Evaluating individuals in team projects. *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education* (Vol. 32, pp. 172–175). ACM.

Open Source Software Projects and Tools

Ellis, H., Morelli, R., & Hislop, G. (2008). Work in progress-challenges to educating students within the Community of Open Source Software for Humanity. *IEEE Frontiers in Education Conference* (pp. 7–8), IEEE Press.

Ellis, H. J. C., Purcell, M., & G. W. Hislop, G. W. (2012). An approach for evaluating FOSS projects for student participation. *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. (p. 415). ACM.

Gehringer, E. F. (2011). From the manager's perspective: Classroom contributions to open-source projects. *2011 IEEE Frontiers in Education Conference* (pp. 1–5). IEEE Press.

Lancor, L., & Katha, S. (2013). Analyzing PHP frameworks for use in a project-based software engineering course. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (pp. 519–524), ACM.

Liu, C. (2005). Enriching software engineering courses with service-learning projects and the open-source approach. *Proceedings of the 27th International Conference on Software Engineering* (pp. 613–614). St. Louis, MO, USA: ACM.

Morelli, R., Tucker, A., Danner, N., De Lanerolle, T., Ellis, H., Izmirli, O., Krizanc, D., and Parker, G. (2009), Revitalizing computing education through free and open source software for humanity, *Communications of the ACM*, 52 (8) 67-75.

Pedroni, M., Bay, T., Oriol, M., & Pedroni, A. (2007). Open source projects in programming courses. *ACM SIGCSE Bulletin*, *39*(1), 454.

Tucker, A. (2009). Teaching client-driven software development. *Journal of Computing Sciences in Colleges*, 24(4), 29–39.

KEY TERMS AND DEFINITIONS

Agile development: Methodology to develop software in short iterations, called sprints, in which existing code is first refactored, tests for a new requirement are written, and test-driven code is developed.

Free and open source software: Software licensed and freely distributed along with its underlying code.

Nonprofit organization: Usually a charity or service organization that uses any surplus revenues to support its mission and achieve its goals.

Requirements: Functional capabilities (what the system will do) to which the software system must confirm.

Domain model: Representation of conceptual classes or real-situation objects in the domain of interest.

Layered architecture: Organization of the software classes into layers such that higher layers call upon services of lower layers.

Model-View-Controller architecture: A three-layer architecture in which domain knowledge is maintained by the domain model objects, displayed by the view (user interface) objects, and manipulated by control (application logic) objects.

Code base: All the source code files stored in a source control repository that handles various versions and tracks implementation issues.

Real-world projects: Software engineering projects that are sponsored by real clients and result in production-grade software systems.