Selecting Open Source Software Projects to Teach Software Engineering

Thérèse Smith, Robert McCartney, and Swapna S. Gokhale
Department of Computer Science and Engineering
University of Connecticut, Storrs, CT 06269
[tms08012, robert, ssg]@engr.uconn.edu

Lisa C. Kaczmarczyk
Consultant
San Diego, CA 92130
lisak@acm.org

ABSTRACT

Aspiring software engineers must be able to comprehend and evolve legacy code, which is challenging because the code may be poorly documented, ill structured, and lacking in human support. These challenges of understanding and evolving existing code can be illustrated in academic settings by leveraging the rich and varied volume of Open Source Software (OSS) code. To teach SE with OSS, however, it is necessary to select uniform projects of appropriate size and complexity. This paper reports on our search for suitable OSS projects to teach an introductory SE course with a focus on maintenance and evolution. The search turned out to be quite labor intensive and cumbersome, contrary to our expectations that it would be quick and simple. The chosen projects successfully demonstrated the maintenance challenges, highlighting the promise of using OSS. The burden of selecting projects, however, may impede widespread integration of OSS into SE and other computing courses.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; K.3.2 [**Computers and Education**]: Computers and Information Science Education

General Terms

Documentation, Design, Experimentation

Keywords

Software Engineering, Maintenance, Program Comprehension, Open Source

1. INTRODUCTION AND MOTIVATION

The need for skilled software engineers has compelled nearly every computing curriculum to include Software Engineering (SE) as a mandatory course. The objective of this SE course is to train students in principles and practices of modern software engineering and to prepare them for careers in the software industry. An overwhelming number of software engineering activities involve understanding and evolving existing, legacy code. Comprehension of such legacy code may be difficult because it could: (i) have been subject to many fixes and enhancements, (ii) be poorly written, (iii) have little to no documentation, and/or (iv) lack support. Computing students must therefore be trained to "reverse engineer" design decisions and their rationale from such code, often under serious time and resource constraints.

SIGCSE'14, March 3–8, 2014, Atlanta, GA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2605-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2538862.2538932.

A major difficulty in highlighting comprehension and evolution challenges to SE students is to find pre-existing code that has characteristics similar to that of industrial code but is commensurate with students' preparation and background. The rich and diverse Open Source Software (OSS) projects can readily supply such pre-existing code [6]. It can also provide a context to practice design, development, and testing skills. OSS-based SE projects will force students to comprehend sparsely documented and/or poorly written code. OSS can thus be leveraged to emulate industrial challenges in academic environments.

Our objective was to integrate OSS projects into our sophomore-level SE course. Because OSS projects exhibited large variability in terms of size, complexity, and quality, our greatest obstacle was to select suitable projects. The projects could not be too large or complex because students would not be able to understand them well enough to extend them. These projects also could not be too small or simple because the students would not find it necessary to use SE principles, rather they could fall back to the disorganized generate-and-test approaches that were sufficient in the introductory courses. Given these constraints, we sought to manually identify, evaluate, and prepare projects for integration. This paper describes how our selection effort was fraught with unexpected challenges and was labor intensive. The selected projects, however, provided opportunities for deep involvement with the code, meaningfully highlighting the maintenance difficulties. Our experience thus suggests that in order to use OSS for SE instruction, the burden of selecting and preparing projects must be alleviated.

This paper is organized as follows: Section 2 discusses the promise of OSS. Section 3 compares OSS repositories . Section 4 presents project selection process. Section 5 describes how the chosen projects met course objectives. Section 6 surveys related work. Section 7 concludes the paper.

2. OSS: PROMISE AND PERILS

The OSS revolution is having a lasting impact on the way software is developed, disseminated, and adapted. This is evident from the myriad of OSS projects already available, and the pace at which software engineers contribute to the development of these projects. This revolution has thus created easily accessible, exponentially growing [9] volume of code, some of which can be used as example code in SE education. Popular OSS repositories such as Sourceforge [29], Freecode [12], CodePlex [7], and W3C [33] host more than 200,000 projects. We use these repositories simply as code sources; students do not contribute back to them.

OSS projects span a variety of domains such as tools for software development, financial analysis, security and networking, data manipulation and visualization, audio and video engineering and text editing, operating and database systems, and games and entertainment [29]. These projects target common platforms including Windows and Linux and are written in languages such as C, C++, Perl, Fortran, Python, Java, Tcl, Objective-C, Ada, and Php. Although the OSS development process appears *ad hoc*, many projects are completed successfully with rich functionality and high reliability because their latent social structure allows the projects to grow in an organized manner [2, 31]. Moreover, because most engineers participate in OSS to build reputation, or for self-development [23] and altruistic reasons, they may be inherently committed to SE principles, which may lead to projects that are carefully designed, engineered, and maintained. OSS projects may thus exhibit varied characteristics and quality similar to industrial systems, even though their requirements processes may differ from that of traditional software systems [26]. These projects can thus provide a valuable resource for teaching SE.

The rich diversity of OSS systems, however, presents one of the greatest hurdles in teaching with these projects. All chosen projects must be of comparable difficulty and complexity. Independent selection by students from OSS repositories is not only unlikely to produce that result but it may also be inordinately time consuming. Therefore, teaching SE with OSS will require identification and evaluation of a collection of suitable projects. We seek a collection because it improves the chance that students will find an interesting project and also minimizes cross-group collaboration. Such collaboration is undesirable because it diminishes each student's contribution to critical analysis of the documentation and reflection on the architecture and data representation that is necessary for comprehensive understanding [27].

3. ANALYSIS OF OSS REPOSITORIES

Each OSS repository hosted a large number of projects and showed varying search and selection capabilities. Therefore, we explored many repositories, subject to a preliminary criteria, to identify those with potential for efficiently providing appropriate projects.

3.1 Preliminary Criteria

Our objective was to integrate OSS into an early, introductory SE course for sophomore and juniors. Our students had limited experience in Java programming and could write, test, and debug only small volumes of code after having taken a data structures course. They could also use Eclipse, create UML class diagrams, and produce documentation. Our SE course was "maintenance-centric" and the laboratory exercises performed on OSS projects challenged students to practice code comprehension, use appropriate tools (source code control systems, reverse engineering), and implement an enhancement of their choice. The students were expected to work methodically and document their activities systematically, especially, those undertaken to complete their enhancement. Given the background and skills of our students and the objectives of the course, we devised the following preliminary criteria:

- *Programming Language (PL):* The projects must be programmed completely in Java because it was the only language known by all of our students.
- *Code Size (CS):* Project size should be approximately 10,000 lines; smaller projects may not adequately emulate maintenance difficulties and larger ones may be beyond our students' capabilities [20].
- *Team Development (TD):* Projects with a team were desirable, for possibly quick resolution of issues.
- *Buildability (BD):* Projects should build within one to two days to preserve focus on understanding and enhancing the code.

3.2 Exploration of Repositories

We searched several OSS repositories subject to PL, CS, TD, and BD criteria to narrow a large pool of candidate projects. We believed that these preliminary criteria would be easy to assess; obviously it should be possible to determine code size, programming language and whether a team was involved with a project with minimal effort. Also, we expected most OSS projects to be close to building. Contrary to these expectations, however, we found that mostly large projects, which were beyond the code size criterion built easily and many smaller projects with acceptable code sizes failed to build. Therefore, we deferred examining for buildability and describe our findings in applying only the CS, PL and TD criteria to the OSS repositories.

- *Apache:* Apache hosted many projects with excellent quality, sizeable communities, noteworthy documentation, and passing the BD test. However, the smallest projects wildly exceeded the size.
- *CodePlex:* Projects at CodePlex could be rapidly filtered using "alpha or better," "beta or better," and "stable" only. There were over 9000 projects classified as "beta or better". Though Javascript is a tag, Java is not, and entering Java as a search term returned 389 projects, and restoring "beta or better" dropped the number to 195. Many of these 195 projects were Javascript. Even some in C# were returned, because these included Java in the tags or descriptive text. Thus, our initial impression was that the site was helpful and quick, but not many projects were suitable.
- *Code.google:* A relatively large percentage of projects at this repository were executables rather than source.
- *Freecode:* Freecode offered a very nice search experience, with a number of keywords. It, however, referred to many projects at various other repositories. The first dozen or so projects examined appeared to be executables unaccompanied by code. We inferred that Freecode was an improved search front end but not a repository and had many executable-only hits.

- *GNU:* GNU had many projects, including a subset for Windows, but those checked were not in Java.

- *Sourceforge:* Sourceforge had many projects, and useful search keywords beyond the general search area. Searching for Java projects, however, still yielded a small number of projects in other languages. It also permitted searches over several categories of activity and achievement such as "alpha", "beta", "stable", "mature", and "inactive". This was used to eliminate projects whose stages of development were too early or too late. Beyond these terms, users could create additional search qualifiers, which appeared to be matched against project text descriptions. The searches were not as selective and the results had to be pared manually. However, the popularity of the site combined with our experience indicated usable material.

- *Tigris:* This repository hosted several projects, but these exceeded the code size.

- *GitHub:* This repository offered search terms including language and most recent push date, which provided approximately 18,000 Java projects.

- *W3C:* W3C featured excellent quality software, some in Java, but the projects appeared too big.

Sourceforge appeared to be the most productive based on CS and PL, and hence, we explored it further against the TD criterion. However, limiting the code size to 10,000 lines while simultaneously requiring a team of developers virtually resulted in no projects. Many appealing projects with several developers but over 17,000 lines of code could be found easily. Several projects with fewer than 5,000 lines of code were by single individuals. It seemed that there were not many projects in the mature, stable or alpha, or even beta state with more than one developer that also had fewer than 12,000 lines of code. Thus, we revised the TD criterion because it conflicted with CS, and elected to accept single developer projects with approximately 10,000 lines, resulting in about 1000 projects for detailed evaluation.

3.3 Comparison of Capabilities

During our exploration, we found that the search and download capabilities of OSS repositories differed widely. All repositories could narrow projects with pre-defined key words and allowed users to create additional search terms. Matches to pre-defined key words, however, differed from matches to user-created search terms because projects may be manually classified according to pre-defined key words, while user-created key words may be matched against the text associated with the project. These conjectures arose because the search results were different with pre-defined key words *vs.* using them as user-defined search terms.

The repositories also differed in their specific pre-defined key words. The classification scheme "beta or better" seen at CodePlex was more closely aligned with users' thinking than the scheme at Sourceforge which has "beta" and "mature". Code.google and Freecode also supported pre-defined search criteria, but the labels at former were dramatically smaller than the number of search terms at the latter. GitHub provided a desirable ability to use recency as a search term, compared to obtaining that information after a search. The naming scheme may suggest content, source or "only" executables: src could be seen in the name.

The careful cataloging at Freecode greatly aided the search process, however, some projects found through Freecode were hosted at Sourceforge. The relative frequency of finding source code, as opposed to binaries, was higher via Sourceforge *vs.* Freecode. Thus, it appeared that Freecode placed relatively more emphasis upon searching and less emphasis on hosting than Sourceforge.

While each site appeared to be addressing a specific audience of software users and developers, the degree of coordination among the community of contributors could be different. Domains at Sourceforge could be imagined to have grown according to the contributions. Apache's emphasis on community process, and a seeming interrelatedness of the categories (e.g., Web server, mail, XSLT formatted output, XML parsing, etc.), suggested more coordination.

The repositories also exhibited varying support for searching based on project size. GitHub allowed searching based on repository size, which was only somewhat related to code size. At Code.google, the number of lines of code was reported only for some projects. Other repositories did not allow a search based on project size. Most of the sites offered reasonable support for downloading project artifacts. At Code.google, we found many projects made available as multiple individual files, unsuitable for download. Projects, if they were packaged to be downloaded as archives, could be imported directly into a prepared Java project in Eclipse as archive files. GitHub downloaded with a Git clone. Sourceforge downloaded with the originator's choice of CVS, SVN, Git and Mercurial.

Based on these lessons, we contemplated the characteristics of an ideal repository for project selection. Such a repository would host a large collection of projects, provide apt and usable search terms, and distinguish from unsuitable projects. For example, being able to separate "executable only" projects from those with source code could eliminate a large subset of the projects in one search query. The repository would also allow projects to be scoped and searched based on their size. Finally, it would facilitate downloading project artifacts. Being able to exercise one action, such as a repository clone, or a single download, would be preferred over manually downloading individual project files.

4. PROJECT SELECTION

A whole summer of 12 hour days was spent examining candidate software projects. Three computers were kept running, continuously downloading Java software that looked interesting based upon manual inspection of subject matter and age. Of the 1000 projects downloaded and (manually) examined, 200 passed the initial criteria. These 200 were further investigated. Several additional criteria were applied to ensure that the chosen projects were appropriate to meet the objectives of our maintenance-centric course.

- *Code Size (CS):* Initially, approximate sizes were estimated to lie between 5,000 and 10,000 lines. Accurate sizes were counted after eliminating duplicate files.

- *Programming Language (PL):* To be completely certain, PL was applied as a search qualifier rather than typing it in the search entry field because this could return projects with Java in their description, but were not necessarily programmed in Java. However, when Java was specified as a qualifier, the search produced projects written exclusively in Java.

- *Application Domain (AD):* We sought projects that were "cool" and appealing to the students. Our prior interactions with students suggested that projects with graphics such as 3D visualization, audio/video tools, and those with GUIs would satisfy this criteria. While the projects at Sourceforge were searchable according to domains, establishing correlation between domain and size was cumbersome. Although projects in most domains were too big, **some** in Home and Education, Audio and Video, and Games were of suitable size.

- *Modular Design (MD):* Projects should be modular and loosely coupled to effectively demonstrate best practices that aid maintenance. Modularity was assessed using metrics that characterized code organization including the number of files, number of directories, and the depth of the directory structure. Directory naming and structure such as the Model, View, Controller [13] was also used to provide clues about a project's modularity and organization.

- *Recent Activity (RA):* Projects with recent updates and commits were sought for ease of seeking help. We evaluated activity based on the date of the most recent commit, and the number of committers, if available.

- *Documentation Quality (DQ):* Projects should *not* be exceptionally well documented, because this would only trivialize the comprehension and evolution challenges. Moreover, all the projects should be documented more or less uniformly, so that the students would be challenged similarly. To assess quality, we considered multiple aspects such as readability, accuracy, degree of detail, availability, helpfulness, recency and correctness. If the documentation was not in English, we considered that in assessing its helpfulness, although strictly this aspect is different from quality. Despite our quest for uniformity, documentation differed widely and came in many forms including web sites, user guides, reuse from other sources, pictures and diagrams, and comments in the code.

- *Buildability (BD):* Initially, we immediately rejected projects that did not build. Upon realizing that only big projects built easily, we became lenient and invested reasonable time to build a project. Thus, we discarded a candidate project with ordinary appeal if it could not be compiled with an hour-long effort. A project, with highly desirable appeal was eliminated only if several hours of debugging failed to build it. Many projects did not build because of missing jar files, which could be obtained using jarfinder. Only a handful required elaborate and *ad hoc* search to obtain or substitute the missing files. We maintained a collection of the missing files that we found for future use. Thus, effort incurred in building projects could be categorized as "minimal", "moderate" or "extreme".

In summary, our process of evaluating Sourceforge projects revealed that: (i) It is difficult to find team projects with approximately 10,000 lines of code; (ii) Small, single developer projects do not build easily; (iii) Quality of documentation can vary widely; and (iv) Code organization may offer misleading clues into its modularity.

5. PROJECT SUITABILITY

Table 1 summarizes the characteristics of the 16 selected projects with respect to each criteria. Java projects spanned the domains of Games, Art, Skills, Indexing, Searching and Client/Server applications [25]. The projects were moderately modular and recent; being no more than two years old. Only a few projects had two developers. Sizes ranged from 5,500 to 10,500 lines, and most were lightly documented. After we spent hours, students could compile the projects quickly based on our guidance and readily available jar files.

Next, we discuss how these projects fared in meeting our maintenance-centric objectives. Table 2, which summarizes the range of enhancements, shows that most students engaged with both the architecture (A) and the data representation (DR) of their projects. Students were initially frustrated with the difficulties in understanding code that was not written by them, without much support. However, this initial frustration was later replaced with a healthy sense of pride, accomplishment, and self-achievement, as they navigated and completed their chosen extensions. End-of-the semester surveys [19, 14] indicated that the students appreciated that software maintenance and evolution was: (i) challenging, resource intensive, and time consuming; (ii) could be significantly aided by documentation and comments; and (iii) assisted by a good architecture but hindered by a brittle one. The chosen OSS projects thus successfully achieved our objectives of illustrating comprehension and evolution challenges and also provided a context to learn and practice the skills to meet these challenges.

Table 2: Project Enhancements and Engagement

Project	Enhancement	Engmnt
1. carDriving	GUI, Tutorial	A, DR
2. Coppit	GUI, Rules, Bug Fixes	A, DR
3. Domination	GUI	A, DR
4. Solitaire	GUI, Rules	A, DR
5. JigsawPuzzle	GUI	A, DR
6. JugglePat	GUI, Coordinate music	A, DR
7. Melosion	Entry of a new song	DR
8. Monopoli	GUI	A
9. Simulum	GUI, Coordinate music	A, DR
10. MusicCoach	Play, pause, rewind, GUI	A, DR
11. NocNorade	Rules	A, DR
12. Picofarm	GUI	A, DR
13. SlimeWarrior	GUI	A, DR
14. vgt-battleships	GUI	A, DR
15. Sudoku	Rules	DR
16. Puggle	Add doc types	A, DR

A = Architecture, DR = Data Representation

6. RELATED WORK

OSS projects have been used to teach SE activities such as design, testing, quality assurance, maintenance, and usability. They have also been used to teach basic and advanced programming and software development concepts. These works have adopted varied approaches for project selection; allowing students to choose with or without guidance [30,

Table 1: Characteristics of Selected OSS Projects

Project Name	Project characteristics	Project Name	Project Characteristics
1. carDriving	**AD:** Game, graphics, sound, physics **MD:** 25, 1, 1 **AL:** 2011/06/16, 2 committers **DQ:** Web site, some documentation **CS:** 8931 **BD:** Minimal	9. Simulum	**AD:** lovely, 3D effect **MD:** 42, 17, 2 **AL:** 2010/01/25 **DQ:** Web site **CS:** 7586 **BD:** Minimal
2. Coppit	**AD:** Board game with computer player **MD:** 17 files, 3 directories **AL:** 2010/10/05 **DQ:** Wikipedia page **CS:** 6461 **BD:** Minimal	10. MusicSkillsCoach	**AD:** Sound, useful for musicians **MD:** 33, 9, 4 **AL:** 2011/01/14 **DQ:** good naming **CS:** 5259 **BD:** Moderate
3. Domination	**AD:** Board game **MD:** 49, 6, 1 **AL:** 2010/10/08, 6 committers **DQ:** Web site, class explanations **CS:** 8006 **BD:** Moderate	11. NocNorade	**AD:** Partial game **MD:** 26, 8, 3 **AL:** 2007/01/08 **DQ:** Web site **CS:** 5586 **BD:** Moderate
4. FourRowSolitaire	**AD:** Card game **MD:** 16, 1, 1 **AL:** 2011/06/28 **DQ:** Comments at top and within files **CS:** 5633 **BD:** Moderate	12. Picofarm	**AD:** Client/Server, Soap, RMI **MD:** 33, 4, 1 **AL:** 2007/03/28 **DQ:** Website, extended comments **CS:** 7049 **BD:** Moderate
5. JigsawPuzzle	**AD:** Entertainment **MD:** 49, 11, 1 **AL:** 2010/07/10 **DQ:** comments every few lines of code **CS:** 7774 **BD:** Extreme	13. SlimeWarrior	**AD:** Animation **MD:** 20, 6, 3 **AL:** 2011/02/13 **DQ:** Included html pages, YouTube **CS:** 10401 **BD:** Moderate
6. JugglePat	**AD:** Patterns, animation **MD:** 20, 3, 3 **AL:** 2009/07/17 **DQ:** Comments, usage guide **CS:** 8411 **BD:** Moderate	14. vgt-battleships	GUI, computer opponent **MD:** 33, 6, 3 **AL:** 2010/01/25 **DQ:** Commands, variables in Polish **CS:** 5951 **BD:** Moderate
7. Melosion	**AD:** Music game, animation **MD:** 64, 13, 3 **AL:** 2011/08/02, at least 4 committers **DQ:** Dutch Web site, class diagram **CS:** 9229 **BD:** Extreme	15. Sudoku	**AD:** Puzzle generation **MD:** 102, 16, 5 **AL:** 2010/06/30 **DQ:** Readme, good naming **CS:** 10636 **BD:** Minimal
8. Monopoli	**AD:** Board game, nice animation **MD:** 33, 5, 2 **AL:** 2009/07/17, 2 committers **DQ:** Italian comments, every few lines **CS:** 5353 **BD:** Minimal	16. Puggle	**AD:** Search Indexing **MD:** 49, 8, 2 **AL:** 2011/04/07 **DQ:** HTML at Sourceforge **CS:** 10848 **BD:** Moderate

AD = Application Domain, MD = Modularity, AL = Activity Level, DQ = Documentation Quality, CS = Code Size, BD = Buildability

24, 18]; selection by faculty from familiar projects [16, 8, 21, 22], or popular ones such as Apache and Mozilla [4], JUnit [3], software studio [22], or pertaining to specific domains such as Web and Web 2.0 [34] or tools [5]; software written in graduate and upper division courses [17, 11, 10, 20], and soliciting projects from academia and industry [32]. Meneeley *et al.* [20] also faced disappointing results at Sourceforge when searching subject to similar criteria. In response, they decided to use a single project developed by their graduate students. We continuted our search to find comparable projects, and hope that sharing our lessions would offer systematic guidance and tools for efficient project selection.

7. CONCLUSION AND FUTURE WORK

Open Source Software (OSS) can be used as example code in SE education to emulate the challenges faced by software engineers in comprehending and working with existing, legacy code. This paper reported on our experiences and lessons in selecting OSS projects for teaching SE. To enable project selection, despite the peculiarities of OSS repositories, we developed criteria that would result in suitable projects for a sophomore/junior-level SE course. Our search turned out to be far more time consuming and burdensome than anticipated, but it also led to interesting insights into the capabilities of OSS repositories. The selected projects were successful in demonstrating the comprehension and evolution challenges. Thus, the burden of selecting and preparing projects must be mitigated for OSS to be widely adopted in SE and computing education.

We propose to explore several avenues to reduce this selection burden. These include investigating repositories such as GitHub and social coding for good [28], slightly larger projects and those beneficial to users [15], and examining whether different code and community metrics can assess initial suitability before investing in preparing projects. Contributing the selected projects for use by other instructors [20, 1] is also a concern of the future.

8. ACKNOWLEDGEMENTS

This material is based on work supported by the National Science Foundation under grant DUE-1044061. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

9. REFERENCES

[1] Computing Portal: Connecting Computing Educators. `http://www.computingportal.org/`.

[2] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *Proc. of SIGSOFT Intl. Symp. on Foundations of Software Engineering*, pages 24–35, 2008.

[3] E. Brannock and N. Napier. "Real-World testing: Using FOSS for software development courses". In *Proc. of the Annual Conference on Information Technology Education*, pages 87–88, 2012.

[4] Y. Cai, J. Popyack, S. Mancoridis, and J. Salvage. Contemporary canonical software courses. NSF Proposal, 2009.

[5] D. Carrington and S. K. Kim. Teaching software design with open source software. In *Proc. of Annual Frontiers in Education Conference*, volume 3, pages S1C–9–14 vol.3, November 2003.

[6] T. Clear. Comprehending large code bases-the skills required for working in a brown fields environment. *ACM SIGCSE Bulletin*, 37(2):12–14, 2005.

[7] Codeplex. Codeplex - open source project hosting. `http://www.codeplex.com/`.

[8] D. Damian, C. Lassenius, M. Paasivaara, A. Borici, and A. Schroter. "Teaching a globally distributed project course using Scrum practices". In *Proc. of Collaborative Teaching of Globally Distributed Software Development Workshop*, pages 30–34, 2012.

[9] A. Deshpande and D. Riehle. The total growth of open source. *IFIP International Federation for Information Processing*, pages 197–209, July 2008.

[10] J. D. N. Dionisio and K. D. Dahlquist. Improving the computer science in bioinformatics through open source pedagogy. *SIGCSE Bull.*, 40:115–119, June 2008.

[11] J. D. N. Dionisio, C. L. Dickson, S. E. August, P. M. Dorin, and R. Toal. An open source software culture in the undergraduate computer science curriculum. *SIGCSE Bull.*, 39:70–74, June 2007.

[12] Freshmeat. Welcome to freshmeat.net. `http://freshmeat.net/`.

[13] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, March 1995.

[14] S. Gokhale, R. McCartney, and T. Smith. "Teaching Software Engineering from a Maintenance-Centric View". *The Journal of Computing Sciences in Colleges*, page 42, 2013.

[15] M. Goldweber, J. Barr, T. Clear, R. Davoli, S. Mann, E. Patitsas, and S. Portnoff. A framework for enhancing the social good in computing education: A values approach. *ACM Inroads*, 4(1):58–79, 2013.

[16] D. Hepting, L. Peng, T. Maciag, D. Gerhard, and B. Maguire. Creating synergy between usability courses and open source software projects. *ACM SIGCSE Bulletin*, 40(2):120–123, 2008.

[17] C. Liu. Adopting open-source software engineering in computer science education. In *Proc. of ICSE Workshop on Open Source Software Engineering,*, pages 85–89, 2003.

[18] R. Marmorstein. "Open source contribution as an effective software engineering class project". In *Proc. of the Annual Joint Conference on Innovation and Technology in Computer Science Education*, pages 268–272, 2011.

[19] R. McCartney, S. Gokhale, and T. Smith. "Evaluating an early software engineering course with projects and tools from open source software". In *Proc. of the 9th Intl. Conf. on Computing Education Research*, pages 5–10. ACM, 2012.

[20] A. Meneely, L. Williams, and E. F. Gehringer. Rose: a repository of education-friendly open-source projects. *SIGCSE Bull.*, 40(3):7–11, June 2008.

[21] J. Nandigam and V. Gudivada. "Source Code Exploration as a Case Study Towards Application Comprehension". In *Proc. of the 9th Intl. Conference on Education and Information Systems, Technologies and Applications*, 2011.

[22] T. Nurkkala and S. Brandle. "Software Studio: Teaching Professional Software Engineering". In *Proc. of the Technical Symp. on Computer Science Education*, pages 153–158, 2011.

[23] S. Oreg and O. Nov. Exploring motivations for contributing to open source initiatives: The roles of contribution context and personal values. *Computers in Human Behavior*, 24(5):2055–2073, September 2008.

[24] M. Pedroni, T. Bay, M. Oriol, and A. Pedroni. Open source projects in programming courses. *SIGCSE Bull.*, 39(1):454–458, March 2007.

[25] E. Reynolds. "UConn Github". `https://github.com/`, 2011.

[26] W. Scacchi. Understanding the requirements for developing open source software systems. *IEE Proceedings Software*, 149(1):24–39, 2002.

[27] S. E. Sim and R. C. Holt. The ramp-up problem in software projects: A case study of how software immigrants naturalize. In *Proc. of the Intl. Conference on Software Engineering*, pages 361–370, 1998.

[28] `socialcoding4good.org` viewed 6/5/2013.

[29] Sourceforge. Sourceforge.net: Find and develop open source software. `http://sourceforge.net/`.

[30] S. Sowe and I. Stamelos. Involving software engineering students in open source projects: Experiences from a pilot study. *Journal of Information Systems Education*, 18(4), 2008.

[31] S. Sowe, I. Stamelos, and L. Angelis. Understanding knowledge sharing activities in free/open source projects: An empirical study. *Journal of Systems and Software*, 81(3):431–446, 2007.

[32] E. Stroulia, K. Bauer, M. Craig, K. Reid, and G. Wilson. "Teaching distributed software engineering with UCOSP: The undergraduate capstone open-source project". In *Proc. of the Community Building Workshop on Collaborative Teaching of Globally Distributed Software Development*, pages 20–25, 2011.

[33] w3c. World Wide Web Consortium. `http://www.w3.org/`.

[34] G. Xing. "Teaching software engineering using open source software". In *Proc. of the 48th Annual Southeast Regional Conference*, page 57, 2010.