6-1-2007

An Open Source Software Culture in the Undergraduate Computer Science Curriculum

John David N. Dionisio
Loyola Marymount University, dondi@lmu.edu

Caskey L. Dickson

Stephanie E. August
Loyola Marymount University

Philip M. Dorin
Loyola Marymount University

Ray Toal
Loyola Marymount University

An Open Source Software Culture in the Undergraduate Computer Science Curriculum

John David N. Dionisio, Caskey L. Dickson,
Stephanie E. August, Philip M. Dorin, and Ray Toal
Department of Electrical Engineering & Computer Science
Loyola Marymount University
1 LMU Drive, MS 8145
Los Angeles, California 90045-2659 USA
dondi@lmu.edu caskey@technocage.com saugust@lmu.edu, pdorin@lmu.edurtoal@lmu.edu

Abstract

Open source software has made inroads into mainstream computing where it was once the territory of software altruists, and the open source culture of technological collegiality and accountability may benefit education as well as industry. This paper describes the *Recourse* project, which seeks to transform the computer science undergraduate curriculum through teaching methods based on open source principles, values, ethics, and tools. *Recourse* differs from similar projects by bringing the open source culture into the curriculum comprehensively, systematically, and institutionally. The current state of the project is described, and initial results from a pilot exercise are presented.(1)

Keywords: open source software development, curriculum reform, teaching framework

1. Introduction

Open source software engineering revisits and innovates the core principles of computer science. In the discipline's nascent years, software emerged as a new type of scholarly work, akin to mathematicians' proofs and biologists' field data. Computer scientists freely shared and published source code for a growing library of programs, advancing both knowledge and understanding. Industry took notice, and software transitioned from a product of study to a product of commerce. The practice of open source retreated to the academe and the altruistic.

The paradigm of software-as-trade-secret continues today, but is faltering. Increasing software complexity and security risks have created a backlash against closed source [18]. In the last decade, software engineering began to re-embrace the open source model, which has since proven itself in industry and research for designing, creating, and maintaining quality software [4].

Undergraduate computer science education has taken notice of open-source trends as well. Traditionally, most undergraduate students produce toy programs and algorithms from scratch, work alone, and are never placed in a position where they have to work with someone else's code. Thus, they find themselves paradoxically out of place when they enter the job market, where team development is the norm. To resolve this mismatch, Loyola Marymount University (LMU) has initiated a project that seeks to make an open source culture an integral part of computer science undergraduate education.

The project, called *Recourse*, adapts the open source development culture for undergraduate computer science education, employing a novel curriculum progression and associated teaching methods. Ultimately, the *Recourse* project's goal is to establish the open source culture as computer science's educational norm, making collaborative software development intrinsic to the curriculum. The foundational premise is that open source concepts and practices can be translated into improved undergraduate computer science learning and, ultimately, a better match between undergraduates and their subsequent endeavors.

2. Previous and Related Work

Open source development practices have been promoted as a potential approach toward effective software quality control, which has been observed to be in critically short supply [18]. While these practices are gaining momentum in industry, they have not yet permeated the undergraduate curriculum [4].

A number of computer science programs have explored new methods for teaching undergraduate computer science, with similar goals of bridging the gap between student preparation and industry expectation [11, 13, 14, 15, 17, 19, 20, 21]. These methods are deployed, however, on an ad-hoc, course-by-course basis, with little institutional foundation for their use and management. Other endeavors have adapted aspects of the open source culture, either to an individual class within a specific subject area [8] or as a source of tools for computer science

program assessment [1]. Some work explores collaboration in isolation, without expressly adopting open source elements [3].

The *Recourse* project differs from these approaches by reforming the curriculum through open source principles systematically, comprehensively, and institutionally. Teaching methods are explicitly adapted from current best practices in open source software engineering [4]. With cutting-edge development techniques and tools permeating the program, the project hopes that these elements become second nature to students, allowing them to focus on the *science* behind the *computer*. Thus, while the project may certainly produce better software developers, its ultimate goal is to create better *computer scientists* who happen to have industry-ready development skills.

3. Characteristics of an "Open Source Culture"

At this writing, the official *open source definition* stands at version 1.9 [16]. While the definition ostensibly concerns what open source software *is*, it ultimately reflects a set of *values* — values that collectively characterize the open source *culture*:

- *Source code is available, modifiable, and long-lived.* Many of the open source definition's criteria are concerned with obtaining access to and subsequently evolving source code. Source code exists in multiple copies due to its free distribution. As a result, any piece of open source code potentially has a very long life, independent of its originator.
- *Accountability implies community.* The open source definition protects both authors and users by requiring an explicit "paper trail" for how a particular program changes over time. *Communication* and *answerability* are crucial to this paper trail. The net result is that a *community* of developers and users of different levels and proficiencies naturally forms around an open source project, and this community is another key component of the culture.
- *Responsibilities accompany rights.* The open source definition attaches categorical, irrefutable importance to the *license* that accompanies all open source software. Adherence to the license relies on the honor system, and this honor system pervades not only the specifics of open source licenses, but also covers other junctions of rights and responsibilities, such as: giving credit where credit is due; appropriate access to software, documentation, and communities; and acknowledging any security, privacy, confidentiality, or legal issues surrounding the development and use of open source software.

The open source definition and its underlying values form the basis of the *Recourse* project.

4. Open Source Teaching Framework

The *Recourse* project attempts to ad*a*pt — not ad*o*pt — open source culture and values to an undergraduate computer science curriculum. With adaptation to a curriculum, the final objective shifts from software development to learning computer science. While the end is not identical, we observe that the means may be analogous.

4.1 Curriculum Progression

The first element of the teaching framework is an "arc" of progression through the curriculum, from freshman to graduating years. In terms of activities and artifacts produced, this arc parallels the growth curve of a maturing software developer. In terms of intellectual and academic themes, the progression provides a computer science equivalent to the Dewar-Bennett Mathematical Knowledge-Expertise Grid [7, 9, 10], which was, in turn, adapted from Shavelson's science knowledge typology and Alexander's *model of domain learning* (MDL) [2, 12].

In a *Recourse* curriculum, freshman courses focus on the study, testing, and fixing of existing code — corresponding to the initial phases of participation by a software developer in an open source community (or any other team development effort, for that matter). *Pre-existing* is a key component of this stage: conventional teaching at this level usually involves writing "toy" programs from scratch. By teaching basic programming techniques in the context of pre-existing code, a student may be exposed to big-picture ideas sooner and as a result may realize the value of cleanly written, well-documented code from the outset — a concept which is traditionally very difficult to teach.

Sophomore courses, which typically introduce data structures, algorithms, and computer system organization, provide an appropriate foundation for coding specific functions and modules from scratch, accompanied by unit tests. Again, this activity corresponds to the software development stage where a programmer begins to implement new but well-bounded functionality. Unit tests serve as an unambiguous (and easily automated) mechanism for validating the correctness of submitted work.

Junior-level courses (and beyond) introduce subfields within computer science, expecting sufficient programming proficiency as the general concepts of sophomore courses find specific applications in operating systems, programming language design and implementation (i.e., compiler construction), networking, interaction design, databases, computer graphics, artificial intelligence, and others. Activities for these courses take the form of term-length projects: students now design and implement software from scratch. An open source software developer finds an analogous milestone when starting his or her own open source projects. At this point, students come full circle, as the long life of code becomes apparent: software written at the junior level finds its way back to the freshmen, as pre-existing code that must be examined, fixed, and completed.

Finally, the senior year expects students to demonstrate proficiency in computer science, through capstone projects that synthesize prior material based on individual interests. LMU has implemented such projects since the 1980s, and in the context of an open source culture, they gain renewed meaning as they correspond to an open source developer's growth into a mature, confident, and self-motivated creator with the skills and experience to take on and even initiate formidable software projects.

At the end of the four phases and themes of this curriculum "arc," a student will have acquired the knowledge and skills of a computer science bachelor's degree not only through conventional texts, lectures, and exercises, but also through collaborative programming, shared code, and accountability for one's work. Graduates of such a program would be *both* better computer scientists *and* competent software developers who can hit the ground running immediately, whether in industry or research.

4.2 Instructional Techniques

Alongside the curriculum arc, individual teaching techniques have been derived from the open source culture. As they are based mainly on coding practices, note that they do not comprise an *entire* educational approach, and are instead meant to be applied in the context of overall exercises for teaching computer science, system design and documentation, and programming.

4.2.1 "Sample Code Bazaar"

Sample Code Bazaar refers to the creation and maintenance of live, organized, searchable, student-accessible sample code libraries. Students access the sample code using the same tools and processes that they would encounter elsewhere — they must learn to interact with a source code repository or they are unable to look at examples nor do their work.

The primary student activity surrounding the *Sample Code Bazaar*, in addition to viewing, downloading, and trying examples presented in class, is the creation of "derived works" from the sample code. Assignments consist of downloading a sample program, making a specified change to it, then submitting the new version.

4.2.2 "Test Infection"

The *Test Infection* technique adapts the workflow of test-driven development approaches to classroom assignments [5]. The technique's name derives from Erich Gamma's observation that, once a programmer adopts test-driven development, he or she "never goes back" and subsequently advocates this change to fellow software developers [6]. It may be combined with the *Sample Code Bazaar*, or applied to code written from scratch. With *Test Infection*, both the instructor and the students validate submitted code by preparing test fixtures for detecting as wide a variety of errors as possible.

A particularly engaging approach to *Test Infection* is to build the test suite vs. implementation matrix in class. As tests fail or incorrectly pass wrong implementations, we can display and discuss the code that is in question. The code can then be modified on the fly, until the correct behavior is achieved ("going green," to borrow a phrase from the test-driven development arena). This way, students are exposed to yet another recommended development practice in the context of learning the material at hand.

4.2.3 "The Cyclic Life of Code"

The *Cyclic Life of Code* technique reflects the longevity of code in the open source world. Any student-submitted code, from isolated exercises to term-length projects, becomes a permanent part of the computer science program's repository. This code feeds back into the *Sample Code Bazaar* or *Test Infection* techniques as sample programs, implementations against which unit tests can be run, or ongoing projects for incremental improvement or refactoring.

Since all student submissions are stored permanently, code that a student wrote in freshman year may be revisited in later years. Revisiting their own old code may give students a greater appreciation for proper documentation and structure — after all, if they have issues with re-assimilating their own code a year later, what more if the code is inherited by someone else?

4.2.4 "Release Early, Often, & Open"

The benefits of iterative and incremental development over "waterfall"-based development have been known for decades. Undergraduate student projects under *Recourse* follow this development model, facilitating early, frequent, and unrestricted dialog among students, instructors, and fellow students on current progress.

The *Release Early, Often, and Open* approach for managing software projects applies to both student coursework *and* faculty research. Open source in research is already common today; the *Recourse* project's teaching techniques serve primarily to make this model an explicit norm that students can use as examples for their own work.

5. Initial Results

We are currently in the process of gathering results for several case studies. One such case study, involving the *Test Infection* technique, is reported here.

5.1 Procedure and Findings

The instructor provided six students with a plain English description for a simple programming language; the description was written to be as precise as possible without actually expressing a formal grammar. The students were to specify a grammar for the language and implement a parser for it, with an accompanying test suite to determine whether the parser recognizes or rejects strings correctly.

Table 1 lists the results of the initial phase, in which the student test suites (Suites 1 to 6) were run against the instructor's implementation. Because only a single correct implementation was prepared beforehand, test suites with errors of omission could not be detected; this situation must be accounted for in the final version of *Test Infection*. Overall, four errors in the test suites were detected, labeled with the glyphs *a* to *d*. Suite 1 had two errors (*a* and *b*), Suite 2 had three errors (*a*, *c*, and *d*), and Suite 3 had one error (*b*).

Table 1: First-phase results for *Test Infection*

Suite 1	Suite 2	Suite 3	Suite 4	Suite 5	Suite 6
ab	*acd*	*b*			

From Table 1 each student's test suite was run against the instructor's single correct implementation. Test cases that wrongly failed something that the implementation actually handled correctly were detected and fixed. Four distinct errors were found, indicated by the glyphs *a*, *b*, *c*, and *d*.

Once the test suite errors were corrected, each test suite (S1 to S6) was then run against each student's implementation (Parser 1 to Parser 6). The instructor's test suite (S0) and parser (Parser 0) were included in this matrix as well; the results are shown in Table 2. As in Table 1, each glyph represents an error detected by a test. Since the test suites' cases are all correct at this point, the errors now represent errors in the *implementation* and not the test. Thus, errors of omission can be detected — note how Parsers 1, 5, and 6 had errors that were not caught by all test suites (including the instructor's!).

In the final version of the *Test Infection* teaching technique, many of these errors of omission would be caught in the initial testing-the-testers run, as long as the instructor had an incorrect implementation that is known to have this problem. Errors of omission caught in the second phase would represent errors for which the instructor did not have a prepared incorrect implementation. In a fully-realized *Recourse* curriculum, such errors can then be folded into the library, and over time, more and more errors of omission can be caught right away.

Table 2: Second-phase results for *Test Infection*

Suites	S0	S1	S2	S3	S4	S5	S6
Parser 0							
Parser 1				*u*			
Parser 2	*vvw*	*vx*	*v*	*vvvw*	*vv*	*vvy*	*v*
Parser 3							
Parser 4							
Parser 5	*w*			*uw*	*w*		
Parser 6		*z*			*z*		

From Table 2, each test suite (including the instructor's) was run against every implementation. Errors in the parsers are indicated by the glyphs *u* through *z*.

5.2 Student Reaction

This trial run was conducted openly — that is, the test suites were run in class, for both phases. Thus, the students participated in interpreting and fixing the erroneous test cases in the first phase, and again participated in fixing the parser errors detected in the second phase.

Subjectively, the excitement and attention level in the classroom was genuinely positive. To the students' credit, identification and repair of errors (in both phases) were performed in a constructive manner, resulting in a strong sense of teamwork and community involvement.

Students rated the exercise as more enjoyable than a conventional programming assignment, where they would have turned in a program to be corrected by the instructor one-on-one. The openness of the exercise provided increased motivation to turn in a correct implementation — the desire not to be "embarrassed" was an incentive, although the "embarrassment" itself was ultimately good-natured, resulting in students helping each other. It must be noted that while this mirrors the "blame log's" and the pressure not to submit bad code in collaborative software development, great care must be taken during in-class reviews, as not all students may handle this type of critique constructively. It may have helped that the instructor's own test suite missed some cases — after all, if the instructor made some oversights, acknowledged them, and fixed them openly, then there should be no serious stigma when students' work encounters similar issues.

6. Next Steps

We are working on specifying, designing, and implementing a support platform for automating appropriate elements of a *Recourse*-based curriculum, such as the management of student work and the instructor's preparation and assessment of this work. Much of the required functionality (revision control, test frameworks, communication, issue tracking) already exists in real-world open source software development, so some adaptation and wiring together of existing tools and services is expected.

Some questions regarding the adaptation of the open source culture to an undergraduate computer science curriculum remain:

- How should student work under the *Recourse* teaching techniques be graded?
- What intellectual property rules govern student-authored software projects under this paradigm?
- The accompanying hardware/software support platform may require significantly more resources than are currently available for a computer science undergraduate program. Can a shared infrastructure be established, so that multiple institutions may utilize a single system?

These issues will be tackled as the project progresses.

7. Conclusion

This paper has introduced the *Recourse* project, describing its goals, objectives, curriculum arc, and teaching strategies. An initial case study of one of the project's teaching techniques was also presented and discussed.

The ultimate, albeit ambitious, goal of *Recourse* would be for its methodology to become the norm for undergraduate computer science education. The project's desired outcomes are not only worth pursuing but *must* be pursued, if universities are to produce better bachelors-level computer scientists.

References

[1] A. Abunawass, W. Lloyd, and E. Rudolph. COMPASS: A CS program assessment project. In *Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education*, pages 127–131, Leeds, United Kingdom, June 2004.

[2] P. A. Alexander. The development of expertise: the journey from acclimation to proficiency. *Educational Researcher*, **32**(8):10–14, 2003.

[3] S. Azadegan and C. Lu. An international common project: implementation phase. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, pages 125–128, Canterbury, United Kingdom, June 2001.

[4] L. Barnett and C. E. Schwaber. Applying open source processes in corporate development organizations. Technical report, Forrester Research, Inc., Cambridge, MA, May 2004. http://vasoftware.com/sourceforge/request_info-dl.php?paper=9.

[5] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[6] K. Beck, E. Gamma, and D. Saff. JUnit test infected: Programmers love writing tests. http://junit.sourceforge.net/doc/testinfected/testing.htm, 2006.

[7] C. Bennett and J. Dewar. Taxonomy of mathematical knowledge expertise. In Mary Huber and Pat Hutchings, editors, *The advancement of learning: Building the teaching commons*, pages 40–41. Jossey Bass, San Francisco, CA, 2005.

[8] M. Claypool, D. Finkel, and C. Willis. An open source laboratory for operating systems projects. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, pages 145–148, Canterbury, United Kingdom, June 2001.

[9] J. Dewar and C. Bennett. 8-dimensional mathematical knowledge-expertise grid. http://myweb.lmu.edu/carnegie/webport/Knowgrid.htm, 2004. Loyola Marymount University.

[10] J. Dewar and C. Bennett. Teaching students to talk and think about mathematics. http://kml2.carnegiefoundation.org/html/poster.php?id=504, June 2004. Loyola Marymount University.

[11] A. Feldman. Homework 1 for computer science 117: Submitting programs. http://math.boisestate.edu/<alex/courses/cs117/f04/hw2.html, 2004. Boise State University.

[12] E. Fox and P. A. Alexander. Reading, interest, and the model of domain learning: A developmental model of interest, knowledge, and strategy in text comprehension. In *American Educational Research Association*, San Diego, California, April 2004.

[13] P. Hanrahan. Computer graphics homework 2. http://graphics.stanford.edu/courses/cs348b-02/homework/hw2.html, 2002. Stanford University.

[14] J. Houlahan. How to submit homework electronically. http://www.cs.jhu.edu/<houlahan/cs107/esub.html, 2004. Johns Hopkins University.

[15] Laboratory in Software Engineering. Homework validation and turnin. http://6170.lcs.mit.edu/www-archive/Old-2000-Fall/handouts/turnin.html, 2000. Massachusetts Institute of Technology.

[16] Open Source Initiative. The open source definition. http://opensource.org/docs/definition.php, 2005.

[17] J. Katz. Programming resources: Homework FAQ. http://www.wam.umd.edu/<taowei/414web/pages/FAQ00.htm, 2003. University of Maryland.

[18] C. C. Mann. Why software is so bad. *Technology Review*, June 2002.

[19] Mathematical Sciences. Electronically submitting coursework. http://www.divms.uiowa.edu/help/msstart/submit.html, 2004. University of Iowa.

[20] Z. Shao and Y. R. Yang. How to submit assignments. http://flint.cs.yale.edu/cs112/help/submit.html, 2004. Yale University.

[21] C. Wyman. Submitting homework for computer graphics. http://www.cs.uiowa.edu/<cwyman/classes/fall04-22C151/howto/hw-submit.html, 2004. University of Iowa.

Endnote

(1) Partial support for this work was provided by the National Science Foundation's Course, Curriculum, and Laboratory Improvement Program, Award No. 0511732.