EXPERT CODE REVIEW AND MASTERY LEARNING IN A SOFTWARE DEVELOPMENT COURSE

Sophie Engle, Sami Rollins
Department of Computer Science
University of San Francisco
San Francisco, CA 94117
sjengle@cs.usfca.edu, srollins@cs.usfca.edu

ABSTRACT

For students to be successful in upper-division courses and as junior developers, they must master concepts such as code design and concurrency. However, traditional grading and partial credit often allows students to pass courses without demonstrating appropriate mastery. This paper reports on our experience applying mastery learning and expert code review to our software development course. We compare two consecutive semesters of this course—one using a traditional approach and the other using mastery learning and expert code review. We discuss our experience setting student expectations, the differences in grades and code quality between the semesters, and provide recommendations on how to improve and adapt this approach for other courses.

1. INTRODUCTION

The ACM Computer Science Curriculum 2008 highlights "significant project experience" as an expected characteristic of computer science graduates [2]. At our university, students are required to take a second-year software development course that provides this project experience and prepares them for programming-heavy upper division courses. This course teaches students fundamental software development principles including modular code design and refactoring, as well as advanced implementation topics such as concurrency and efficiency. Students incrementally implement a simplified search engine through a series of large programming projects that build upon each other.

After several iterations of the course, we discovered that some student were passing the course without having mastered the most critical concepts the class is intended to teach, and would move on to upper division courses with broad but weak programming skills. As a result, these students would often fail and need to repeat a course in the junior or senior year, in many cases delaying graduation.

This paper reports on our experience applying a mastery learning approach combined with expert code review to ensure that students passing the course have demonstrated mastery of software development principles and advanced programming topics. We compare two consecutive semesters of this course. The first semester uses a traditional grading approach, whereas the following semester uses mastery learning and code review. Our results demonstrate that this new approach forced students to refactor their code and master complex topics such as code design and concurrency. We were also better able to capture issues with the code that may not have been caught through automated testing alone. We discuss these results further after providing a background for the course and outlining the critical elements of our approach.

2. BACKGROUND

Our software development course is designed to be taken by undergraduate sophomores in preparation for upper division courses, and has been offered seven times to date. Undergraduates entering the course have, minimally, taken one semester of Python programming and one semester of Java programming. The course acts as a bridge between the lower division classes and upper division courses. The main goal of the course is to give students the foundation and experience necessary to implement a well designed and large piece of software with roughly 2,000 lines of code. To accomplish this, students must learn to develop iteratively, test, debug, and refactor code. The emphasis of the course is on code design instead of correctness—students often already have the ability to produce mostly correct results,

but often choose the quick and dirty path.

The class has four main components: lectures, labs, exams, and projects. We employ a fairly traditional lecture-based model with hour-long lectures twice per week. These lectures usually consist of a broad overview of the concepts being covered followed by interactive code demonstrations. Each week there is also an hour-long lab during which students with limited programming experience benefit from hands-on help with weekly homework assignments. One midterm exam and a final exam also comprise part of the final grade.

The main focus of the class is a series of programming projects that build upon each other to result in a simplified search engine at the end of the semester. Though our mastery learning approach is not specific to this particular set of projects, we have found that the search engine is a good fit for the course. Students are very motivated to learn how to build a piece of technology they use everyday. Moreover, the search engine provides a good opportunity to teach, in the context of a real application, several critical computer science concepts such as concurrency and efficiency.

The initial iterations of this course assigned seven projects. The **Word Count** project requires students to open a text file and count the number of times a word appears, giving them a gentle re-introduction to text and string parsing in Java. For the **Inverted Index** project, students must create a generalized and encapsulated inverted index data structure, and populate that data structure by recursively traversing a local directory. The **Partial Search** project requires students to extend the inverted index to support efficient partial search of multiple word queries, and rank and sort the search results. For the **Multithreading** project, students must make the inverted index data structure thread-safe using a custom multiple readers/single writer lock, and implement multithreaded construction and searching of the index. The **Web Crawler** project requires students to download web pages using raw sockets, parse HTML links, strip HTML tags, and populate the inverted index with the resulting plain text. Finally, the **Search Engine** project requires students to implement a web server that allows search, user registration and login/logout, search history viewing, account maintenance, and additional features of the students' choice. This final project requires students to demonstrate thorough understanding of web technologies and modular, robust code design.

The course has been very popular and many students, anecdotally, report that the experience gained in the course is extremely helpful in the future courses as well as in searching for software development jobs. However, we noticed that some students were passing the course with a C-level understanding of most concepts without having mastered key principles. These students would move on to the programming-heavy upper division courses with broad but weak programming skills. As a result, these students would often fail and need to repeat a course in their junior or senior year, in many cases delaying graduation.

Several factors were contributing to this problem. First, it is challenging to grade subjective concepts such as code design. It is tempting to award partial credit to poorly designed solutions, which allows some students to narrowly pass the class by accumulating points from exams and other assignments. Second, mastery of a concept such as refactoring is most clearly demonstrated by forcing the student to reimplement a submitted solution. We originally believed that the iterative nature of the projects would encourage students to refactor their code based on feedback on earlier assignments. However, we discovered that students are loath to modify code they believe works unless their grade depends upon it. Finally, it can be difficult to assess whether students have appropriately applied an advanced concept such as multithreaded programming without careful review of their code; a solution may give correct results most of the time, but not consistently.

3. APPROACH

To address these challenges, we revised the course to apply two primary principles: **mastery learning** [1] and **expert code review**. We offered this course for the first time in Spring 2012. Our approach includes three main features. First, students are required to revise and resubmit projects until they demonstrate mastery by earning an A-level grade. Final course grades are assigned based on the number of assignments completed with mastery. Second, assignments are designed such that top students

are expected to resubmit at least once, forcing students to practice an iterative and agile approach to development. Finally, extensive code review by the instructor is used to identify design problems or issues with correctness that are difficult to test automatically. Implementing this approach requires careful planning for the assignment structure and grading. Consider the following table:

Project	Suggested	Cutoff	Letter	Percent
Project 1 Inverted Index	Week 04 2/17	Week 09 3/23	F	100%
Project 2 Partial Search	Week 10 3/30	Week 12 4/13	D	90%
Project 3 Multithreading	Week 12 4/13	Week 15 5/04	C	80%
Project 4 Web Crawler	Week 14 4/27	Week 16 5/11	B	70%
Project 5 Search Engine	Week 16 5/11	Week 16 5/11	A	60%

The table above illustrates the suggested submission schedule given to students in Spring semester. We elected to reduce the original set of seven projects by two, assigning the word count and HTML parsing projects as homework. The schedule was slightly accelerated when compared to previous semesters. Following the suggested deadlines intentionally gave students less time to complete assignments, encouraging an agile approach. The large break between the suggested deadlines for project 1 and 2 accommodated for the midterm exam and spring break.

The cutoff in the above table indicates the date a final solution, which demonstrates mastery of the project, must be submitted in order for a student to receive a grade high enough to pass the course. Students are allowed no more than one submission per week, and a new project may not be submitted until the previous has been graded. The cutoff serves a practical purpose by ensuring there is sufficient time for grading each submission, and it prevents students from procrastinating under the assumption that all work can be completed and submitted at the end of the semester.

The grading scheme is another key element of our approach. Students do not receive partial project grades. Instead, they are graded on the number of projects completed with mastery. Completing the first three projects, for example, results in a C-level overall project grade. The overall project score is worth 60% of the final grade, and the project grading scheme is designed to make it difficult for students to pass the course without mastering at least projects 1 through 3.

To enforce mastery learning, students must both get the correct output and pass an extensive code review evaluating the design of the program. Our goal is to focus on code quality in the grading process, therefore students are given automated tests that they are expected to execute prior to submission to ensure their solutions generate correct output. Even so, some students fail to properly execute the tests they are given or fail to correctly interpret the output. As a result, the first step of the grading process is to verify that the program output is correct and, if not, the submission is returned.

Expert code review of each project is a critical element of our approach. Code review, however, is well known to be extremely time consuming. Our approach identifies several specific categories for each project. Once a student demonstrates mastery of the metric in an earlier project, we assume the student will continue to demonstrate mastery in later projects since each project builds on the previous. For the first project, for example, we focus on the following criteria: encapsulation (e.g. no pass-by-reference of private data members), generalization (e.g. separating parsing logic from data structure), proper use of keywords (e.g. private, static, etc.), and code style (e.g. braces, spacing, naming of methods, etc.). Later projects focus on more advanced topics such as runtime efficiency and thread safety.

4. RESULTS

To evaluate how well this approach met our goals, we compared two consecutive semesters of the course that were taught by the same professor. The Fall 2011 semester uses a traditional grading approach, whereas the Spring 2012 semester uses our mastery learning and expert code review approach. We focused only on undergraduate computer science majors that attempted at least one project in our analysis. This resulted in 9 students for Fall and 12 students for Spring. For both semesters, we analyzed project averages, source lines of code, and how many students submitted each project. We also tracked

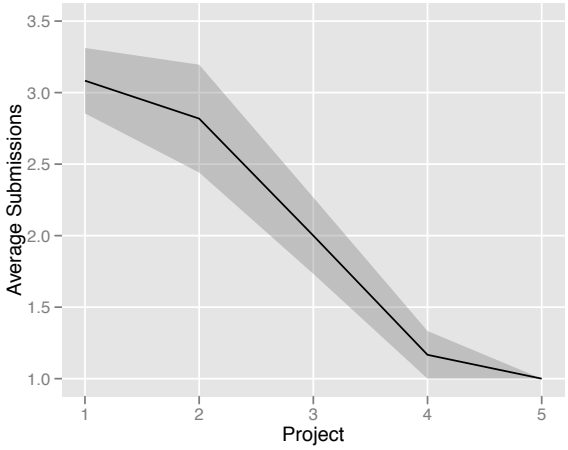
the number of submissions per project per student in the Spring semester, as well as the type of errors found during the code review. Our main findings are as follows:

- Students' mastery of code design and refactoring improved.
- Students' mastery of complex concepts such as concurrency improved.
- Students did not progress nearly as far with the projects.
- The time commitment required for the code review process was reasonable.

The following table provides an overview of our results, which we discuss next.

	Avg Submissions		Average SLOC		Average Grade	
Project	**Fall**	**Spring**	**Fall**	**Spring**	**Fall**	**Spring**
Project 1 Inverted Index	NA	3.1	186	218	58%	100%
Project 2 Partial Search	NA	2.8	341	418	76%	90%
Project 3 Multithreading	NA	2.0	494	706	93%	76%
Project 4 Web Crawler	NA	1.2	686	914	84%	38%
Project 5 Search Engine	NA	1.0	2360	1781	96%	16%
Overall Project Average	**NA**	**2.5**	**569**	**590**	**81%**	**82%**

The above table demonstrates that this approach was successful at forcing students to practice iterative development. Every student had to refactor and resubmit projects at least once, and students had on average 2.5 submissions per project. Students who received an A- or above in the class averaged a near ideal 2.1 submissions per project, whereas students who did not receive a passing grade averaged nearly 3.3 submissions per project. The following figure also illustrates that as the projects progressed, students required fewer submissions. For example, students who attempted project 1 averaged 3 submissions, but this average dropped to 2 submissions for students who attempted project 3.



We also used the SLOCCount tool to compare the physical source lines of code (SLOC) between the two semesters [10]. Students in the Spring semester produced more code, on average, than students in the Fall semester. This can be attributed to the better enforcement of code design requirements such as generalization and encapsulation, which often lead to larger code bases. Anecdotally, students also reported increased confidence in their code designs near the end of the semester.

The project grades demonstrate that Spring students performed better, on average, for the first two projects, but fewer students reached the later assignments. For example, only 33% of Spring 2012 students attempted project 5, resulting in more scores of 0%. In contrast, 100% of students submitted the latter projects in Fall 2011.

Setting student expectations was the major factor contributing to this slowed progression. The switch to mastery learning between consecutive fall and spring semesters took students by surprise. It was difficult to reset student expectations, especially regarding the lack of partial credit and strictness with

which projects were now graded. Some students were also careless with their submissions, forgetting required files, using incorrect directory names, or failing to verify their submission passed all of the supplied tests. Those submissions would not be eligible for code review, and the students would have to resubmit the same project following week. We are improving this response time by performing a pre-pass on each project earlier in the week. Any problems with the submissions or output results found are immediately reported to the students, giving them a small window to fix those problems before the code reviews are performed.

The extensive code reviews also captured issues with the code that otherwise may have been overlooked, better enforcing the mastery of these topics for the Spring semester but slowing the overall progression. One of the most common issues found during the code review process for project 1 was breaking encapsulation by returning of a private variable reference in a public method. In most cases, this did not affect the runtime or output of the program, but could lead to serious issues reusing the code elsewhere. Similarly, a requirement of the multithreading project was to create a thread-safe inverted index. Though many submissions properly locked the methods in use by the particular project, other methods were left without any synchronization. As a result, the programs would run correctly but reusing the inverted index elsewhere would result in issues with data consistency.

4. ADAPTATION

We have identified several key features of our software development class that make it a particularly good fit for the mastery learning and expert code review, and feel that this approach can be adapted for any course that shares these features. These features include the submission schedule, incremental projects, topic mastery, curriculum placement, and low faculty to student ratio.

For example, having space in the class schedule to allow time for resubmissions is critical for this approach. We achieved this by reducing the total number of projects and accelerating our original schedule to allow time for feedback and resubmissions. For this to be successful, however, it is important that students have a clear understanding that they are *expected* to resubmit and iterate on their solutions.

The incremental nature of our projects is a good match for mastery learning. Since these projects build on each other, it is easy for students to understand why they need to revisit one project before moving on to the next. Moreover, it places students in a better position to perform well on later projects. However, the course material needs to be structured such that it is acceptable for students to master a subset of topics. In our case, we determined that having students master only the first three projects was preferable over having some experience with all five projects. Though helpful, it is not necessary in future classes for students to have mastered the implementation of a web crawler or server.

The sophomore year is an appropriate place in the curriculum to apply this approach. Students have enough programming foundation such that they may focus on quality rather than correctness. In addition, students are prepared for a problem that can be generally specified, giving them ample freedom to design their own approaches, and complex enough that they are unlikely to produce a well-designed solution on the first try.

Finally, code review can be a time consuming process. Part of what makes this a feasible approach is the low 15:1 faculty to student ratio in our courses [6]. The grading process took anywhere between 5 to 12 hours a week depending on the number of submissions that week. This time includes downloading code from the students' repositories, reviewing the code, and returning detailed feedback via email. Using software specifically designed for code review may also help decrease the amount of time spent on the submission process.

6. RELATED WORK

Mastery learning has been used in a variety of fields and has been shown to have a dramatic impact on student learning [1,4]. Urban-Lurain and Weinshank explored the application of the model in a CS 0 class and observe that the grades students received more accurately reflected what they had mastered [9]. More recently, LeJeune reported on using mastery learning along with contract grading in a CS 1 class finding that students felt the model improved their learning and final grades [5]. As LeJeune

points out, a large part of the motivation for applying the mastery learning technique at the introductory level is to ensure that students achieve a correct solution; awarding partial credit for a non-working program can be difficult and not accurately reflect what a student has learned. We applied the mastery learning model at a more advanced level than these other studies, such that it will help students to bridge the gap between lower and upper division courses.

Code review has been used in the past to help students produce better quality code [3,7,8]. In most cases reported in the literature, however, the focus is on peer review of code in lower-level classes whereas our focus is on expert code review. This focus on peer code review is understandable as code review is extremely time consuming for the instructor. Moreover, in introductory classes it is more feasible to give students directed criteria to consider, such as whether variables have been named appropriately. As students advance, however, having an expert available to review program is necessary to identify problems such as data structure inefficiencies or code that is not sufficiently modular.

7. CONCLUSION

Our software development course bridges lower and upper division courses, and provides students with essential software development skills. We applied mastery learning and expert code review to this course to improve the quality of code produced by the students and better prepare them for upper division coursework. Using this approach, students had to produce well designed code before moving on to the next project, and were graded on the number of projects completed. We found that using this approach, mastery of complex concepts improved. We can improve this approach further by streamlining the submission process and better setting student expectations.

We feel that several key features of this course made it a good fit for mastery learning, and that this approach may be adapted to other courses that share these features. This includes having space in the schedule for refactoring and resubmission. The projects should also be iterative, and it must be acceptable that only a subset of topics covered by those projects are mastered. Finally, this approach makes sense for a course offered in the sophomore year, where students have enough programming foundation such that they may focus on quality rather than correctness.

REFERENCES

[1] B. S. Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, 13(6):4–16, June 1984.

[2] L. Cassel, A. Clements, G. Davies, M. Guzdial, R. McCauley, A. McGettrick, B. Sloan, L. Snyder, P. Tymann, and B. W. Weide. Computer science curriculum 2008: An interim revision of CS 2001. *Report from the Interim Review Task Force*, ACM and IEEE Computer Society, December 2008.

[3] C. Hundhausen, A. Agrawal, D. Fairbrother, and M. Trevisan. Integrating pedagogical code reviews into a CS 1 course: An empirical study. *In Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, pages 291–295, New York, NY, USA, 2009. ACM.

[4] C.-L. C. Kulik, J. A. Kulik, and R. L. Bangert-Drowns. Effectiveness of mastery learning programs: A meta-analysis. *Review of Educational Research*, 60(2):pp. 265–299, 1990.

[5] N. LeJeune. Contract grading with mastery learning in CS 1. *Journal of Computing Sciences in Colleges*, 26(2):149–156, Dec. 2010.

[6] Office of Institutional Research. The University of San Francisco Fact Book and Almanac 2010. The University of San Francisco, January 2011.

[7] D. A. Trytten. A design for team peer code review. *In Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 455–459, New York, NY, USA, 2005. ACM.

[8] S. Turner, M. A. Pérez-Quiñones, S. Edwards, and J. Chase. Student attitudes and motivation for peer review in CS2. *In Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, pages 347–352, New York, NY, USA, 2011. ACM.

[9] M. Urban-Lurain and D. J. Weinshank. "I do and I understand": Mastery model learning for a large non-major course. *In The Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, pages 150–154, New York, NY, USA, 1999. ACM.

[10] D. A. Wheeler. Sloccount version 2.26. http://www.dwheeler.com/sloccount/, August 2004.