

Large Team Projects in Software Engineering Courses

David Coppit
The College of William and Mary
Department of Computer Science
Williamsburg, VA
david@coppit.org

Jennifer M. Haddox-Schatz
Daniel H. Wagner Associates, Inc.
Hampton, VA
jennifer@va.wagner.com

ABSTRACT

A key goal of educators teaching software engineering is to provide students with useful experience that will benefit them after graduation. A key component of this experience is usually a class project that is meant to expose students to the issues associated with real software development efforts. Unfortunately, educators rarely have the time required to manage software projects in addition to their normal pedagogical duties. As a result, many software engineering courses compromise the project experience by reducing the team sizes, project scope, and risk. In this paper, we present an approach to teaching a one-semester software engineering course in which approximately 30 students work together to construct a moderately sized (22 KLOC) software system. This approach provides a more realistic project experience for the students, without incurring significant managerial overhead for the instructor. We present our experiences using the approach for the spring 2004 software engineering course at The College of William and Mary.

Categories and Subject Descriptors: K.3.1 [Computers and Education]: Computer and Information Science Education—Computer science education

General Terms: Design

Keywords: software engineering education, large team projects

1. INTRODUCTION

Software continues to play an increasingly vital role in the functioning of society, and because of this, the demand for skilled software engineers persists. Unfortunately, those who do enter industry as software developers are often ill-prepared for the work that they will be expected to perform [3, 4]. A primary cause of this problem is that the institutions responsible for producing tomorrow's software professionals are still in the process of determining the proper way to teach the concepts and skills students need when they enter the workforce.

A primary component of most software engineering courses is the software development project. For many computer science programs, this is the first opportunity that students have to work together to build a software system of significant size. The course

project reinforces the lecture material, making concrete for students the necessity and utility of the tools, processes, and techniques for large software system development.

Of the various models for course projects, the “large project” model often teaches students the most about software engineering [6]. Unfortunately, the overhead of managing a large group of students can make this model a difficult one to implement, even when the professor makes a conscious effort to delegate the work. As a result, many educators opt for an alternative model, such as the “small project” model, where small groups of 4-6 students develop smaller software systems, or perhaps even the same small system.

We believe that smaller projects and teams lack essential elements of realism. Students fail to see the need for key software engineering activities such as up-front design, documentation, version control, etc. They fail to understand the importance and difficulties of communication. Most importantly, the students do not experience the issues that arise when no one person can fully understand the system. Instructors are therefore placed in a difficult situation, balancing the desire to provide a realistic project experience with the practical limitations of a classroom setting.

In this paper we describe our approach for integrating a large-scale development project into a one-semester software engineering course. Our approach results in a course that exposes students to some of the issues they will face on real-world projects, and yet requires only modest amount of additional management overhead on the part of the instructor. We describe our project management structure, the development process we used, and its interaction with the associated lectures. We follow the advice given in [7] that recommends against projects that are “known quantities,” choosing large projects that have not been tried before.

We evaluated our approach in a software engineering course at The College of William and Mary during the spring of 2004. During this course, a class of twenty-one undergraduates and three graduate students developed a billiards simulator in Java with 2D and 3D views of the table, and support for multiple games. In addition to our own self-evaluation, we also provide an evaluation of the approach from the students' perspectives, based on survey results collected at the conclusion of the course. Overall, the course was a success in that students learned key software engineering concepts, reinforced by a project in which they dealt with issues such as communication within a large group, working within a management hierarchy, and using new technologies and tools.

The next section characterizes the difficulties associated with large projects in more detail. In Section 3 we describe our approach and how it addresses some of the issues enumerated in the previous section. The last two sections contain an evaluation of our approach and conclusion.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCSE'05, February 23–27, 2005, St. Louis, Missouri, USA
Copyright 2005 ACM 1-58113-997-7/11/00002 ...\$5.00.

2. LARGE PROJECT CHALLENGES

Unfortunately, most computer science programs emphasize software development skills (“programming”), devoting only a semester or two to *software engineering*. As a result, students only experience software development as a low-risk activity in which small systems are developed by one or a few people, during all-night coding sessions the day before a deadline. Because the systems are small, planning activities such as developing the requirements, specifications, and design are not necessary. Indeed, a single person can easily understand the whole system, and possibly implement it as well. In the end, students know that their software has no long-term use, so the focus is on providing enough quality to pass a professor-supplied test suite.

The professor of a software engineering class must work hard to change these expectations, usually in the student’s final year. One good way to do this is to scale up the size of the software project so that students can not possibly use the same techniques that they used in other computer science courses. At the same time, professors must accommodate the realities of a pedagogical setting.

One issue that immediately emerges as the size of the project increases is the additional management overhead. The professor is likely tempted to assume the role of manager because he or she wants to help the students succeed. However taking on all of the responsibilities of a manager (and perhaps the customer as well) is an additional burden on top of the usual responsibilities of teaching.

Unlike true software development projects, professors lack a number of key capabilities of most managers. Managers have the power to interview, hire, and fire project members. In contrast, professors can not hand-pick the students who will take their courses, nor can they “fire” students who are not doing their share of the work. Managers also have the responsibility of performing employee reviews, which can be subjective. Conversely, a professor should ideally have an objective approach for evaluating individual student performance in a group project.

A key difference between professional software developers and students is that professional developers are likely to be far more motivated than students simply because they are being compensated with a salary, benefits, bonuses, etc. Students are not being compensated in a tangible way; they are simply trying to earn an acceptable grade and hopefully trying to learn new topics and concepts. Thus, students do not have the same incentives to perform well as professionals, particularly if the course is one in which the student does not have a high interest. Further compounding this problem is the fact that at many institutions, such a course is not offered until a student’s last semester when “senioritis” kicks in.

Another issue is that most professional software engineers are full time employees who can devote each and every hour to the project(s) at hand. In fact, sometimes more than a standard forty hour work week is given to the project by the developer when overtime is required. In contrast, students have multiple courses and varying schedules. As a result, they can not be expected to treat a course project as a full-time job, and can have a difficult time scheduling meetings. This can have a significant detrimental impact on team communication and collaboration, which is an essential component of large software system development.

Finally, where the ultimate goal in industry is the successful completion of the project, the key goal in the classroom must be the education of the students, even when trying to model a real life setting. In industry, a true division of labor is likely to exist: one group works on requirements, one on design, and one on testing. However, limiting students to working on just one of these tasks prevents them from experiencing other activities.

In terms of the project itself, professional developers are often maintainers of software, spending a large amount of time learning the domain and the existing system. Although such an experience would be beneficial to students, the time constraints of a single semester course leave little opportunity for students to acquire detailed domain-specific knowledge, or to become intimately familiar with a large legacy system. Students may also view maintenance as uncreative and uninteresting compared to new development, which could adversely affect student motivation.

These factors combine to present an interesting set of requirements for software engineering instructors who wish to provide a realistic large-systems development experience, while accommodating the realities of an academic setting. Students should work on a system that is large and complex enough that they must specialize their skills and knowledge, and work together in its development. On the other hand, all students should participate in key development activities to ensure that they acquire important practical experience. The project should not require a large amount of domain-specific knowledge, and should be interesting to students. It must be possible to evaluate students individually, despite their group effort. Because of the risk of such a project, the development process and customer must be flexible enough to change requirements as necessary. The course should expose students to state-of-the-art tools supporting fundamental software engineering concepts.

3. APPROACH

In this section we present our approach, which attempts to address the difficulties and requirements described in the previous section. Overall, the course is taught in a single semester, meeting three times a week for 50 minutes, for a total of about 40 meetings. Mondays and Wednesdays the instructor presents lecture material, and Friday is devoted to the project. In addition, small groups of students meet in team meetings for two hours a week.

3.1 Scheduling

The lectures are planned to integrate with the project. The lectures in the first half of the semester provide an overview of the phases of the software development lifecycle, with an emphasis on object-oriented analysis and design. The second half of the semester is devoted to more in-depth lectures on special topics such as formal methods, software architecture, and project management.

Our rationale for this approach is that students must quickly acquire a working knowledge of software engineering in order to become productive in the course project. For example, students must develop the initial requirements, specification, and design for the software early in the course. Without some classroom exposure to these concepts, students will have difficulty performing these tasks well. At the same time, these project activities can not be delayed due to the limited amount of course time.

Overall students work on different aspects of the project. However, we must also ensure that every student learns the fundamental material. For key activities such as requirements definition and system design, we duplicate the project work class-wide in the form of homework assignments. We then either choose the best product to use for the project, or synthesize the project standard from a few of the best submissions. Similarly, we use the first few homework assignments to ensure that every student learns the required development tools. We believe this approach allows students to exploit parallel development on different parts of the project, while still ensuring that every student experiences the most essential elements of software engineering.

The only exception to the use of class-wide homework is the development of the initial high-level design. This work is done by the

managers and team leaders during the first few weeks. We believe that a small team of designers can function more efficiently, rapidly creating the framework for the rest of the development effort. This approach also helps to ensure that the managers and team leaders have a shared vision for the product.

At the same time, the rest of the class is learning about the tools that will be used during project development, and brushing up their programming skills. Example tools include the integrated development environment, and software for documentation, testing, and version control. We seek to update the toolset every time the course is offered, so that students are exposed to state-of-the-art tools. However, as other authors have suggested [2, 8], our emphasis is on teaching the fundamental concepts of software engineering, rather than simply learning various tools and technologies.

3.2 Project Management

Project management is performed by students. The professor only makes “command decisions” when intervention is absolutely required. The professor may also serve as the customer, validating the requirements document, test plan, etc., negotiating features and schedule, and evaluating the quality of the resulting software. Of course, the professor is also responsible for resolving course-related administrative issues.

For a class of approximately 30 students, we use three to five managers. The managers are responsible for tasks such as client communication, reallocation of students to teams, setting milestone goals and deadlines, and running the Friday project meetings. Importantly, the managers identify work tasks, assign them importance and timeliness values, and certify their satisfactory completion. As we will soon discuss, this process is essential for reducing most of the management overhead, providing a student evaluation mechanism, and tracking the progress of the project. Managers do little if any software development.

Students self-organize into teams that generally correspond to the major modules of the system, or activities in its development. Each team has a technical lead, chosen either by the team or the managers. The technical lead runs team meetings, performs detailed scheduling of work, communicates with the managers, ensures the quality of the team work, and helps to resolve technical difficulties. Technical leads are hands-on leaders who perform some software development in addition to their other duties. A team may consist of one person, as is the case with the single “buildmaster,” who manages the build process for the software.

Developers do most of the software development. They attend all team meetings, and communicate issues weekly to the team lead. Developers are encouraged but not required to work in pairs, especially if a particular developer needs help becoming familiar with the part of the system they are working on. Of course, any student in the class can skip the management hierarchy in order to bring issues to the instructor’s attention.

We believe that this approach works well for students, who can choose teams and activities that they want. For example, some students may wish to become adept at documentation, or perhaps a particular module of the system. At the same time, students can freely move from one team to another as their interest in a particular team’s work wanes, or as the remaining work is completed. At the same time, student team self-selection, along with the evaluation technique described later, ensure that managers are largely relieved of the difficulties of allocating students to teams.

We have the undergraduates complete short surveys to determine their technical strengths, interests, and prior development experience. Based on survey responses we select several students to serve as managers or technical leads for sub-modules of the system.

Team building exercises can also be used early in the semester to help identify the best candidates for management or technical leadership. During the semester, the professor and managers can adjust the management hierarchy as necessary. However, unlike real software development organizations, the professor can not easily hire or fire developers.

3.3 Improving Communication

The team meeting period provides a predefined meeting time for the students. This is crucial in that it provides a guaranteed time for the students to work together in person. Students are free to move from one team meeting to another as their interests change, or if they must resolve cross-team issues. The meetings are therefore scheduled to accommodate the differing schedules of the evolving team members. Managers also meet each week with the professor to discuss the project and resolve any issues that have arisen. Managers also use this time to plan the Friday meetings.

We also advocate several additional mechanisms to improve project communication. A web-based discussion forum provides fast communication, and is easily archived and searched. Students should also configure the version control system so that it notifies them of changes made by others to the modules they are developing. Students should attempt to schedule similar work times outside of the team meetings so that they can communicate via instant messaging when they can not be physically together. We believe that these mechanisms help to ease communication difficulties that students face.

Finally, the Friday classes are devoted to cross-team project communication. In the first few Friday meetings, some students research the project development tools, and present an overview to the rest of the group. Later class meetings are used to elevate important issues from teams to the entire class, and to update the rest of the class on developments within teams.

3.4 Development Process and Grading

We use an agile software process model based on Extreme Programming (XP) [1]. Our primary departures from XP are a more significant emphasis on documentation and up-front design. We believe that this approach is flexible, allowing unexpected risks to be quickly resolved and functionality to be modified as necessary to meet the hard deadlines of an academic course. At the same time, students are exposed to the more traditional notions such as requirements analysis and documentation. (The documentation is evolved along with the code, “faking” the up-front design process as described by Parnas and Clements [5].)

A key component of managing the development process, evaluating the contributions of individual students, and assessing project progress is the *issue tracker*. The issue tracker is a common database of all work that must be completed for the project. Accessed via the web, anyone in the class can view the list of work to be done, create new tasks for the list, and assign themselves to a task. Managers assign each task a 1-10 value for the priority and timeliness, as well as a modifier value. The overall point value of a task is the priority times the timeliness plus the modifier. Managers establish their own guidelines for proper values for these variables, and periodically evaluate each other to ensure consistency.

As developers complete work, they change the status to “Waiting for Technical Lead.” At this time, the team leader checks the work to make sure that it has been done satisfactorily, and then promotes the task status to “Waiting for Manager.” The manager then reviews the task, potentially modifies the point total, and then closes it. At this time, the points for the task are divided evenly among the people assigned to the task.

The task management system is based on the open-source Issue Tracker software [9]. We heavily modified the software in order to support a number of features we needed, including custom reports that automatically compute the project grade for each student in the class. We compute a student's grade as the average of (1) the percentage of completed points of the total points for all tasks in the system, and (2) the percentage of points that the student has earned for their share of the work. There are a number of additional details that space does not permit us to discuss; they will be presented in a forthcoming paper. For example, one must compensate for "overachievers" who do more than their share of the work, thereby "stealing" earnable points from other students.

We believe that this system provides managers with significant leverage in order to motivate the students. Instead of motivating developers with money, managers can give bonus points for exceptional work, or penalize poor work by subtracting points. The system is also extremely flexible, allowing managers (and the professor) to add all manner of work to the system that needs to be done. For example, tasks can be created for cleaning up the code, writing documentation, fixing a broken build process, etc. From the student's perspective, they can be sure that they will be rewarded for hard work that they do, and that their classmates who work less hard will be rewarded as appropriate.

4. EVALUATION

4.1 Use of the Approach: Spring 2004

We applied the approach presented in the previous section during the spring of 2004 at The College of William and Mary. The class is cross-listed as a graduate course, and consisted of 22 undergraduates and 3 graduate students. The graduate students were part time students, who had all developed software at some time during their employment. The undergraduates consisted mostly of junior-level students.

The project chosen by the instructor was a billiards game. (The professor served as the customer in this case.) The project involved a degree of difficulty for the students as it would require them to deal with graphical user interface issues, both 2D and 3D graphics, and physics to simulate the movement of the pool balls. Early on the requirements included a networked version of the game, but this was dropped fairly quickly as it became clear that achieving that functionality would greatly sacrifice the quality of the software.

The development schedule for the course was divided into three milestones. The first milestone required completion of the 2D 8-ball billiards game. The second milestone required the students to produce a 3D prototype and have the 2D view completed. The last milestone called for the completion of the 3D view and the addition of multiple games. We assigned homework early on to familiarize the students with the tools of the project: CVS for version control, JUnit for testing, Apache Ant for build management, and Issue Tracker for task tracking. As the project went on, students also adopted additional tools, such as Jalopy for automatically reformatting code, and Eclipse as the IDE. Later homework had the students rigorously document the requirements of the system, develop a set of use cases, finish the initial design in UML, develop a test plan, and write a formal specification of the billiard balls and the table, along with key operations on them.

In the issue tracker, the students completed approximately 400 tasks consisting of 2800 points total. Students completed approximately 84% of milestone one by its due date, about 96% of milestone two, and about 90% of milestone three. The key difficulty for the first milestone was a slow start for the project work. We believe that these grades were reflective of the quality of the software.

In the end, the students were able to complete a playable billiards game having both 2D and 3D views, good user documentation, a fairly good software architecture. Implementing proper physics turned out to be too challenging: the results were incorrect in circumstances involving a ball moving at high speed and many collisions (e.g. during the break). Some game rules were not implemented correctly, and there were also cross-platform speed and CPU consumption issues.

4.2 Self-Evaluation

4.2.1 Successes

Overall, we believe our approach was successful in that the project provided all of the students with practical experience with key software development activities, while forcing them to work together as a class to develop approximately 22,000 lines of code (6,000 non-comment, non-whitespace lines). They experienced the difficulties of a large project in which no one person fully understood the entire system. The students were exposed to the technical challenges of defining requirements and creating a design for a large system, and integrating independently developed modules. They also gained experience working with unfamiliar code, both in terms of each other's code and that of the Java3D platform. They learned to use a number of essential software tools, and gained experience working within a management hierarchy and communicating effectively within a large group.

At the same time, the cost of delivering a more realistic project experience was minimal on the part of the professor, whose project-related duties were largely confined to grading the homework assignments (i.e. reviewing requirements, designs, etc.), and grading the milestones by determining whether the milestone requirements were met. Managers performed the day-to-day project management, and the professor only intervened when absolutely necessary.

While our graduate student managers were more experienced in software development than the undergraduates, they were not necessarily more skilled at management. While we have not yet evaluated the hypothesis, we believe that it may be possible to identify suitable managers from the undergraduate class. For example, we found that undergraduate team leaders were quite effective, allowing the managers to delegate many of their decisions regarding the completion of tasks. (On the other hand, managers liked managing the completion of tasks because it allowed them to maintain intimate knowledge of the progress of the project.)

In our subjective judgment, the point system we devised was very effective at both tracking project activities and evaluating the contributions of individuals. The system was sufficiently flexible. We were able, for example, to give bonuses following each milestone to those people who performed exceptionally well. We also feel that the system accurately measured the contributions of each individual, and the group as a whole.

4.2.2 Challenges

However, the system and our use of it can still be improved. For example, the current system does not handle sub-tasks or dependencies between tasks. We also found that it was best to assign intermediate deadlines to tasks to ensure that they would be finished in a timely manner. Items with short deadlines were worth more, encouraging students to complete them sooner. This helped to overcome the procrastination we observed, where the students would try to complete most of the work a few days prior to the deadline. We also found that we needed to improve the traceability of tasks to requirements, to help ensure that all the requirements would be met in the final system.

Our choice of project for the course was not optimal. Initially, we thought the physics component of the game would provide an interesting bit of technical difficulty. Instead, we found that accurately modeling physical interactions was overly difficult, consuming more time than necessary. Otherwise, the choice of a game was good at capturing the interest of the students.

For the spring 2004 course, we did not use scheduled team meetings. We added this component to our approach in response to the most significant difficulty students faced: scheduling conflicts that made it difficult to meet and work on the project together. We had believed that the Friday meetings would provide some communication time, and that students would self-organize outside of class. We now believe that forcing the team members to adhere to a weekly meeting time is essential.

One surprise we encountered was the strong impact of individual personalities on the overall success of the project. For example, people who were content with lower grades frustrated those who wanted the project to be fully complete. We also found that timid people tended to remain disengaged, never becoming active participants. Further, skilled developers may have been best left as developers instead of team leaders because the additional management duties adversely affected their productivity

4.3 Student Feedback on the Project

At the end of the course, we surveyed the students to get their evaluation of the approach. The graduate students who served as managers had a generally positive experience. They felt that the lessons they learned were useful and interesting. One of the graduate students, who also works full-time as a software developer, commented that she has been able to directly apply lessons she learned from this course to her job.

We asked the students to rate various aspects of the course in terms of their usefulness. The ranked averages, starting with the most useful, are as follows: version control, weekly meetings, intermediate deadlines, issue tracker, discussion forums, the management hierarchy, pair programming, coding standards, unit testing, project documentation, software ownership, prototyping.

The students reported in the surveys that the course helped them in a number of ways. It helped them learn a number of tools, and forced them to become involved in software development. They also learned the importance of planning and communication in a real-world project. However, the students reported that the combination of homework and project was too much work, and that more up-front design should be done in order to allow independent work to begin sooner. The managers suggested the use of weekly team meetings sections to help alleviate scheduling problems. The top four challenges the undergraduates reported were communication, scheduling, differing visions for the project, and differing levels of commitment to the project. In fact, one student said that the professor's warning about communication problems was the "understatement of the year."

The students also felt that the lack of a domain expert negatively affected the realism of the physics modeled in the project. They felt that working in a large team is too difficult, and as a result, the workload among students was uneven, scheduling conflicts among students made person-to-person interaction and communication difficult, and that management overhead prevented meaningful feedback on the quality of the design prior to implementation. The undergraduates suggested that we use the system as a starting point for the next class, extending it in some way or even evolving it into a different game that uses similar constructs. This would allow the next class to experience software maintenance issues.

5. CONCLUSION

Providing students with meaningful development experiences in software engineering courses is essential if we are to produce graduates who can enter the workforce and be productive. While the model we have presented has its problems and will continue to be refined, we believe that it is a good first step toward this goal. We believe that it provides students with a software development experience that most had never encountered, at a reasonable cost to the instructor's time.

We have adopted the suggestion of team meetings to improve teamwork. We also plan to make individual schedules a more significant factor in the allocation of people to teams. The students also suggested that we use smaller teams and projects, intuitively understanding that this would ease management overhead. However, this misses the point of what we are trying to accomplish. For many software development efforts, a smaller team is not a luxury one can afford.

We do agree that maintenance was an important component of software engineering that was absent from this course. However, it may be hard to motivate students to perform maintenance on a system with which they have no emotional investment, especially if that system will be discarded part way through the semester to start a new project. A feasible alternative would be to begin the next offering of the course with part or all of the system that was developed by the previous class, and have the next class modify or extend it in some way. For example, the next class could use the physics module to build a bowling game.

6. ACKNOWLEDGMENTS

The authors thank Meghan Revelle for her insightful comments on a previous version of this paper. We also thank the anonymous reviewers for their valuable comments, some of which could not be addressed due to space limitations.

7. REFERENCES

- [1] Kent Beck. Embracing change with Extreme Programming. *IEEE Computer*, 32(10):70–77, October 1999.
- [2] Bertrand Meyer. Software engineering in the academy. URL: <http://archive.eiffel.com/doc/manuals/technology/bmarticles/computer/teaching.pdf>.
- [3] David Lorge Parnas. Software engineering: An unconsummated marriage. *SIGSOFT Software Engineering Notes*, 22(6):40–50, November 1997.
- [4] David Lorge Parnas. Software engineering programmes are not computer science programmes. *Annals of Software Engineering*, 6(1–4):19–37, April 1999.
- [5] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–7, 1986.
- [6] M. Shaw and J. Tomayko. Models for undergraduate project courses in software engineering. Technical Report CMU/SEI-91-TR-010, Software Engineering Institute, 1991.
- [7] Mary Shaw. We can teach software better. *Computing Research News*, 4(4):2–4, 12, September 1992.
- [8] Mary Shaw. Software engineering education: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering—The Future of Software Engineering*, pages 371–80, Limerick, Ireland, 4–11 June 2000. IEEE.
- [9] TuxMonkey.com. The issue tracker homepage. URL: <http://www.issue-tracker.com/>.