Computer Science Education
Vol. 16, No. 1, March 2006, pp. 53–73

Routledge
Taylor & Francis Group

Implementing Large Projects in Software Engineering Courses

David Coppit*
Department of Computer Science, The College of William and Mary, USA

In software engineering education, large projects are widely recognized as a useful way of exposing students to the real-world difficulties of team software development. But large projects are difficult to put into practice. First, educators rarely have additional time to manage software projects. Second, classrooms have inherent limitations that threaten the realism of large projects. Third, quantitative evaluation of individuals who work in groups is notoriously difficult. 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 20 to 30 students work together to construct a moderately sized (15 KLOC) software system. The approach combines carefully coordinated lectures and homeworks, a hierarchical project management structure, modern communication technologies, and a web-based project tracking and individual assessment system. Our approach provides a more realistic project experience for the students, without incurring significant additional overhead for the instructor. We present our experiences using the approach the last 2 years for the software engineering course at The College of William and Mary. Although the approach has some weaknesses, we believe that they are strongly outweighed by the pedagogical benefits.

1. INTRODUCTION

Software continues to play an increasingly vital role in the society, and because of this, the demand for skilled software engineers persists. Unfortunately, those who enter industry as software developers are often ill-prepared for the work they are expected to perform (Parnas, 1997, 1999). A primary cause of this problem is the difficulty of providing the students with a realistic software development experience in an educational setting.

This difficulty is most apparent in the capstone software engineering course of many curricula. This is often the first opportunity that students have to work together to build a software system of significant size. Ideally, educators would employ the "large project" model of software engineering education, where multiple students collaborate

*Corresponding author. Department of Computer Science, The College of William and Mary, Williamsburg, VA 23185, USA. E-mail: david@coppit.org

ISSN 0899-3408 (print)/ISSN 1744-5175 (online)/06/010053-21 © 2006 Taylor & Francis
DOI: 10.1080/08993400600600443

on the development of a significant software system (Shaw & Tomayko, 1991). By using this approach, students better see the need for key software engineering activities such as up-front design, documentation, version control, etc. They also better understand the importance and difficulties of communication. Perhaps most importantly, the students experience the problems that arise when no one person can fully understand the system.

Unfortunately, there are several significant difficulties with the large project model. First, the overhead of managing a large group of students can be overwhelming for an instructor, even when he or she makes a conscious effort to delegate the work. Second, a classroom setting is not a business setting, and students are not professional software developers. Finally, large teams make it more difficult for the educator to accurately assess the performance of individual students.

As a result of these difficulties, many educators opt for a less realistic 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. Unfortunately, this only reinforces the view that many students develop of software engineering 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 poorly motivated.

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 a modest amount of additional management overhead on the part of the instructor. We describe our project management structure, our student evaluation methods, the development process we used, and the interaction of the project with the associated lectures and homework. We follow the advice given in (Shaw, 1992) 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 2004 and 2005 semesters. In the first year, 24 students developed a 22 KLOC billiards game. In the second year, 21 students developed an 8 KLOC automobile traffic simulator. We present our experiences and insights regarding the use of the approach. We conclude that although there are some risks and difficulties associated with the approach, on balance the pedagogical benefits to the students greatly outweigh the potential problems. We feel that the approach helps students to learn key software engineering concepts, reinforced by a project that deals with issues such as communication within a large group, working within a management hierarchy, and using new technologies and tools.

In the next section we further motivate our work by describing the challenges of employing a large team approach to software engineering education. We then describe our approach and how it addresses the difficulties of large team projects. Next, we describe our project tracking and student assessment system in more detail. We then describe our experiences employing the approach, followed by an evaluation and conclusion.

2. THE DIFFICULTIES OF LARGE PROJECTS

2.1. Practical Limitations of the Classroom

There are a number of difficulties that hinder the use of large projects in software engineering courses. One issue that immediately emerges is the additional management overhead. The instructor 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.

Secondly, the classroom has constraints that do not typically encumber professional software development. Instructors cannot selectively "hire" or "fire" students. Unlike managers who can evaluate developers subjectively, instructors should evaluate individuals in a quantitative, objective manner. Students are not full-time developers, and are not motivated by salary or benefits. (Although they can be motivated by a grade, the significance of a grade varies from student to student.) Students also have widely varying schedules that can hinder effective teamwork, and the semester has a hard deadline for the project's completion. Finally, the key goal in the classroom must be the education of the students, rather than the completion of the project. For example, some work must be done by all the students to provide them all with a standard set of experiences.

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 and time constraints 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.

2.2. The Challenges of Individual Evaluation

Any time students work together on a group project, instructors must face the challenge of accurately evaluating the work of individual students. In this subsection we elaborate upon the criteria described by Hayes, Lethbridge, and Port (2003). We also characterize certain student behaviors that affect evaluation methods.

Perhaps the most important challenge is that the system be fair. It should be quantitative and as objective as possible, with little grader bias or arbitrariness. The system should not be easily manipulated by students in order to achieve a particular grade for themselves or others. Students should feel that they receive the grade that they earn—good or bad.

The system should also be consistent. Ideally, it should be independent of the type of project. It should be robust in the face of uncontrollable factors such as project size and scope, changing technology, etc. It should also be independent of the overall quality of the class. It should enable the grading of different types of work, such as documentation, team leadership, software implementation, testing, and version management.

The system should be open to scrutiny, and provide students with timely and understandable feedback. The more responsive the system is, the more quickly students and the instructor can receive feedback on the student's progress, and take any necessary action. The system should also reflect the educational objectives of the course. Importantly, the system must not require an inordinate amount of work for the instructor. It should scale well to different sizes and numbers of groups. It should be possible to delegate some of the evaluation work to trusted people such as teaching assistants.

The assessment approach should encourage the behaviors that lead to effective teamwork. Students should be allowed to specialize their expertise in certain aspects of the system or certain types of work. At the same time, the instructor should be able to require all students to participate in certain activities for pedagogical reasons. The evaluation approach should encourage high quality work, and instill a sense of pride and accountability in the students for the work that they do.

Unproductive student behaviors should be discouraged by the grading approach. The following descriptions provide a useful characterization of some typical student behaviors that can affect a grading technique:

- Hitchhiker (also know as a slacker or freeloader): Seeks to maximize their grade while minimizing the amount of work that they must do. This type of student is a common source of much bitterness on the part of his or her group members (Schultz, 1999), because the group members must do an inordinate amount of unrewarded work in order to make up for the hitchhiker.
- Overachiever: Seeks to do as much work as possible, even beyond that which is required to achieve the highest possible grade. This type of person can make it difficult for other students to contribute in a meaningful way.
- Underachiever: Does not perform their full share of the work, and is content earning a lower grade (if that grade is properly given).
- Procrastinator: Fully intends to do his or her share of the work, but only immediately prior to a deadline. This makes it difficult for team members to plan work that depends on that of the student—forcing everyone to become a procrastinator.
- Dilettante: Becomes involved in many portions of the project, but only in a superficial way, never fully completing their portion of the work. Other students can be overwhelmed or intimidated by the student, choosing not to contribute toward work that is already "taken."
- Tutor: A stronger student who knowingly helps an underachiever or hitchhiker to achieve a higher grade, perhaps by doing the weaker student's share of the work.

- Conspirator: One of a group of students who collude to exploit the grading system to achieve a higher grade.

3. APPROACH

In this section we present our approach, which attempts to address the difficulties 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. To date we have employed the approach for class sizes of about 25 students. Mondays and Wednesdays the instructor presents lecture material, and Friday is devoted to the project. In addition, small teams of students meet in team meetings for two hours a week.

3.1. Choice of Project

Choosing a good project is challenging due to the many properties it should have. First, the amount of work for the project should be appropriate to the size of the class. One should keep in mind that the work involved goes beyond just programming to also encompass other activities such as testing, documentation, build management, and maintaining the project website. The requirements of the project should be flexible, allowing nonessential requirements to be trimmed to meet deadlines. At the same time, the requirements should be fairly independent, so that students can more easily work in parallel.

We recommend that a project be broken into four to five major milestones. The last milestone should be slightly beyond that which the instructor believes the students would normally accomplish, in case the class exceeds expectations. The remaining milestone can be dropped later as necessary. Instructors should carefully consider whether the project requires specialized domain-specific knowledge. If so, this can be a significant development bottleneck and obstacle for the success of the project.

As an example of a reasonably good project, consider the traffic simulator project we used for the Spring of 2005. The concepts are readily accessible to students, and the project has a certain entertainment value. Students can work relatively independently developing the automobile, roads, intersections, simulation, and graphical user interface. The project can be structured in several milestones: for example, (1) vehicles on a single lane, straight road, (2) vehicles on a multiple lane road, (3) vehicles changing lanes on a curved road, (4) vehicles moving through intersections, (5) vehicles with artificial intelligence algorithms for driving to specific destinations, routing around traffic congestion. The only technical challenges are the mathematics used to model the behavior of drivers, and the implementation of a time-step simulation. In order to mitigate the mathematics risk, we used a model that had already been developed and described in a published paper.

3.2. Project Management

Project management is performed by students. The class is structured in terms of teams of 3 – 5 developers, a team leader for each team, 1 – 3 high-level managers, and

the instructor. The instructor only makes "command decisions" when intervention is absolutely required. The instructor 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 instructor is also responsible for course-related administrative issues.

The managers are responsible for activities such as team motivation, client communication, setting milestone goals and deadlines, resolving personnel issues, monitoring ongoing risks and issues, ensuring accurate task descriptions, and maintaining the overall conceptual integrity of the software. Managers meet weekly with the instructor and team leaders to discuss and resolve high-level issues. Managers do little if any software development, but rather provide design consultation for teams.

The choice of student managers is moderately important, and should be considered carefully by the instructor. For courses cross-listed at the graduate level, graduate students are an obvious choice for management. The instructor can also use previous experience with students, student skill surveys, and other selection criteria. Management can be changed at the completion of any milestone. The stigma of being "demoted" can be removed by characterizing the change in management as a pedagogical decision whose purpose is to expose more students to managerial responsibilities.

Students self-organize into teams that generally correspond to the major modules of the system, or activities in its development. That is, students choose to work on any task that interests them. Very quickly the students find themselves working consistently with others that have the same interests, and a team is dynamically formed. We have found that such teams tend to form around major activities (such as documentation or build management) and major modules of the software. As the work for a team is completed, it naturally dissolves as people move on to other tasks that interest them.

Each team elects a team leader. The team leader runs team meetings, performs detailed scheduling of work, attends weekly meetings with the managers and instructor, ensures the quality of the team's work, and helps to resolve technical difficulties. Technical leads are hands-on leaders who perform some software development in addition to their other duties. Team leaders identify work tasks, assign them importance and timeliness values, and certify their satisfactory completion. As we will soon discuss, the creation and monitoring of work tasks is essential for reducing most of the management overhead, evaluating the students, and tracking the progress of the project.

Developers do most of the software development. They attend all team meetings and communicate issues frequently to the team leader. Developers are encouraged but not required to employ pair programming (Beck, 1999), 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.

3.3. Lectures and Homework

The lectures and homework are coscheduled with the project milestones. 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, it would be difficult for students to perform these activities well. At the same time, these project activities cannot be delayed due to the limited amount of course time.

Homework assignments are based on the class project, and are used to ensure that all the students learn and practice important software engineering activities. For example, the students work in pairs to develop the requirements, specifications, and design. The managers then either choose the best product to use for the project, or (more commonly) synthesize the project standard from a few of the best submissions.

We also use the first few homework assignments to ensure that every student learns the required development tools. 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 (Meyer, 2001; Shaw, 2000), our emphasis is on teaching the fundamental concepts of software engineering, rather than simply learning various tools and technologies.

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 project.

3.4. Improving Communication

We employ an array of technologies and techniques to enable effective communication. For example, we require that teams meet for two hours each week. This provides a predefined meeting time that ensures that students can work in person together despite their conflicting schedules. 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 rescheduled as necessary to accommodate the differing schedules of the evolving team members.

We also use several technologies to improve project communication. A web-based discussion forum provides fast communication, and is easily archived and searched. Students can also configure the version control system so that it notifies them of changes made by others to the modules they are developing. Students exchange instant messaging screen names so that they can chat when they are working at the same time but in different locations.

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 class. 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. The Friday meetings are also used by the instructor to demonstrate tools, address administrative issues, and provide development and management advice.

3.5. Development Process and Grading

We use an agile software process model based on Extreme Programming (XP) (Beck, 1999). 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 trimmed 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 (1986).

We use approximately five major milestones during development, corresponding to the completion of major functional requirements. There are also weekly minimilestones. The key distinctions between the two types of milestones are (1) that the managers and instructor set the major milestones, whereas the team leaders set the minimilestones, and (2) project grades are assigned at the major milestones. At the end of each major milestone, the project points are reset, thereby allowing students who performed poorly for some reason to not be handicapped for the next milestone. We also assign bonus points for exceptional work at the end of each milestone. We believe that this approach balances the use of periodic grading, while still encouraging students to keep development on track.

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. The task management system is based on the open-source Issue Tracker software (version 4.0.4) (Taylor et al., 2005). We heavily modified the software in order to support our pedagogical needs. For example, we implemented a custom report that automatically computes the project grade for each student in the class.

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. As work is completed, team leaders and managers can track the progress of the project. Point values

associated with work also provide a quantitative measure of the contribution of individual students. We elaborate upon this aspect in the next section.

We believe that this system addresses a number of the difficulties of managing a large project. Because students assign themselves to tasks, they tend to self-organize along sets of related tasks that correspond to their team. At the same time, the assignments are dynamic, allowing team leaders and managers to create tasks on-the-fly for activities such as code cleanup, website development, and fixing a broken build. The point system is used to motivate and evaluate students. For example, if a task is important and timely, it will be worth more, and will therefore be more likely to be completed quickly by a developer. 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 appropriately.

4. INDIVIDUAL EVALUATION

In this section we describe our technique for evaluating the performance of individual students in a group context. As mentioned earlier, the system is based on the issue tracker task management system. Students earn points as they complete project tasks. These points are then used to compute the student's grade.

4.1. Overall Grade

Given the project-oriented nature of the course, we believe that the project grade should represent a sizable portion of the students' overall grade. For example, a reasonable breakdown might be 20% for the midterm exam, 20% for the final exam, 20% for homework, and 40% for the project. The project grade for developers and team leaders is computed as described below, and is the average of each of the milestone grades.

For managers, the overall project grade is broken down further to incorporate instructor and developer evaluations. For example, one could use 30% milestone grade and 10% developer evaluation. The developer evaluation is a chance for the developers to assess the managers. This can occur at the end of the semester, or perhaps at the end of each milestone.

4.2. Project Grade (Developers)

Tasks can be created by anyone in the class, at which point preliminary values in the range 1 – 10 are assigned to the *difficulty* and *priority* of the task. These values are multiplied, then added to a *modifier* value to compute the overall task point value. A task that is created by a student is initially owned by that student. Within about a day, a manager assumes ownership of the task, verifying the appropriateness of the initial student-assigned point values and adjusting them if necessary.

Point values accurately capture the essential qualities of different kinds of tasks. For example, a difficult but unimportant task might be assigned a difficulty of 10 and a

priority of 1. Using the default modifier of 0, the overall point value would be $10 \times 1 + 0 = 10$. Conversely, an important but easy task would have values of 1 for the difficulty and 10 for the priority, yielding the same overall point value. Modifiers are used by team leaders and managers to adjust the base point value. For example, a manager may add or subtract points for exceptional or substandard work, or may increase the overall point value to encourage developers to complete the task more quickly.

As developers complete work, they change the status of the task to "Waiting for Team Leader." 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 by adjusting the modifier value, and then closes it. At this time, the points for the task are divided evenly among the developers assigned to the task. Even division simplifies the mathematics, and prevents collaborators from arguing over relative contributions. We believe that at the granularity of individual tasks any student who is unhappy with a partner's contributions will simply not work with that person in the future.

We compute a student's grade as a combination 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. The former is a "group grade" that measures overall project completion, whereas the latter is an "individual grade" that measures the student's own contribution.

By weighing the relative contribution of these two scores, the instructor can control the extent to which the project is fully completed versus the accurate assessment of individual students. For example, using a 0% contribution for the group grade would result in the project being somewhat incomplete due to students who are content with less than a 100% grade. On the other hand, using a 0% individual contribution would cause some students to work very hard to complete the entire project, but would allow other students to hitchhike, earning a good grade for little or no work.

4.3. Project Grade (Managers)

Because managers do not perform tasks in the issue tracker system it cannot be used to evaluate their contributions to the project. One simple way to assign managers a grade at each milestone is to simply use the group grade—the rationale being that a good manager will help the entire class to succeed. However, as we discuss later, mismanagement of the project may cause tasks to not be entered into the system, resulting in a high group grade despite the low quality of the software. In this case, it may be better to set the milestone grade to be the percentage of requirements fulfilled at each milestone.

An alternative approach is to evaluate managers subjectively. This is feasible given that the instructor meets with the managers each week. One of the key criteria for evaluation is the extent to which the managers are managing the developers in terms of the task list. Task points should be consistent, and the tasks in the system should be reflective of the actual work that remains to be done. In the case where all the tasks are

complete but the software is of poor quality, the blame rests on the managers rather than the developers.

Other criteria for evaluation include the extent to which the managers provide technical guidance and leadership, their ability to resolve personnel issues, their planning of the minimilestones, and their identification of risks. If managers are changed, they should only be changed at the start of each milestone, so that students are evaluated using the same criteria for the entire milestone.

4.4. Example

There are several details which complicate the evaluation technique described above. We now present a small example that presents these details while computing individual grades. Note that this process is completely automated by the issue tracker report generator.

We begin with a group consisting of four students, devel_1, devel_2, devel_3, and devel_4. In addition, there is one student who has dropped the course, devel_5_drop. The task list is as follows:

ID	Summary	Points [Difficulty* Priority + Modifier]	Points/ Person	Status	Assigned To
1	Improve the appearance of the 3D objects	6 $(3*1+3)$	6	Waiting for Manager	devel_1
2	BUG: You can hit the 8-Ball first	8 $(1*4+4)$	8	Closed	devel_3
3	Implement "push out"	12 $(3*4+0)$	12	Closed	devel_1
4	Update top-level build.xml	6 $(2*3+0)$	6	Closed	devel_4
5	Place Cue Ball	9 $(3*3+0)$	9	Closed	devel_3
6	Generate JavaDoc hourly	12 $(4*3+0)$	6	Closed	devel_1, devel_2
7	Run Jalopy hourly	9 $(3*3+0)$	9	In Progress	devel_4
8	Milestone 2 Bonus	10 $(10*1+0)$	10	Closed	devel_2
9	Update documentation	3 $(1*3+0)$	3	Closed	devel_5_drop
10	Consult physics tutorial	25 $(5*5+0)$	25	Invalid	devel_4

Task 1 is finished and waiting for manager verification. Task 8 is a bonus from the previous milestone. The last task was deemed to be invalid. (Tasks can be invalid for a number of reasons, such as being a duplicate or being unnecessary.) Task 6 was performed by two people, who share the points.

Bonus points given to individual students do not affect either the group grade or the individual grades for other students. Similarly, points for invalid tasks, or those completed by nongroup members, do not contribute toward the grade in any way. Thus, the *earnable points* are computed by subtracting all bonus points for all students, points for invalid tasks, and points earned by nongroup members.

The total number of points from the task list above is 100. From this we subtract the 10 bonus points, 25 points for the invalid task, and 3 for the student who dropped the course. This leaves 62 earnable points. Of these, 47 points have been earned. This yields a group grade of 76%.

Next we compute the individual grades. Theoretically overachievers can complicate the grading process by "stealing" work from other students. Our initial version of this grading algorithm subtracted extra points earned by overachievers from the total earnable points in an effort to ensure fairness. As we will describe later, we have dropped this computation because it introduces additional complexity, is largely unnecessary, and can encourage procrastination on the part of non-overachievers.

Managers can encourage developers to perform a task by increasing the modifier value. If a task is not done despite an attractive modifier value, managers always have the ability to do the work themselves, in which case the points for the task benefit the group grade, but are factored out of the earnable points during the individual grade computation since managers are not part of the development group.

Student Name	Points Earned	Bonus Points	Target Points	Individual Grade	Group Grade	Final Grade
devel_1	18	0	15.5	100	76	88
devel_2	6	10	15.5	100	76	88
devel_3	17	0	15.5	100	76	88
devel_4	6	0	15.5	39	76	57

With a total of 62 points for the group and 4 students, this means that each student must earn 15.5 points. We compute the individual grades, dividing the points for any collaborative work, and adding in any bonus points. Once that is done, we can compute the final grade as a combination of the group and individual grades. In this case we use equal 50% weights for the two grades.

The overachievers devel_1, devel_2, and devel_3 have earned more than 15.5 points, but we cap their individual grades at 100%. (The additional work is rewarded in the increased group grade.)

Note how this grading system combines the group's progress with the individual's in order to compute the final grade. An overachiever is not guaranteed an A grade unless the other students also contribute more. For example, when devel_4 completes task 7, his individual grade will become 97%, and the group grade for all the students will become 90%. Other developers who are unhappy with their grade can elect to help devel_4 complete the work, or can create new tasks for other work that also needs to be done.

Note that the first task is waiting for manager approval. Before a milestone grade is computed, the managers complete their evaluation of any open tasks so as to not unfairly penalize developers who completed the task on time. The work performed by

devel_5_drop contributes toward the group grade, but does not otherwise affect the other student's grades since his points are not a part of the earnable points.

5. USE OF THE APPROACH: 2004 AND 2005

In this section we present our experiences using this approach during the 2004 and 2005 spring semesters. It should be noted that while the overall approach used was the same, certain details were modified as the approach was refined. For example, the first year we did not require teams to meet for 2 hours each week. In both years, the job of verifying the work associated with a task fell primarily on the managers rather than the team leaders. We also used the "overachiever compensation" scheme in the grading of individuals to factor out points that were "stolen" from the earnable point pool by overachievers. We will discuss the reasons for these modifications in detail in the next section.

5.1. Use of the Approach: Spring 2004

The class is cross-listed as a graduate course, and in the Spring of 2004 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. We used the graduate students as managers, and they were reasonably motivated and engaged in the project. The notion of team leaders emerged during the course of the semester as teams elected leaders.

The project chosen by the instructor was a billiards game. (The instructor 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 user interface design, 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 such functionality would greatly sacrifice the quality of the software.

The development schedule for the course was divided into three milestones. The first milestone required a prototype 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: concurrent version system (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 integrated development environment (IDE). Later homework had the students rigorously document the requirements of the system, develop a set of use cases, finish the initial design in the unified modeling language (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 2,800 total points. For each milestone, we estimated that the students completed approximately 84%, 95%, and 90% of the requirements, respectively. Each milestone usually had a few tasks that were not completed, in which case they resulted in a reduced group grade, and were carried over to the next milestone. We arrived at this estimation by assigning a scalar importance value between 1 and 5 to each itemized requirement, and computing the percentage of the total value of the requirements that was completed.

In the end, the students were able to complete a playable billiards game having both 2D and 3D views, good user documentation, and 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 having 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.

During the course of the semester we made minor changes to make the assessment more fair. As we mentioned earlier, we added team leaders as an additional level of management to help relieve the work of the managers. We also factored the bonuses out of the computation in order to avoid unfairly penalizing other students. After identifying the problem of the "dilettante," we modified the system so that it only allowed a student to sign up for at most 2 unclosed tasks at a time.

We were also forced to scale the students' grades based on the percentage of requirements completed, since we found that the issue tracker did not contain tasks for all the work that needed to be done. We discuss this issue in more detail in the next section. Finally, in order to help prevent procrastination, we also added "minimilestones" to the approach.

5.2. Use of the Approach: Spring 2005

The second time we employed this approach was the Spring of 2005. This year the class consisted of 21 undergraduates and 1 graduate student. In order to maintain a reasonable manager-to-developer ratio, we promoted two volunteer undergraduates to the management level. In this case none of the managers had any professional software development experience.

The project was an automobile traffic simulator. Given a road configuration, "source lanes" that generate cars at a particular rate, the simulator models the flow of traffic toward "sink lanes." Features include a 2D graphical interface with zoom and pan support, curved roads, roads with multiple lanes, and intersections. The students implemented a driver model and lane changing model developed by our customer, an expert in micromodels of traffic.

Based on our previous experience, we decided to partition the development into five rather than three milestones. As before, initial homeworks focused on the tools and design documents, whereas later homeworks related to the special topics of the lectures.

The students completed approximately 320 tasks worth 1,900 total points. For each milestone, the students completed approximately 100%, 92%, 91%, 78%, and

97% of the milestone requirements, respectively. This year, the managers were warned to be consistent and parsimonious in their use of points for tasks. As a result, the average number of points per task was 5 rather than 7 as in the previous year.

As in the previous year, the students were able to complete an impressive amount of collaborative work. Not only was the simulation realistic and interesting, the associated work products such as the documentation were also of high quality. Students encountered and overcame many of the communication difficulties that occur in sizeable projects, while also gaining experience developing a nontrivial system.

One significant problem that arose during the semester was procrastination, which hurt the developers the most in milestone four, where only 78% of the requirements were met. Figure 1 shows graphs of the individual and group grades for each developer prior to and after the "big push" to complete the fourth milestone. In these graphs, the students are simply ordered according to grade. The horizontal line represents the group grade. We see that prior to the final effort to finish the milestone, half of the students had not performed more than half their share of the work. Afterwards, nearly all had performed their share of the work in issue tracker.

A similar pattern occurred for each of the milestones in both years. In the first year we attributed this to procrastination, but we realized in the second year that our grading scheme contained incentives for students to delay working on the project. Comments from students indicated that at least some of them were exploiting the overachiever-procrastination factor in the grading scheme. We discuss this issue more in the next section. Fortunately, we believe that the modified approach that we present in this paper will address this problem.

This example also demonstrates a second difficulty with our approach. Despite the apparent 99% completion of the milestone, the software was of inadequate quality. The reason was that the issue tracker lacked key tasks to finish the functionality required for the milestone and to fix bugs in the code. This experience highlights one of the difficulties of the approach. Because tasks are added to the system as needed, the developers may falsely target only the tasks in the tracker, rather than the quality of the software itself.

In practice, we have taken two approaches for dealing with this situation. The first is to make it clear to the developers that the software will be assessed relative to the

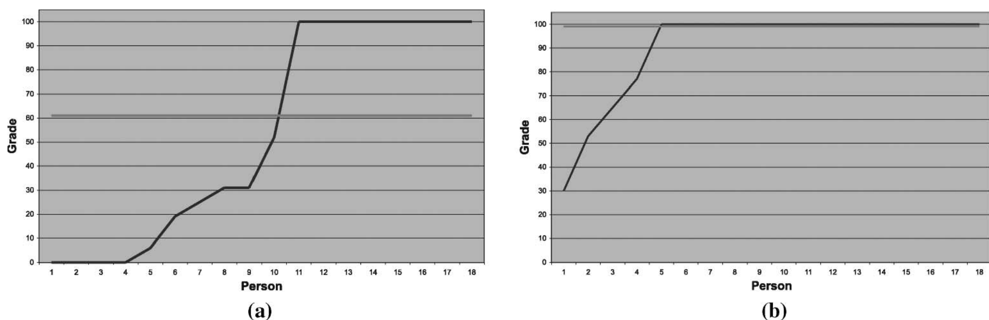

Figure 1. Grades for milestone 4: (a) before milestone wrapup, (b) after milestone wrapup

requirements in this case. This approach is less satisfactory since it involves additional work on the part of the instructor, and seems to indicate a deficiency in the grading system. The second, perhaps more realistic and effective, approach is to make it clear to the managers that their responsibility is to manage the work to be done. So a situation in which the resulting software is inadequate when the developers have done the work asked of them indicates a management failure. In this case, the manager grades should suffer, while the developers should receive the grades indicated by the issue tracker system.

6. EVALUATION

In this section we discuss the strengths, weaknesses, and challenges of our approach, based on our experience to date in its use.

6.1. 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 a sizeable system. 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 external libraries such as Java3D. 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.

The approach was also successful in the sense that it provided students with a realistic experience without a substantial amount of overhead for the instructor. Our 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.

We believe that this approach also 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.

In the first year, we were fortunate to have graduate student managers with software development experience. We found in the second year that undergraduates can perform adequately in managerial roles. The key challenge for managers in both years is lack of intimate knowledge of the day-to-day problems faced by the teams. It is for

this reason that we have modified our approach to give team leaders more decision-making power. (One could argue that such problems are part of the difficulties of management, and such a change should not be made.)

We believe that the grading system has a number of key strengths. First, we believe it can accurately measure the performance of individuals within the group effort. It is also quite flexible, allowing the instructor to assess students who contribute to the project in many varied ways. Students who work hard are primarily rewarded with higher individual grades, and also modest increases in the group grade. If all students contribute to the project, the group grade increases and everyone benefits. Bonuses help reduce the impact of resetting point values at the end of milestones. We believe that our approach is resistant to collusion because team leaders and managers directly assess the completed tasks.

Our approach also appears to be quite consistent, despite the fact that students performed a wide array of different work. There is less grader bias because the point values and requirements for completion of tasks are usually set before students sign up for tasks. We believe that the approach can scale well if one employs additional levels of management, although it may make more sense to divide a class of 100 students into three or four large subgroups before applying our approach. While we encountered some difficulties which we outline next, we are confident in the need for a quantitative system that directly ties the student's grade to the work that they complete. As far as we are aware, this is the first time that such an approach has been used to quantitatively evaluate a large group of students in real time.

6.2. Challenges

6.2.1. Task management. One of the key challenges of our approach is precise management of the tasks in issue tracker by the managers. This difficulty comes in several forms. First, points need to be assigned to similar tasks in a consistent manner. Second, points for different kinds of tasks should be assigned fairly. For example, managers must be careful that a simple documentation task is not rated at a higher difficulty than a complex technical issue. To help address this problem, we developed a set of guidelines for the priority points, and set the difficulty points as an estimate of the number of hours that it would take for a manager to complete the task. Managers also periodically cross-checked each other to help ensure consistency.

Third, we found that managers are often too far from the detailed day-to-day activities of developers. It was for this reason that we shifted the job of task verification to the team leaders, who are better equipped to evaluate completed work. This leaves the management of the high-level design and ensuring the consistency of points to the managers.

The fourth task management difficulty is the entry of tasks into the issue tracking system. During both years, we found that at nearly every milestone the issue tracker reported 100% completion, but the software was obviously missing functionality or poorly implemented. To date we have addressed this problem by estimating the percentage of milestone requirements that were met, then scaling the students' grades

by this amount. However, we feel that this technique only hides systemic problems with our approach.

We now feel that a better approach would be to create incentives for managers and team leaders to be more vigilant about creating tasks for work that needs to be done. If tasks are created properly, the existing incentives for developers will ensure that the system works correctly. One way to create these incentives is to notify managers that they will be evaluated in terms of the percentage of requirements that are met at each milestone, not the grade in the issue tracker system. This will firmly place the responsibility of task management on the managers, and will avoid unfairly "punishing" developers who otherwise believe that they are succeeding in their development efforts.

6.2.2. Overachiever compensation. In the original design of our grading system, we were concerned that the work for a milestone was finite, and that overachievers who do more than their share would in effect "steal" earnable points from other students. To accommodate overachievement, we computed the number of extra points earned by these students and removed them from the pool of earnable points. This resulted in lowering the target points per person for others in the class.

We discovered in the second year that this method encouraged some students to procrastinate. These students found that more motivated students in the class would do more than their share of the work in order to improve the overall group grade. As a result, these extra points would be removed from the total earnable points, thereby making it easier for procrastinators to earn a good grade by doing less work just before a milestone deadline. For example, during the second year overachiever compensation reduced the number of points that needed to be earned by 28% on average. It was clear that overchiever compensation was unfairly giving procrastinators good grades while simultaneously increasing the workload for hard-working students.

There are two possible ways to address this problem. The first is to reduce or remove the portion of the student's overall grade that is based on the group grade. This will cause hard-working students to perform less extra work. Another approach, and the one that we believe is more effective, is to remove the overachiever compensation entirely. We justify this change based on three arguments. First, it simplifies the grading system. Second, earning points for a milestone is not a zero-sum game—developers can always create more tasks to improve the software beyond the documented requirements of the milestone. Third, we have yet to see a milestone in which every requirement was perfectly fulfilled, such that there was no work left to be done for developers who still needed points. We now believe that overachiever compensation is not necessary, and have therefore removed it from our approach.

6.3. Student Feedback

Student feedback on the projects themselves have been generally positive. In the first year, the students felt that the mathematics needed to simulate the physics of billiards was too difficult. We believe that this assessment is correct, and highlights the

difficulty of selecting good software projects. In both years, students said that the resulting systems exceeded their initial expectations of what they felt they could accomplish.

The students who served as managers had generally positive experiences. 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 was able to directly apply lessons she learned from the course to her job. Similar feedback from undergraduates who have graduated and taken software development jobs indicates that the course does highlight realistic difficulties. Perhaps most tellingly, students often suggest that we should make the projects and groups smaller to ease the difficulties of communication and teamwork. While this misses the point, it does indicate that the approach we have taken is succeeding in creating a realistic software development situation in a classroom setting.

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 top four challenges the undergraduates reported were communication, scheduling, differing visions for the project, and differing levels of commitment to the project. One student said that the professor's warning about communication problems was the "understatement of the year."

Our use of weekly meeting times in the second year addressed the scheduling problems students encountered in the first year. We believe that the "gaming" of the grading system that the students discovered in the second year can be addressed by removing overachiever compensation. One consequence of this is that developers will compete more for "good" tasks. A few students have complained that the system creates stress, since they must compete with other students for tasks. From our point of view, competition for tasks is good in that it prevents procrastination. Furthermore, we suspect that the definition "good task" varies from student to student; we have not yet needed to arbitrate ownership of a task.

7. 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.

One important component of software engineering that was absent from this course is maintenance. We could modify the course so that the first part is focused on maintaining the system from the previous year. 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. It is also difficult to incorporate two software systems into the limited time provided by a semester schedule.

In terms of the unproductive types of student behaviors, we believe that our revised grading technique is very effective at preventing hitchhiking. Freeloaders quickly gain a reputation and are not able to leech off of the hard work of other students for very long. Overachievers are free to work as hard as they want. We believe that removing overachiever compensation will significantly address procrastination. We can also increase the point value of tasks due at minideadlines to further encourage students to work earlier.

An overachiever can sometimes resent other students whose contributions (or lack thereof) negatively affect the overachiever's grade. This is especially true if the other students procrastinate or do not seem to be doing their share of the work. We believe that removing overachiever compensation will help address this issue, as it will encourage students to contribute, thereby removing the "tragedy of the commons." Another option is to remove the group grade entirely. However, we believe that software development is a team effort, and that this should be reflected in the grading of individuals.

Students are free to underachieve if they wish, but they will receive the grade that they earn. The unavoidable burden of the added work is still shouldered by the other developers. Dilettante behavior is prevented by limiting the number of tasks that a person can sign up for. Our system helps to prevent obvious tutoring from occurring by splitting the earned points between the tutor and the weaker student. (An additional approach might be to force students to choose different partners for different tasks.) Unfortunately, there is nothing we can practically do to prevent a tutor who has not signed up for a task from helping a weaker student to perform the work.

The approach we have outlined is particularly free-form and dynamic. This is by design, as we are working on the hypothesis that by reducing the overhead of the development process to the essential elements we have described, we can teach software engineering concepts in a team setting without unnecessary pedagogical effort. We have not yet evaluated this hypothesis. Indeed, elements of the approach such as the task management and grading system could be usefully combined with more rigorous team-based development processes such as introductory team software process (TSPi) (Humphrey, 2000). This is another area of future work that remains to be explored. Finally, an objective assessment of the effectiveness of this approach compared to more traditional approaches such as the small teams approach remains to be done.

ACKNOWLEDGEMENTS

The author would like to thank the Spring 2004 and 2005 CSci 435 classes at The College of William and Mary for their participation. The author would like to thank Jennifer Haddox-Schatz for her input on a previous version of this paper. Finally, the author thanks the anonymous reviewers for their helpful feedback.

REFERENCES

Beck, K. (1999). Embracing change with Extreme Programming. *IEEE Computer*, *32*(10), 70–77.

Hayes, J.H., Lethbridge, T.C., & Port, D. (2003). Evaluating individual contribution toward group software engineering projects. In *Proceedings of the 25nd international conference on software engineering* (pp. 622–627). Portland, OR: IEEE.

Humphrey, W. (2000). *Introduction to the team software process.* Reading, MA: Addison Wesley.

Meyer, B. (2001). *Software engineering in the academy. IEEE Computer, 34*(5), 28–35.

Parnas, D.L. (1997). Software engineering: An unconsummated marriage. *SIGSOFT Software Engineering Notes, 22*(6), 40–50.

Parnas, D.L. (1999). Software engineering programmes are not computer science programmes. *Annals of Software Engineering*, *6*, 19–37.

Parnas, D.L., & Clements, P.C. (1986). A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, *12*(2), 251–257.

Schultz, T.W. (1999). Students assessing teams. In *Proceedings of the 29th annual frontiers in education conference.* (Vol. 3, p. 13B2). San Juan, Puerto Rico: IEEE.

Shaw, M. (1992). We can teach software better. *Computing Research News*, *4*(4), 2–4, 12.

Shaw, M. (2000). Software engineering education: A roadmap. In *Proceedings of the 22nd international conference on software engineering—The future of software engineering* (pp. 371–380). Limerick, Ireland: IEEE.

Shaw, M., & Tomayko, J. (1991). *Models for undergraduate project courses in software engineering* (Tech. Rep. No. CMU/SEI-91-TR-010). Pittsburgh, PA: Software Engineering Institute.

Taylor, S., Robertson, E., Hogan, J., & McDermott, J. (2005). Issue Tracker (version 4.0.4) [Computer Software] Red Hat. Retrieved January 1, 2005 from http://issue-tracker.sourceforge.net/