Scaling up: How Thirty-two Students Collaborated and Succeeded in Developing a Prototype Software Design Environment

Emily Oh Navarro and André van der Hoek
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425 USA
emilyo@ics.uci.edu, andre@ics.uci.edu

Abstract

Virtually all software engineering courses employ class projects in which students practice their newly-learned skills. By necessity, these projects tend to be of a small scale. In efforts to better educate students in the many aspects and pitfalls of the software process, different alternatives have been tried over time. In this paper, we describe one such experience in which we put all thirty-two students in the course on a single, large project and gave them the open-ended task of building a prototype of "a better software design environment." This lead to a completely new set of dynamics and interesting opportunities to teach topics that normally would not be covered or illustrated by students' experiences in a regular software project. We introduce our course design, present its progression over the quarter, illustrate its strengths and weaknesses, and discuss critical factors for its repeatability.

1. Introduction

ICS 127 is the advanced project course in software engineering at UC Irvine. Students who choose as their specialty the area of software engineering generally take this course sometime during their senior year, after having taken the preceding software engineering project course, ICS 125. In the past, ICS 127 has been basically a repeat of ICS 125 – small- to medium-sized software engineering projects are obtained from various organizations within UC Irvine (e.g., research groups, bookstore, library) or from nearby industrial organizations, and the students work on each project in groups of about four to five. In so doing, they must produce a requirements specification, design document, implementation, and testing report by each of the deadlines designated by the instructor. This approach to teaching a software engineering project course is typical not only at our university, but at most others as well.

During spring quarter of 2004, we decided to try a new approach to teaching ICS 127 that was designed to give the students a different kind of experience – one that would bring up many real-world issues that are not usually introduced into the typical project course. Specifically, we put all thirty-two students enrolled in the class on a single project, and gave them the completely open-ended task of building a prototype of "a better software design environment." By the end of the quarter, the students had succeeded in implementing a software design environment with seven non-trivial features. In the process of working in such a large team, they also learned several significant lessons through experience that none of their previous classes had illustrated to them before, such as the importance of team work and communication, the difficulty of working in a large group, and the importance of collaboration tools. The remainder of this paper describes our experiences using this approach in this particular class, and is organized as follows: Section 2 describes the design of the course, including our goals in teaching the course and how it was run. Section 3 provides a week-by-week progression of the course. In Section 4 we discuss the benefits and drawbacks of our approach. We provide a brief overview of related work in Section 5, and we conclude in Section 6 with a few suggestions for future incarnations of the class.

2. Course Design

The undergraduate computer science program at UCI contains four software engineering courses: ICS52, Introduction to Software Engineering, ICS 121, Software Tools and Methods, ICS 125, Project in Software System Design, and ICS 127, Advanced Project in Software Engineering. This paper is about ICS 127, which students usually take in their senior year.

We set out to teach our incarnation of ICS 127 in a different way than had been done before: we wanted a richer and more dynamic experience to better reflect the kinds of situations in real-world software engineering projects. The students had already had at least three courses in which they worked on software engineering projects using a prescribed process, with a (real or fake) customer, both individually and in small groups. Because the students in ICS 127 were the most advanced software engineering students in the program at this point, we felt that it was appropriate to give them some extra freedom and greater control over their process and project plan than they had previously been given. We felt they were now at a level at which they could appreciate and learn some more advanced issues such as how to manage a development process, the dynamics of working in a large team, and the importance of tools to coordinate collaboration in such an environment. We aimed to build on their previous experiences by taking a much different approach, the goals of which are as follows:

1. Give the students the experience of working on a large-scale project, in terms of number of people and size of project.
2. Give them the ability to accomplish something much larger than an individual or small group could, illustrating to them what kind of complex program is possible when a large team is involved.
3. Give them the opportunity to do something "new" rather than the usual requirements, design, implementation, and testing.
4. Give them a flavor of the real world.
5. Give them the experience of greater responsibility and true "ownership" of the project.
6. Hand them an experience they can take with them to interviews and use to land jobs.

We introduced the assignment to the class in a very non-informative way, by design. On the first day of class, we introduced how the class would be run – no formal lectures, no exams, and no deadlines or deliverables required by the instructor. We simply told the students that their task was to build "a better design environment," and suggested a few novel features that might be included, but made it clear that all other aspects of the product and the process would be determined by them, encouraging them to think of the teaching staff as merely "consultants." We pointed out that although they would be working in sub-groups to tackle certain aspects of the project, these sub-groups would appear and disappear as appropriate for each phase of the project. Moreover, we would also make team membership rotate throughout the quarter so they were intrinsically one large team working on the same project. We also made it clear to them *why* we were approaching this class in a different way, by communicating to them the goals of our approach, mentioned previously. After the first lecture, we sent them "away" with an assignment to write a one to two page "plan of attack" proposal – what they thought their first steps should be in undertaking the project.

A difficult task in such an unstructured setting was how we were going to evaluate the students. With no deliverables or exams, we opted to assign grades based on the following criteria: Approximately 60-70% was based on peer evaluations within the subgroups, which the students were required to turn in every week. The rest of the grade was based on the final demo of the software (if they completed one of their subgroup's assigned features, they received the maximum points for this criteria), our observations of class performance, and personal logs of each student's activities, which they were also required to turn in every week.

The class met twice a week, for one hour and twenty minutes each session. Although the activities performed during this class time varied from week to week, it generally consisted of presentations by the students, detailing what they had done and what they planned to do next, as well as class discussions involving our reflection, feedback, and suggestions.

3. Week by Week Progression

The class was taught during a ten week quarter. Here, we summarize the activities week-by-week, including some of the issues that were discussed and decisions that were made.

Week 1: In the first week we communicated the goals of the class, presented the assignment, and had each of the students turn in a proposal about how they believed the project should be attacked. We then conducted an open discussion of the "plan of attack" in which the students proposed their ideas, steering this discussion towards: (1) which tasks needed to be done immediately, and (2) which types of teams should be created for handling each task. The following four teams were created as a result:

1. *Tool team*: The tool team consisted of four people. Their purpose was to research, evaluate, and choose appropriate tools, such as configuration management (CM) systems, development environments, mailing lists, message boards, programming languages, and so on. In addition, they had to set them up, along with tutorials for the class.
2. *Management team*: The management team consisted of five people, and had to design and enforce the development process, set deadlines, and define any standards for the team to use. The management team's first task was to create an initial process plan for the project.
3. *Commercial research team*: 15 students comprised this team, and their purpose was to research commercial design environments and figure out which of their features were good, bad, could be improved on, were missing, or could be done in a better way. In addition, they were to brainstorm about any new features that they felt would be useful. As a result, they were to create a list of suggested features to present by the end of week 2.
4. *Academic research team*: The academic research team consisted of eight people. They were responsible for surveying the research literature and discovering the types of design environment features that had been proposed. They too were tasked with developing a list of suggested features. They were also to have their feature list ready to present to the class by the end of the following week.

Week 2: In week two, the management team presented their process plan to the class. This plan consisted of a strict waterfall process and laid out strict deadlines for the entire project. The rest of the class was quite dismayed at this plan, as the deadlines were rather unrealistic – it called for a formal requirements document to be due the following week. This led into a discussion of why it is not always appropriate to use the waterfall model or to require a formal set of documents, and what some other alternatives are. From this, the class modified the plan: since this was such an open-ended project and did not have an explicit customer but instead was a "new product based on market research", they decided to use a spiral model, where they evaluated every week what they had done and what the next steps should be. In addition, a formal requirements document would not be created; instead, a list of features would be created jointly by the two research teams.

Week 3: Both the commercial research team and the academic research teams presented their lists of features. The class then attempted to pare down the combined list to a feasible size, but it proved to be too difficult with so many people – the discussion went around in circles and agreements could not be reached. It was decided that two representatives from each research team, along with two members of management, would meet to finalize the list.

At this point, a defining event occurred. One student proposed extending an existing design environment, Argo UML, rather than developing one from scratch. We made it clear that they had to carefully weigh the tradeoff between the benefits of already having an existing design

environment infrastructure from which to build, with the difficulty and effort involved in reading through and understanding the source code of a large system to be able to modify it. The class decided to extend Argo UML. This decision helped mediate the argument over the feature list by automatically taking off of the list a number of features already in Argo UML.

At the end of week 3, the class rotated teams. The class again evaluated what the next steps should be, and what teams they needed for these steps. As a result, the new teams were:

1. *Management team*: This new management team consisted of four people. The team primarily had the same responsibilities as the previous management team. However, each of them was also assigned to at least one feature team, and attended all of their meetings.
2. *Feature teams*: This team had 14 people as a whole, but was later (after the feature list was finalized) broken down into six sub-teams, one sub-team for each feature group. It was their job to do the design and then implementation of the features.
3. *QA team*: The QA team had six members, one for each feature group. Their responsibility was to create test cases for their team's features, help with design and implementation, and then test the features. It was planned that they would reconvene as a team towards the end of the project and test the integration of each feature into the final product.
4. *Argo UML team*: This team consisted of eight people whose job it was to gain an understanding of the Argo UML code. Each of these eight people were also assigned as a "liason" to one of the feature teams, so they could point then in the right direction as to what part of the code they needed to modify. This team would later become responsible for integrating all of the different features towards the end of the project.
5. *Tool team*: Because the responsibilities of the tool team had decreased by this time, the team downsized to two people, and these two people were also members of a feature team with a relatively simpler set of features.

The first thing the new management team did was finalize the feature list once and for all. They divided it into three phases, phase one consisting of the most important features, phase two containing the less critical features, and phase three consisting of the features that would only be implemented if there was extra time. The management team decided that UML designs of all teams' phase one and two features should be turned in by week 5. Also, the new management team now took over scheduling the activities for each lecture period.

Week 4: In class discussion, it was brought up that as the feature teams began to design, many of them realized that they did not understand what the features were that they were supposed to be designing. This was due to the fact that the feature list only included brief narratives of each feature, and most times the people designing and implementing the feature were not the same people who had researched that particular feature before. As a result, management had the previous members of the research teams who proposed each feature produce a more detailed description of that feature. To account for this delay, the deadline for submitting designs was pushed to week 6. The Argo UML team continued to examine the code.

Week 5: In class, each team presented whatever part of their preliminary designs for their features that they had completed at the time. The Argo UML team continued to learn the code, but also began helping their assigned feature teams with their designs.

Week 6: Although many teams were done with their designs, very few of the feature teams actually turned in their design documents to the CVS server, as requested by management. One or two of the teams had begun coding at this point, but most were still having trouble figuring out what part of the Argo UML code they needed to modify, and had created their designs without Argo UML in mind at all. This chaos was slowly put under control with many meetings, discussions, and advice by the Argo UML team.

Also during this week we had a "look back" lecture period in which we asked the students to each write on a sheet of paper two things that they though were going well in the class, two things that were not going well, two positive things about the class, and two negative things about the class. We then collected them and discussed the most frequently mentioned issues

in class. Students were surprised at how hard it was to learn the Argo UML code. Many were confused at the authority structure in the class (was the professor or the management team in charge?). Moreover, they expressed resentment at management for their "heavy-handedness", but at the same time complained that there was too much uncertainty about what they were supposed to get accomplished when. Several students felt that working in such a large group had both positive and negative aspects: On one hand, it was hard for so many people to come to a consensus, make decisions, communicate, set goals, and get tasks accomplished. On the other hand, they liked the real-world flavor such an environment provided.

Week 7: Most of the teams began coding their first feature during week seven. Class time was spent with each feature team presenting their work done to date. In addition, the question of how and where each team should check in their source code was raised. This lead to a discussion, lead by the instructor, about the concepts of branching and merging. Up until now, the students had never used these techniques, and many didn't even know that they existed.

Also during week seven, one of the feature teams declared that they were at a standstill – one "designer" wanted to implement one of their features first, while the other one wanted to implement a different one first, and the QA person was stuck in the middle. After much discussion involving the feature team and the management team, everyone decided it would be best for the project if the team split into two, one "designer" for each feature, and the QA person and Argo UML liason would work with both of them.

Week 8: The last few teams that were still stuck on the Argo UML code started coding their features. There was no class this week, due to travel by the teaching staff.

Week 9: All of the teams continued coding. Management began to get complaints of certain students "slacking" while others did all of the work. There was a high level of stress evident in all of the teams at this point, especially in the integration (former Argo UML) team, who noticed that none of the teams expected to finish coding before week 10. This lead to a useful class discussion about prioritization and integration approaches.

Week 10: Many groups didn't turn in their code until the day before the class was to give their final demos. Hence, the integration team spent the entire night before that class period integrating the code. During this night, an interesting discovery was made: in spite of the class's use of CVS, there were two different versions of Argo UML floating around – some teams had extended one version, and some the other version. In then end, however, all seven features were successfully demonstrated, and five of seven were integrated into one product.

4. Critical Perspective

As a new course with a very different focus and set up, it should come as no surprise that we learned a number of interesting things, some of them strengths, some of them weaknesses, and some of them simply surprises. We discuss some of the more critical such items below.

4.1 Strengths

Putting on a course like this incurs a risk: what if the students do not step up to the plate? As is most often the case with unusual class set ups, however, the students in our case did rise to the occasion and worked extremely hard to make the experience as successful as possible. In the process, our interactions with them were able to enlighten them about a number of "real-life" aspects of software engineering they were experiencing that go beyond the typical software engineering course:

- **New product development versus traditional requirements analysis.** We found that presenting the project as a new product gave the students valuable experience in doing "market research", made them more independent in their approach, and promoted a real sense of ownership of the project. At the same time, it challenged their notions of the

software process (illustrated by the initial strict waterfall proposal and the group's unhappiness with that, which we could channel into a discussion that used the reasons why they felt unhappy to lead us to a more risk-oriented approach) and made the students unsure about exactly what was expected. We channeled this last fact into a long discussion on ownership of products and processes, and the necessity to step up to the plate and independently think, research, and design solutions.

- **Interpersonal communication and teamwork.** In reading the mid-quarter and final class evaluations, personal logs, and peer evaluations, as well as in conversations with the students, many students often commented about how they truly saw for the first time the difficulties of working in a software development team, and the importance of good communication and teamwork. Previous projects always involved 4 or 5 team members, in which 1 would take charge (a natural pattern). Our course challenged this notion with a project so large, no single person could "take over", multiple sub-teams were required, and inter-team communication was a must to the success of the project.
- **Natural selection in the allocation of tasks.** Students expected to be assigned tasks, but we required them to instead volunteer themselves for tasks. Though surprising to the students at first, group formation was actually easy and there was a natural division of who did what that illustrated people's strengths. Some students excelled in their job as quality assurance team members, others naturally fit the management team, and yet others loved to be on the integration team because it allowed them to have their hands in lots of tasks at the same time. This illustrated to the students the power inherent in a well-designed team, and also made them appreciate the differences in strengths among themselves. While the stronger teams took the tougher features and the weaker teams the simpler features, most of the students in the end felt they contributed significantly.

All of this led to a large amount of work that was accomplished. The students managed to get about half of the desired features implemented (yet another lesson: not everything can be completed at the same time!), and all but two integrated in a finished demo product (one of the two was so complicated and far-reaching, it was demoed on its own – yet another lesson for the students that they took to heart). The students initially were quite skeptical about the endeavor and were eager to start coding early, but the focus on architecture and design really helped in easing the load later on and demonstrated the value of these steps when a large group of developers is involved.

4.2 Weaknesses

We also had some difficulties in the course. The main issue was a lack of understanding about who was in charge: the management team or the instructor. For quite some time, there was a constant tension. The members of the management team assigned deadlines and deliverables, but students would look at the instructors to ask whether this was "truly happening" (influenced perhaps, by the fact that the instructors set the course grades). At the beginning, we tried to step in only when it seemed the students were going off in a dangerous direction or when they explicitly asked us for advice. Later, we changed our strategy to explicitly turn management directives into deliverables. This helped in reinforcing the authority of the management team and getting intermediate results on time; but did let go a little bit of the objective of "virtually total autonomy of the students".

We also felt that management had deeper insights into the performance of everyone than we did and that some inter-personal problems or other issues did not surface to our level. This was positive, in the sense that the students really tried to resolve issues themselves, but also detrimental in a few cases (such as the break-up of one of the teams; early intervention likely would have been able to deal with the issue much better). From an educational perspective, it was also problematic because grading became more difficult than anticipated. Although we

had peer evaluations, personal logs, class participation, and personal impressions to go by (and successfully intervened a few times when it was clear a student was not participating), in the end we felt we knew that some of the performances in the class were not accurately reflected in this data, and – almost inevitably with these kinds of courses – erred on the high side with some of the grades.

Finally, we note that the natural selection in the assignments of tasks also led to some problems: some students felt they put much more effort into "their" feature than others put into "theirs." This is a normal occurrence, and we discussed it as such; in no organization will you find everyone performing at the same, top level, and in most organizations – as here – you really have no basis on which to judge who is doing how much, only a perception. Despite our repeated attempts at explaining this, some students still felt "cheated" when the final grades came out and thought some others deserved lower grades.

4.3 Surprises

There were also some unanticipated events that happened in the course. One team split up; in the end, despite using a CM tool and a dedicated team, the integration team was faced with two different base versions of the code; some students really did not participate (and received a grade accordingly); and there generally was a lack of use of lessons from previous software engineering classes. Regarding the second and fourth point: it was a struggle for the students to recognize up-front the need for CM, bug tracking, and so on. We blame this on previous courses in which the topics are introduced, but their benefits are never experienced (or better said, the pain when the tools are not used is never felt). We often made recommendations, but students would not follow suit. Only when they experienced the pain of non-use (for instance, they attempted to hand-read 100,000 lines of Argo UML code, and sent zip files back and forth with modified versions of the code) would class discussions reveal serious problems with their approach and would they start to recognize and use appropriate tools.

In general, however, we were positively surprised by most aspects of the course. Students were very capable, operated largely autonomously, unearthed many issues on their own, and brought their own discussion topics to class when things would not work the way they felt it should work. They learned on the fly, distributed responsibilities appropriately (though struggling initially with having to let go of control since it was impossible to own the entire project; they had to rely on their teammates), and in the end had a unique experience they normally would not have had at the college level. Evaluations for the class were correspondingly high, and many students were thankful for the experience (best illustrated by the first class, in which the course project was announced: immediately 10 students signed up "because of the experience" and 7 dropped the course "because of the experience"). Students did not necessarily think it was easy, but many have now reported back that they have been able to put many lessons into use in their jobs and, in fact, some students found jobs because of the class.

5. Related Work

CSEE&T has seen several different trials of innovative approaches to group software engineering projects. These have included a large-scale, ongoing project that groups of students work on from semester to semester [3, 4], requiring students to work on a real-world project sponsored by an outside, industrial organization [1, 2], basing the project on the latest development approach (e.g., extreme programming [5]), and others. While all of these approaches succeed in making the project more realistic, our approach takes a different slant by significantly scaling up the project and explicitly putting the responsibility in the hands of thirty-two students. As a result, our approach exhibits a different set of lessons and phenomena. We suggest that our approach be placed later in the cur-

riculum and follow previous software engineering courses (that may or may not include the above approaches) to give students some base experience. Only at that point will they be advanced enough to handle the responsibility given to them in our approach.

6. Conclusions

We have presented a novel course for advanced software engineering students that puts the students squarely in charge, lets them experience a project so large they have to collaborate in order to complete the assignment, and that challenges them in ways that are representative of their future careers. The course proved it is possible to scale up software engineering education, with the caveat that students must have a serious software engineering background to begin with – gained through either industrial positions or one or more software engineering courses. The response to the course was overwhelmingly positive, with students appreciating having the experience, learning how to collaborate at a larger scale, and being able to develop a serious product.

We realize that this experience was, in many ways, unique to the group of students, the instructors, the project, the teaching style, and other environmental factors. It will undoubtedly require attempted replication of this approach by instructors in other institutions and environments to determine whether or not it is truly repeatable and can provide similar benefits in others situations. However, we believe our experience is best viewed as a "demo" project that shows that such an approach is possible and has the potential to, at the very least, evoke very positive reactions from students in terms of the lessons learned.

We plan to offer this course again, but will make several small adjustments to resolve the issues identified previously. Specifically, we will require meeting notes (to get a better feeling for what is happening "behind the scenes"), more strongly back up the management team (to resolve the issue of authority and in the process force a more equal distribution of work), and be insistent on the use of certain tools (to smoothen the software process itself). However, we realize that these measures may have side effects. For instance, requiring meeting notes may lead to less intense meetings amongst the students, preventing them from experiencing the full set of issues associated with close collaboration. Similarly for the insistence on tools: they may just have such a smooth project as a result that they do not really appreciate the role of the tools in the first place. In any case, we will closely monitor these issues to observe and report any notable differences.

While this approach has the added benefit of requiring less time from the instructor than in other types of courses, it also requires a significant amount of thought and mental energy to achieve the right balance between providing students with sufficient autonomy while offering the appropriate amount of guidance. Probably the most important piece of advice for those attempting a similar approach would be to be prepared to trust the students, perhaps more than is comfortable, but carefully spend the lecture time steering the project as needed.

References

[1] J. H. Hayes, "Energizing Software Engineering Education through Real-World Projects as Experimental Studies," in *CSEE&T 2002*: IEEE, 2002, pp. 192-206.

[2] A. J. Kornecki, S. Khajenoori, D. Gluch, and N. Kameli, "On a Partnership Between Software Industry and Academia," in *CSEE&T 2003*. Madrid, Spain, 2003, pp. 60-69.

[3] J. C. J. McKim and H. J. C. Ellis, "Using a Multiple Term Project to Teach Object Oriented Programming and Design," in *CSEE&T 2004*. Norfolk, VA, 2004, pp. 59-64.

[4] N. Wilde, L. J. White, L. B. Kerr, D. D. Ewing, and E. A. Krueger, "Some Experiences With Evolution and Process-Focused Projects," in *CSEE&T 2003*. Madrid, Spain: IEEE, 2003, pp. 242-250.

[5] L. Williams, "Integrating Pair Programming into a Software Development Process," in *CSEE&T 2001*, D. Ramsey, P. Bourque, and R. Dupuis, Eds. Charlotte, NC: IEEE Computer Society, 2001, pp. 27-36.