Teaching Software Engineering to a Mixed Audience

Michael Godfrey
Cornell University

January 8, 1999

Abstract

This paper describes some observations derived from teaching a course in software engineering to a mixed audience of undergraduates and professional Master's degree students at Cornell University. We describe our initial philosophical goals in teaching the course, some of the problems we encountered, some of the unexpected results, and what we intend to do differently next time.

1 Introduction

The software engineering course at Cornell, CS501, is intended to be taken by advanced undergraduates and professional Master's degree students (*i.e.,* M.Eng. students). While this course is not required by any CS program, it is commonly taken both by undergraduates who wish to improve their understanding of the area, as well as by newly-arrived M.Eng. students who wish to fill out their knowledge of software development. Additionally, we have found that undergraduate, M.Eng., and Ph.D. students from other disciplines (such as electrical engineering and physics) take CS501 as a way of becoming better acquainted with computer science in general.

1.1 Course Background

In previous years, the course (and its accompanying projects) has emphasized practical topics such as the effective use of various programming tools and techniques. For example, several lectures were spent on object-oriented programming and efficient use of the C++ programming language, students were taught about graphical programming and scripting languages using the TCL/TK toolkit, and students were taught about the practicality of "smaller" professional programming tools such as `purify` and `quantify`. This wide-ranging theme was also supported by the structure of the course projects: instead of an incremental term-long project done within a single group, the students attacked a series of shorter, independent projects each done in a different group and concentrating on different aspects of software development. The students were then allowed to express their creativity on a relatively unconstrained term project that spanned the second half of the course.

This wide-ranging and applied approach was popular with the students, who felt they came away from the course with an extremely useful kit of knowledge and experience. They had learned to use some real languages and tools, and had learned how to use them effectively and efficiently. Nevertheless, while this course structure was undeniably useful and popular, we decided to move away from its emphasis on languages and tools. To this end, our main goal so far has been to re-orient the lecture material toward a broader understanding of software development, its inherent and accidental difficulties, and how and why software engineering *as practised* often differs sharply from both software engineering theory and the practise of traditional engineering disciplines.

We have maintained a significant level of technical detail within the course; we have moved the teaching of much of the detail into recitation sections conducted by TAs and into course handouts, and the assignments continue to emphasize various practical problems of software development. Additionally, some of these topics still form an essential part of the course curriculum. For example, part of the mandate of CS501 is to teach object-oriented programming in C++; while most recent undergraduates already have reasonable facility with object-oriented programming, many M.Eng. students coming in from industry do not. Our compromise has been to teach object-oriented programming at a fairly advanced level while at the same time incorporating discussion of higher-level issues such as design patterns and the nature of inheritance.

1.2 Background of the Students

A significant concern in developing the material for this course was that the background of the students varied greatly. Many of the students were Cornell undergraduates who were, in general, very bright but often relatively inexperienced and immature as programmers. The M.Eng. students, on the other hand, were more varied. Some had significant industrial experience, while others had come directly from undergraduate studies at Cornell or another university. We were surprised to discover that, for example,

- students who considered themselves to be "experienced" programmers often had only a vague idea of what an interface is or why it's important;
- students who had come directly from undergraduate studies had rarely worked in a group of more than two people; and
- students were generally unaware of the relative trade-offs of various languages and tools that they claimed to be familiar with.

For these reasons, the initial lectures and assignments concentrated on giving an overview of topics that we considered fundamental, such as modularity, object-oriented programming, working in groups, and code inspections. We tried to "level the playing field" as much as possible early on so that we could give the students more freedom in the course project in the second half of the term.

We found that the students' expectations of the course also varied greatly. M.Eng. students who had worked in industry tended to be more mature and patient than the other students. They were more likely to be interested in higher-level issues, to seek the "why" behind the "what", and to be able contribute real insight into class discussions. Undergraduates and M.Eng. students who had come directly from undergraduate programs were often much narrower in their experience and also less patient. They tended to be bright and eager, and many of them had software development experience as summer interns and co-ops at companies such as Microsoft and Intel. However, we found that they were often much less interested in philosophical issues; they wanted to know about "hot" techniques and tools that would be of immediate use to them, such as effective programming using Java or Microsoft's COM platform. The real challenge of teaching this course was to engage both groups of students.

2 The Course Assignments and Project

2.1 The Assignments

The students were required to complete several medium-sized assignments followed by one large project. The assignments were tightly structured and concentrated on particular software development topics, such as code walkthroughs, object-oriented programming and design patterns, re-engineering old code, and user interface design. The final project was more free form: students has to conceive, design, and implement an original software system with a graphical user interface. In our first offering of the course, the students were asked to create an original game; in our second offering, they were asked to create a software engineering tool of their own design.

Although it is common for software engineering courses to use a single, term-long project, we feel that this combination of medium-sized, structured assignments followed by a more open-ended final project worked well for us. The smaller assignments helped to ensure that core skills were developed early on; since the backgrounds of the students was varied, we wanted to "level the playing field" by the time the final project came around. The open-endedness of the final project allowed students to express the creativity that the earlier assignments had stifled somewhat. We found that many of the students were quite enthusiastic about the final project. In particular, three of the software engineering tool projects were exceptional and may yet lead to either research papers or commercial development.

Of course, the disadvantage of using several smaller assignments is that the final project is necessarily of a lesser scale than in a course that employs a single term-long project. However, we feel that within a single software engineering course, it is vital to ensure competence in basic software development skills. Should we offer a sequel course to CS501, it is likely that we will use a single term-long project for that course.

2.2 The Course Project

In our two offerings of CS501, we have used substantially different themes for the final project. The first time we taught the course, the students were asked to implement an original game with a graphical interface. The second time we taught the course, the students were asked to implement a software engineering tool of their own design.

The use of a game as the course project (as had been done by other instructors in previous offerings of the course) was quite popular with the students, especially the undergraduates. We found that most of the games were quite sophisticated and showed mature understanding of topics such as object-oriented programming (simulations) and use of networks (multi-player games). We are satisfied that using a game as a software engineering course project is reasonable, as long as the initial proposals are carefully screened and the progress of the projects is tracked.

Despite this success, we decided to experiment with the second offering of the course and ask students to design and implement a software engineering tool as the course project. Several concrete suggestions were given, based on lecture material and previous assignments, but students were also encouraged to suggest an original tool. We were extremely pleased with the results; the overall calibre of projects was very high, and three projects were exceptional. In particular, some of the better students were very enthused and put a lot of effort into their projects. Several students asked to be allowed to extend their course projects into independent study projects, and one group is currently considering turning their tool into a commercial product.

We must confess that several students were unhappy with this change, as they had been excited about the possibility of creating a game. However, even more students commented that they had learned a lot about the nature of software development and tool design. In hindsight, it seems clear to us that that this change in project themes was for the better; not only did the students gain experience in designing and building a software system, but they also had to explore seriously an area of software engineering itself in designing their tool.

2.3 Working in Randomly-Chosen Groups

For our first offering of the course, the assignments and the course project were done in randomly-chosen groups of four that were changed with each new task. This gave students a wide variety of experiences in working within groups. We must confess that this approach was generally unpopular with the students. (One student complained that it was unfair because, "In industry, you'll always be able to choose who you want to work with.") However, we feel that this "variety of experience" is extremely important for future software developers. After the course was finished, one M.Eng. student commented that while he had not enjoyed working in random groups, he considered it to be "good medicine" and was grateful for the experience.

For the second offering of the course, we relaxed our approach somewhat: the initial assignments were still done in randomly-chosen groups, but the students were permitted to pick their own partners for the final project. Since the project is worth a significant portion of their final grade, we decided that it was reasonable to allow the students the freedom to choose who their partners would be. We feel this blended approach is a reasonable compromise and that it worked well. Again, several students commented after the class was over that there were glad for the experience of randomly-chosen groups, even if it was unpleasant at times.

3 Course Philosophy

In addition to covering the traditional areas of software engineering — requirements, design, testing, *etc.*— we have tried to provide a high-level and philosophical view of software development that is not commonly found in software engineering textbooks. We now elaborate on some of the themes we explored.

3.1 History is Important

One of the major themes of the course was the saying of Santayana, "Those who cannot remember the past are condemned to repeat it." Since ours is a fast moving field, we feel it is important to give students an understanding of how and why techniques, tools, and notations have evolved as they have. For example, we discussed the evolution of programming languages in terms of both programming abstractions and politics. Too often, we have found that students believe in a kind of social Darwinism with respect to software engineering (*e.g.,* if it's not written in C, it can't be real; interfaces were made to be broken). Not surprisingly, we found that the M.Eng. students who had worked full-time in industry were the most receptive to and interested in these topics; those without industrial experience were less patient and less willing to entertain the idea that history is important.

3.2 The Technological Peter Principle

The Technological Peter Principle (the phrase is a facetious creation of the author) states that any good technology is used up to the level of its natural incompetence. For example, no matter how fast and smart our computer networks become, they are never fast or smart enough because we also come up with new resource-intensive applications for them. A consequence of this is that since performance is always important, we are often tempted to ignore issues of reliability, security, portability, and sacrifice general good engineering principles in the name of efficiency.

A related lecture topic concerned the appropriate use of scripting or "little languages" such as `perl` and TCL/TK. We discussed a white paper by Dr. John Ousterhout on the use of scripting languages, strongly versus weakly type languages, and the nature of inheritance. This topic stirred particular interest within the class. While many students had used such languages before, we found that few had seriously considered the some of issues raised in the paper and lecture, such as the nature of software "glue" and the problem of maintaining non-trivial applications written in these languages. Once again, we found that many students seemed to believe simply that "newer is better" and "cool is king".

3.3 Professional Responsibility

We held informal in-class discussions of the responsibilities and professionalism of software engineers. For example, one of the materials used was the paper based on Prof. Nancy Leveson's keynote address to ICSE-94, "High-Pressure Steam Engines and Computer Software"; in this paper Prof. Leveson argues, by historical analogy, that we often have paid insufficient attention to the very real and vital problems of software reliability in safety-critical systems. The response from the class was surprisingly strong. Some students seemed offended by these discussions. One student said that if issues such educational standards of software engineers were truly important, they would be probably required by law. However, after the course was over several other students commented that they had found these lectures to be highlights of the course. They were surprised that no one had ever asked such questions in their undergraduate career.

3.4 Tightening the Reigns vs. Chomping at the Bit

We have found, not surprisingly, that most students who have come straight from undergraduate degrees did not like the constraints that disciplined software development usually entails. Some were even surprised to hear that many companies follow strict development processes. M.Eng. students who had some industrial experience were less shocked, but often did not understand just why structured activity is so important. We argued that structure in software development is useful for several reasons: it provides stability and traceability over time; it gives a handle on the scale and complexity of large, evolving systems; and it entails repeatable results, especially with regard to reliability. However, the message we presented was *not* that of "salvation through structure"; there is no magical crank which when turned will produce interesting, useful, and profitable software systems. Many students were intrigued by the idea of there being a trade-off between creativity and structure.

4 Next Time ...

We have now taught CS501 twice. Based on our experiences (and 20/20 hindsight) we plan some changes for the next time we give this course.

4.1 Show Them the Future

It has often been commented that in software engineering practise significantly lags theory. Of course, we agree that this is a fair comment, but we also seek to address this issue by presenting some of the larger and more interesting problems of software development, and how research systems are trying to address them. For example, the various problems of software configuration management (SCM) have been known for a long time. Although sophisticated commercial SCM tools exist, many practitioners use simple and inexpensive tools such as RCS or MS-SourceSafe, if they use any approach at all. We intend to describe the industrial state-of-the-art to students and also describe how research systems are attacking the "next generation" of problems.

4.2 Lightweight Formalism

While software engineering courses at some universities emphasize the formal development of software, we have not done so. We consider that the key aspect of building large software systems is reliable and enforceable *abstraction* rather than formality *per se*. However, logic and mathematics are, of course, key to good abstraction. If we wish our students to create software systems that are "essential" then we must get them to think in these abstract terms too. Consequently, a short section on formal specification will be introduced into the course.

4.3 Presentation Skills

During the final project demos, it became obvious to us that many of the students did not posses finely-tuned presentation skills. While this is perhaps not terribly surprising, next year we will require students to give more formal presentations of their work, including web page mock-ups and in-class talks.

5 Conclusions

In summary, we have several observations based on our experiences in teaching a software engineering course to a mixed audience of undergraduates and professional Master's degree students:

- Students who have never worked in industry full time need convincing that software development is worth doing carefully. While this remark might seem obvious, we feel it bears explicit mention.
- Students with real industrial experience are likely to appreciate high-level discussions of topics with which they already have some experience.
- Using a series of shorter constrained assignments followed by a more unconstrained class project is a good way to "level the playing field" when student experience and ability varies.
- Working in randomly-chosen groups is a valuable experience; however, we feel that it is best *not* to use this approach with the final project.
- Most students appreciate hearing "views from the mountain", even if they don't agree with them. Several students remarked that the found class discussions on philosophical issues challenging and engaging.