Preparing Students for Software Engineering

Steve M. Easterbrook

Software Research Lab.
NASA/WVU IV&V Facility
Fairmont, WV 26554

Theodoros N. Arvanitis

School of Cognitive & Computing Sciences
University of Sussex
Falmer, Brighton, UK BN1 9QH

Abstract

This position paper describes our work with a new course at Sussex University, designed to bridge the gap between computer science and software engineering. We argue that the way in which software engineering is introduced in most computer science degrees makes it hard for students to internalise the lessons of good engineering practice. In particular, programming is seen to be divorced from software engineering. We describe a new course taught to first year undergraduates, once they have learned to program. The course exposes students to the difficulties of large scale software development, including integrating and modifying other people's code. The course uses a trading game in which student buy and sell software modules, making their own evaluations of cost and quality. An important innovation is to force the students to be explicit about lessons learned, as an introduction to process improvement. Early results are promising: the first cohort of students are significantly more motivated in their subsequent software engineering course.

1 Introduction

There is an important gap in the training of computing professionals, between software engineering and the practical aspects of "computer science", particularly programming. Programming is normally taught right at the start of a computer science degree. A typical introductory programming course will introduce students to a particular programming language, showing them how to construct algorithms to solve simple problems, and how to convert those algorithms into programs, through proper use of the constructs available in the chosen programming language. At some subsequent point in their careers, these students will be taught "software engineering". The typical software engineering course covers a broad range of material including management and economic issues, as well as technical aspects, across the entire software lifecycle. In our experience, the result is that programming and software engineering come to be seen as separate activities, entirely divorced from one another. The ideas taught in software engineering do not connect in any meaningful way with the ideas taught in introductory programming. Students remain suspicious of software engineering principles, and happily abandon them when developing software [1].

The result is that for many software developers, there is a gap between what experience and good engineering practice suggests they should do, and what they actually do. This slows the uptake of good engineering practice, not because the developers are ignorant of the key lessons of software engineering, but because they do not apply those lessons to themselves. These people have read the software engineering books, and taken software engineering courses, but do not apply the principles of software engineering in their own work unless they are forced to. For example, it is common to find software developers taking shortcuts to meet a project milestone. These shortcuts nearly always store up problems for the future. However, the shortcuts are still taken, because deep down, the developers believe that the prescribed methods and processes are dispensable. We observe the same problem in student project work. Students are taught to document and comment their programs, but do not bother in practice, because they do not appreciate the value of good documentation.

We believe that the problem is due, in part, to the separation of the teaching of programming (and computer science) from the teaching of software engineering. This separation is endemic to the way we teach software engineering and computer science courses. For example, the practical work on a software engineering course is different from a programming course: on the software engineering course, the students might use some method to derive specifications for software they will never write; on a programming course they get on and write programs. The message is unfortunate: programming courses produce practical results, while software engineering just slows things down.

The problems are compounded by the constraints imposed by university teaching. The same constraints are present in industrial training courses. Firstly, both university and industrial courses are too small for students to get to grips with the problems of developing a large scale software systems. Hence the problems are broken down into self contained exercises. Secondly, there is a need to provide students with 'doable' exercises, for which there is such a thing as a 'good' answer, and for which it is possible for the students to do well, provided they put in the requisite amount of effort. Thirdly, assignment deadlines tend to emphasise the importance of keeping to schedule (i.e. handing in something that works) over the importance of maintaining quality. Students do not get the chance to internalise the rationale for good engineering practice. They do not get any first hand experience of the added value of the techniques they are taught.

2 The Software Design Course

At Sussex, we have been experimenting with a new course structure to bridge the gap between introductory programming and software engineering. We do this through an intermediate course, taken immediately after introductory programming, which exposes the students to the importance of disciplined programming. Instead of forcing students to learn and use a particular software engineering method, we expose them to the problems of large-scale software development and maintenance, while offering them a set of practical, low-level techniques that may help them to get through the practical work. At this stage we do not worry whether students use the techniques: we concentrate on making them think about the problems they meet in developing large programs, and on getting them to talk about their experiences. The intention is to provide the motivation for a subsequent software engineering course.

The course represents part of a strategy of "bottom up" introduction of software engineering topics. We build upon students' recent experiences with programming, by concentrating on the design and implementation processes in the software life cycle, together with basic material on testing and maintenance. Typically, these areas are not covered intensively on advanced Software Engineering courses. For example, our second-year Software Engineering course concentrates on requirements analysis and specification, testing and QA, and the management of software projects. The new course provides the intermediate step, exposing students to the problems of designing and implementing large programs, and encouraging the use of good programming practices. In this way we equip students for the the broader Software Engineering course.

2.1 Course Structure

This new intermediate course is called "software design". It is based on a practical project, running throughout the course, which the students tackle in small teams. The project is divided into three phases.

In the first phase, each team is given a specification of a software module to implement and test. Different teams are given different modules, but for each module there will be more than one team implementing it. The specifications they are given are reasonably detailed, but are incomplete, such that some necessary functions are left out, and the descriptions of the interfaces between modules are insufficiently detailed. Students are encouraged to talk to other teams to ensure their modules will be compatible. At the end of the first phase, teams trade modules with one another, in order to obtain a complete set. For the trading, they use a toy currency, and are free to negotiate prices, delivery dates, and contracts (including technical support) as they see fit.

In the second phase each team integrates the set of modules they have purchased. At the end of the second phase, the students once again trade, so that each team buys an integrated package which must not contain any of their original code. This rule acts as an anti-monopoly clause: if any team sells their module too widely at the end of the first phase, they limit their choices at the end of the second phase.

In the final phase, each team must modify the purchased integrated system according to a changed specification released at the beginning of the final phase. Also, during this phase each team is introduced to the idea of software evaluation by evaluating the purchased integrated system in comparison with their original one.

Throughout the course, the teams submit reports for grading. These reports consist in equal parts of product documentation and reports on their process. Particular emphasis is placed on describing their strategies (for marketing, testing, purchasing, etc.), and on reflecting on what went well and what went badly. Time is set aside throughout the course for students to make presentations to one another, for marketing purposes. Also, throughout the project, each team submits a weekly report (by filling in a paper or web-based form). These submissions are compulsory but not assessed. This allows the students to report their problems and encourages them to critique the running of the trading game, and the course in general. The course tutors use these forms to monitor the students' progress and to identify particular learning difficulties.

2.2 Why a trading game?

The use of a trading game to teach software engineering is not new. In fact, the basic structure is taken from a report of a programming course taught in the early 1970s [2]. The advantages of such an approach fitted our needs very well. It exposes students to:

- reading and modifying other people's code;
- the problems of integrating software written by different teams;
- experience of evaluating software prior to making a purchasing decision;
- the importance of setting "industry wide" standards to ensure compatibility;
- the role of documentation in maintaining software;
- the difficulties of working in a team.

However, we have added a few twists to the idea, which we feel have helped us to achieve our goals for the course:

- The students are doomed to failure. From the initial specification, which contains many inconsistencies and omissions, to the revised specification handed out two weeks before the final deadline, it is impossible for the teams to produce a complete solution. Many students are uncomfortable with this, especially those used to completing exercises and getting good grades. We encourage them to think carefully about prioritisation, and to plan the amount of effort they should put in.
- Oral presentation is given a high priority. Each team is given several sessions in which to make presentations to their customer base. A final, assessed, presentation requires each team to present what they have learned from the course. For many of the students, this course is their first experience of giving technical presentation, and we noticed a dramatic improvement in presentation skills as the course progressed.
- More emphasis is placed on reporting the process than the product. Each submitted report includes a process evaluation. Teams are awarded good grades if they identify weaknesses in their processes, and can suggest process improvements that address any difficulties they experienced. For their final oral presentation, they are asked to imagine that their audience consists of potential (software) employers, who want to know why the members of the team would make better employees than people who have not taken the course. These presentations act as a prelude to the final course debriefing plenary, in which the students compile a list of lessons learned.
- We deliberately fostered an atmosphere in which students could be freely critical about what they are being taught. They are encouraged to voice criticism of the course structure and content, and of all aspects of the practical work. At the very least, this allows them to let off steam when they get frustrated with the practical work. More importantly, it provides a route into discussion about process improvement. Whenever they criticize the course, we try to respond positively to their criticism, but also ask them to come up with coping strategies, so that if they can't change the nature of the course, they can at least cope better with the difficulties it presents.

2.3 Textbooks

During the development of the course, we searched for suitable textbooks. We wanted to avoid both the encyclopedic approaches of the main software engineering textbooks, and the restrictions of books that teach a particular method. What we were after was a set of practical techniques that students who had just learned to program could pick up and use. Furthermore, although we were concentrating on low-level techniques to apply to programming, we did not want to tie ourselves to any one programming language, nor did we want to teach the students a new language for the course. In fact, we had students taking the course who had learned to program in different languages, and who were using their different languages for this course. We did, of course, keep such students separated when it came to trading software!

In the end we gave up trying to find a suitable core text, and supported the practical work with a series of lectures, with associated notes, covering an eclectic set of techniques for the students to try out according to their needs. These included: use of dataflow diagrams and dependency graphs to represent designs; use of procedural and data abstraction to improve modularity; commenting code with pre- and post-conditions; proper use of exception handling; black and white box testing; code walkthroughs and checklists for verification; and debugging and regression testing. In addition we provided introductory lectures on topics such as measuring software quality, formal verification, methods, and process modeling and improvement.

3 Conclusions

The course is not without its problems. Many of the difficulties come from the tension between the goals of modern, pragmatic university students (who want to get good grades), and the pedagogical aims of the course (to experience the difficulties of large scale software development and maintenance). By and large, students expect assignments that can be completed in the time available, rather that the impossible deadlines and fluctuating requirements of a real software project. Despite our best efforts to prevent it, some students did succeed in completing very impressive systems, by putting in unreasonably long hours. These students seem to have gained less from the course than the others, and we hope to prevent this happening in future years.

Even with these problems, the course has been a great success in preparing students for software engineering. It has given them a wide range of experiences, from teamwork and technical presentation, through to an appreciation of how hard it is to modify other people's undocumented code. Most importantly, it has given them practice in the key software engineering skill of learning from failures. The majority of the students enjoyed the course, and rated it highly in terms of relevance to themselves. This first set of students are currently taking their second year software engineering course. They are proving to be far more motivated to learn about software engineering that any previous cohort of students.

At the current time, we have taught the course once, in Spring 1995, and are running it again in Spring 1996. It is too early to tell whether we have achieved our long term goal: we will not be able to determine whether the students who took the course really have taken the lessons to heart, until we observe whether they willingly adopt some of the techniques in their own software projects.

Acknowledgments

We would like to thank the students at Sussex University who took part in the course in 1995, and provided many helpful comments on the course. Also, thanks are due to Amer Al-Rawas and Joe Wood, who helped us to design and run the course.

References

[1] D. L. Parnas, "Education for Computing Professionals" *IEEE Computer*, Pp. 17-22, Jan 1990.

[2] J. J. Horning and D.B. Wortman, "Software Hut: A Computer Program Engineering Project in the Form of a Game", *IEEE Transactions on Software Engineering*, Vol. SE-3 No. 4, Pp. 325-330, July 1977.