Updating the software engineering curriculum at Edinburgh University

Perdita Stevens [*]

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Abstract. This paper reports the experience of Edinburgh University's Department of Computer Science in undertaking, in two stages, a major reform of its software engineering teaching in the third and fourth years of the degree. Our aim was to increase our students' exposure to object oriented and component based software engineering, aspects of modern software engineering which are highly relevant to the careers many of them undertake. One of our more significant decisions was to teach the emerging standard modelling language, UML. The paper describes the background to the decision to change the course, the changes that were made, and the results of the changes. It does not presume to draw universal lessons from the story, but does describe what we see as the significant features.

1 Introduction

In 1994 the University of Edinburgh introduced an honours degree in Software Engineering to complement its existing Computer Science and other degrees. At that time the Department of Computer Science took the view that software engineering was a branch of computer science, and accordingly the software engineering degree was a more constrained version of the computer science degree, omitting some of the more theoretical courses and requiring the more practical courses. It rapidly became clear, however, that there was scope for a broader and deeper coverage of software engineering as a discipline in its own right, and that there would be a great deal of student interest in such coverage.

The department decided to create a new final year module in Software Engineering in order to strengthen the software engineering content of the degree. However, resource problems meant that a new module was not designed in time for delivery in the academic year 95/96. This marked the beginning of the author's personal involvement in the development of this part of the syllabus. I had recently joined the department as a research fellow, after working for some years as a software engineer. Moreover I had been acting as tutor on a course, *Topics in Software Engineering* [3], developed by the (British distance learning)

[*] Perdita.Stevens@dcs.ed.ac.uk

Open University. This course seemed broadly suitable for our needs, and as the existence of a final year software engineering module had been announced to students, it was agreed that I would deliver a course based on it, adapting it as necessary. Because of the availability of good self-study materials (which we purchased from the OU), it was possible to depart from the more usual 18-lecture format of our courses, replacing some lectures with tutorial-style meetings in which students addressed problems in small groups. Lectures summarised the material, students read the detail outside contact time, and the tutorial-style meeting allowed them to check their understanding.

2 The first iteration: successes and problems

Topics in Software Engineering (TSE) was on the whole a success, sufficiently so that in 96/97 I changed the course incrementally and gave it again. There were three parts to the course: let us examine each in turn. (The original OU course also included a section on concurrency using CSP, which we omitted: concurrency was already well covered in our final year.)

1. Structured analysis and design using the Yourdan Structured Method. We abbreviated the introductory section, since our students had a greater background in software than the OU course assumed, and we emphasised the solving of design problems. We (students and staff) felt that the problem solving nature of the course was very valuable, and especially that getting students to address problems in supervised groups was useful. However, we were less happy about the particular paradigm and methodology involved (structured methods, specifically YSM): we wanted to consider teaching OOA/D instead. We had no tool support for YSM; one consequence of this seemed to be that the weaker students, who made a large number of essentially syntactic errors in their YSM diagrams, didn't get as far as considering the more interesting modelling issues.
2. Rigorous specification using VDM. This was the least successful part of the course. The Edinburgh degree course, reflecting the research interests of the majority of the staff, has a slightly higher than usual formal content, so this was not the first rigorous specification language students had been exposed to. While some students enjoyed the material, few saw it as very relevant to their needs. In the second year of presentation I deemphasised the actual writing of VDM specifications and emphasised instead the question of what techniques are appropriate under various circumstances; but this was still felt to be an unsatisfactory section.
3. Issues of large scale software development: project planning, management and quality assurance. This section was seen as particularly relevant to the students, especially those planning to enter a career in software engineering. We supplemented, and in the second year of presentation replaced, the OU material, and emphasised real world experience and the absence of silver bullets. Final year students are mature enough to appreciate discussion of

controversies in software engineering, and it proved possible, for example, to use different views about the rôle of metrics as a starting point for discussion. I introduced a section on reuse and components, which seemed an increasingly vital area, and we discussed the organisational influence on the success or otherwise of a reuse programme. Source material included recent research and practice; a lecture was given over to a software engineer to tell stories and answer questions about real projects, and students were also required to do their own research using the Web and libraries. We find that by the fourth year of a degree course, students have a good deal of relevant knowledge, but sometimes lack confidence articulating and applying it. A course in which students are encouraged to form their own opinions and back them up both from their own limited experience and from the writings of experts serves to reinforce their project and coursework in this respect.

In summary, it became clear that good points were

- the practical, problem-based approach of the course
- the focus on analysis and design: there was no need for another programming course
- the material in the final part of the course: it seemed appropriate to continue to evolve this, rather than to replace it.

However, we wanted to move to an object-oriented paradigm and to go into analysis and design in greater depth. We thought this study should ideally be supported by the use of a CASE tool, both as an experience in its own right and because the sanity-checking provided by tools could enable students to correct some of their own mistakes. To make space for the material it was acceptable to drop VDM.

At the same time, it was also becoming clear that we should take an overall look at our provision of mainstream software engineering courses and see how they fitted together and whether there was scope for improvement on a wider scale than individual courses. At that time, besides many other courses which had a software engineering component, the main provision was:

- The first[1] and second year courses lay the foundations. Students learn two contrasting programming languages, C in the first year and Standard ML in the second year. They are introduced to issues of programming in the large, including the ideas of modularity and encapsulation. They are shown the waterfall model.
  In 96/97 discussion was in progress about the renewal of our first year course; it was subsequently decided to use Java as students' first programming language. This influenced our subsequent discussion of the third and fourth year courses, though we did not feel that it was essential for us to adopt Java as a consequence.

[1] Scottish students typically start university at age 17

– The third year has a compulsory course in Programming Methodology. This popular course provides a thorough grounding in the software lifecycle. On the technical side it discusses possible structures for large systems using the ML module system as a source of examples and exercises. It considers reliability and safety of systems and discusses the influence of human factors. An optional third year course was Software System Design (SSD), which its lecturer Dr Rob Pooley was inclined to think could be improved. This course taught the concepts of object oriented design in the context of C++, and backed this up with semi-formal specification of C++ programs. The twin problems were that the object concepts were tending to get swamped by detailed concerns about the specification language, and that the tool associated with the specification language was unsatisfactory. The basic motivation of the course, to provide a good basis in object orientation, was fine.
  Third year students undertake two major projects, one of which is individual, and the other of which is done in groups.
– Fourth year students do an individual project, which normally involves some conceptual work and the development of a system.

3 The full reform

Dr. Rob Pooley (the lecturer of the third year course, SSD) and I agreed that it would be a good idea to replace both SSD and TSE with an integrated pair of new courses teaching object oriented and component based analysis, design and programming. These courses became known as Software Engineering with Objects and Components 1 and 2, with SEOC1 being a prerequisite for SEOC2. We produced syllabi for the courses, and the department agreed to proceed. This section discusses the decisions we made about the new courses.

We felt that it is important for students to have a thorough understanding of object concepts and detailed design before they can sensibly attempt analysis and high-level design issues. Therefore SEOC1's responsibility is to teach object concepts in the context of an object oriented programming language, and to cover detailed design without expecting students to be adept at making architectural decisions. SEOC2 deepens this understanding and considers high-level design and analysis issues, as well as addressing the process of software engineering within an organisation, including topics such as quality, quality assurance and management.

We wanted to use a class-based object oriented language, and because of our intention to be seen as immediately relevant to students intending to go on to industry, we wanted to use a well-known language. Our main options were C++ – which was obviously the favourite, since we already taught that language and since our students already know C – Smalltalk and Java. We settled on C++ for the first year of presentation, influenced partly by the unavailability of a CASE tool supporting Java on Solaris at that time. Although I have taught a successful object orientation course using Smalltalk, I felt that, given their backgrounds, our students were more likely to be happy with a typed language, and that expecting them to make two paradigm shifts in one course – to objects and away from static typing – on one course was probable unwise. In future we expect to use Java, but we wish to avoid tieing the courses inextricably to any one language. We aim to make clear to students which aspects of SEOC1 are particular to the programming language being used, and also that most aspects are not. SEOC2 has no explicit programming language dependency, though of course it draws on students' experiences with an OOPL.

The remaining questions were about methodology and modelling language. The latter question was rather easy: at around this time it was becoming clear that UML, the Unified Modelling Language, was going to become a standard. We felt that if we chose any other modelling language we would have to switch again within a few years, that UML was a very interesting development in any case, and that UML was the best bet for the employability of our students. Learning to use a modelling language competently, like learning a programming language, involves understanding its syntax, semantics and idioms. It does not automatically produce the ability to write good designs, but it facilitates the expression of designs. Therefore we decided that one aim of the two courses was that students should be competent in UML by the end of them. This decision made us, in 97/98, an "early adopter" of UML, which brought its own problems in the lack of a suitable textbook. There were good textbooks on object oriented design, and there were books at a professional level about UML; but there was no textbook which taught object oriented design with UML as its modelling language. Rob Pooley and I decided to address this problem by writing such a book ourselves; *Using UML: software engineering with objects and components*[1], is to be published by Addison Wesley in November 98. Another problem was that the UML documentation [2], which is under revision, is in places contradictory and confusing; however, we do not feel that the problems we faced were in practice any harder than they would have been with another modelling language; rather, the existence of a (partly) rigorous semantics document made it easier to expose inconsistencies that otherwise might have remained hidden in less formal documentation.

The question of a methodology to use was harder. We considered teaching the infant Objectory (the methodology from Rational designed for use with UML), Booch's OOD or Rumbaugh's OMT. However, no one of these was clearly the most suitable. More seriously, I harboured serious doubts about whether it was useful to label the software development process with a methodology name. It seemed, and seems, to me that methodologies borrow greatly from one another and that successful software development projects (including, but not limited to, those I had been personally involved in) normally adapt and borrow from different methodologies. I suspect that projects which "use the Booch method" for example, often mean little more than that they use the Booch notation; the rise of UML may make this clearer. In the end we decided to teach some common techniques for analysis and design, and to discuss some major methodologies briefly, but not to advocate any particular methodology or teach one in detail. This decision is still under review, but our initial impression is that it is correct. Techniques that we teach include among others use case analysis (with a discussion of its disadvantages) and the use of CRC cards for developing and validating a class model, and for studying interactions.

In the fourth year course in particular, I decided to make greater use of the Web and to increase the extent to which students are encouraged to do their own research. Given how fast the field of software engineering is moving, it seems important that students should leave university with not only an up to date body of knowledge, but also with the skills to keep up with the changing state of the software engineering world. It is valuable for students to see for themselves that opinions differed, and become confident evaluators of material. (For the same reason, I encouraged a guest speaker who I knew to have different opinions from my own on some fundamental issues to put his own views in his guest lecture.)

At the time, few CASE tools that supported UML were available, particularly on the Sun Sparc Solaris platform that we had available. Rational Rose, however, was available on this platform, and Rational proved able to provide an affordable educational licence price. We went ahead using Rose.

4 Experience of the full reform

In the academic year 97/98 both SEOC1 and SEOC2 have been taught for the first time. (Since this year's cohort of fourth year students have not had the opportunity to do SEOC1, of course SEOC2 was taught in a modified form. Almost all students taking SEOC2 had done SEOC1's precursor SSD, however, so the differences did not need to be very great: the main one was that it was not possible to rely on previous knowledge of UML, so a greater proportion of the course was given over to teaching the modelling language than will be the case in the long run.)

SEOC2 was taught in the autumn of 1997. It seems to have been a great success, both popular and academic: evidence for this includes:

- Nearly twice as many students took it as took the predecessor course, and almost every student who started the course took the exam.
- Very positive feedback, both to the lecturer and on course questionnaires; the course was seen as relevant and interesting.
- Several students have used techniques taught in the courses in their final year projects, for example producing UML designs.
- Several students have reported that people who interview them are interested in talking about the course, and that they have the impression it helped their job prospects.

My subjective impression was that the quality of work, both coursework and exam work, was higher this year than previously.

Some problems still need to be addressed:

- The CASE tool Rational Rose was found interesting, and I think it helped the weaker students to resolve syntactic problems, but given the Solaris resources

we have (Sparc 5s with 32Mb RAM) its performance was irritating. We are considering switching to NT.
- The only area of the syllabus that caused widespread difficulty was design patterns. I suspect that this was purely my fault: I was immersed in patterns in my own research, and probably failed to imagine myself into the position of someone who hadn't seen patterns before.

The CS3 course SEOC1, whilst popular, does not seem to have been such a spectacular success as SEOC2, probably because it is more similar in style to other courses. However, it is a clear improvement on the course that it replaces.

5 Conclusion

At the time of writing, these courses have been taught once each, and both appear to be successful.

We think the main area for further improvement in the coming year is the SEOC2 coverage of architecture, frameworks and patterns, which were squeezed this year by the need to teach UML from scratch. These areas seem increasingly important, and will reinforce the existing coverage of reuse and components.

We hope that readers who are involved in similar situations to our own may find our experiences interesting. We would be interested to hear about comparable experiences elsewhere.

5.1 Acknowledgement

I thank Rob Pooley, who was my main collaborator in the reform discussed here.

References

1. Rob Pooley and Perdita Stevens, Using UML: Software Engineering with Objects and Components. Addison-Wesley: to appear, approx Nov. 1998.
2. UML1.1 Notation Guide and Semantics, available from http://www.rational.com/uml/documentation.html
3. Open University course M355, Topics in Software Engineering: further information available from http://www.open.ac.uk/OU/CourseDetails/m355.html