Teaching Software Design and Testing

David Carrington
Department of Computer Science and Electrical Engineering
The University of Queensland
Brisbane, Queensland 4072
Australia
davec@csee.uq.edu.au

Abstract - *This paper describes a course entitled "Software Design and Testing", covering both its conceptual design and the outcomes from teaching it. The course is normally taken in semester four of either a Bachelor of Information Technology or a Bachelor of Engineering (Software) degree program (both eight semesters in length). Students enter the course with some programming experience (in Smalltalk, Ada, C or Java) but without experience with programs of non-trivial size or with cooperative software development.*

Overall, students find this a challenging and demanding course that introduces them to many new ideas. They are required to work cooperatively in teams of three or four, with their final grade partly determined by team effectiveness. Although some students have difficulty working in a team, the overall effect is very positive. Many students acknowledge that they do not appreciate the potential difficulties of working in a team until they experience them first hand.

Introduction

Figure 1 provides a concept map of some[1] of the important relationships between students, teachers and the activities that typically comprise a course. The purpose of the concept map is to expose my understanding for discussion. My role as teacher is to provide conditions and materials that I believe are conducive to learning and to be part of a dialogue to resolve issues and problems that inhibit student learning. I think that we need to be conscious of the learning strategies we use, and that we need to understand how to monitor and assess their effectiveness. I understand that some students may not be used to thinking about how they learn and that consideration of the learning process may seem to conflict with course content demands. However, to produce life-long learners, we need to go beyond Nike's "Just do it" by using reflection to "Do it better".

I am a strong supporter of active learning, seeing learning as a process of change, not just of preparation. Cooperative learning is a particular form of active learning that uses small groups to provide a supportive learning environment.

[1] Inevitably, such a diagram represents a choice about which entities and relationships to emphasise, and which to ignore.

The remainder of the paper describes the course content, the activities chosen to encourage learning and their outcomes, and some ideas for future development.

Course Focus

Software design and testing are central activities within the software development process. This course is responsible for exposing students to:

Software Design: the stage of software development that transforms a specification into a structure suitable for implementation. Teaching about software design must encompass both the product of the design activity (the design object) and the process by which this object is created. The importance of good software design increases with system size and complexity, a point that makes small convincing examples difficult to find.

Software Testing: a verification and validation technique that ensures software is developed to meet both its specification and its users' needs. While testing cannot occur until the software has been developed, preparation for testing can and should occur in parallel with specification and design activities. Teaching about software testing should cover preparation activities like the design of test cases, the execution of tests, and the comparison of actual and expected results to see if the software behaves as intended.

While the two topics are often taught independently, there are benefits from combining them. Students enter the course with some programming experience (in Smalltalk, Ada, C or Java) but without experience with programs of non-trivial size or with cooperative software development. The stated aims for the course are to:

- Demonstrate systematic approaches to software design and testing. The Fusion methodology [6, 9] is used as an example of an object-oriented design technique (although students are made aware of the rapid changes currently occurring in this sphere).
- Increase students' programming experience, particularly in terms of program size, but also in the use of additional program structures and development methods.

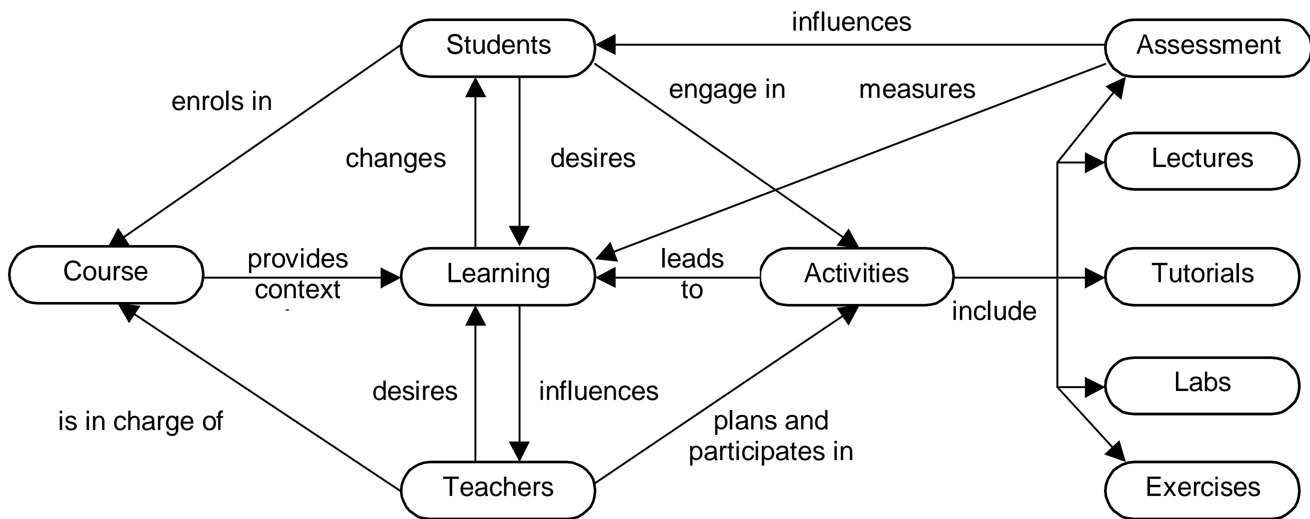

Figure 1. Course activity concept map

Implementation is important so that students observe the consequences of design decisions (and omissions!).

- Provide students with positive experiences of collaborative learning (including software development in teams) and some appreciation of the need for life-long learning skills.

There is no single pre-eminent software design method, and even if there were, the rate of change in the discipline makes it important for students to be aware of more than one method for designing software. Conversely, it is easy to confuse students by overloading them with details of multiple methods so that they fail to appreciate the similarities and differences. The challenge is to provide more than just superficial comparisons while not trying to make students into 'instant experts' in each method.

Fusion was chosen as the primary design method because it provides comprehensive coverage from analysis through to implementation. It also emphasises process issues including checklists for verifying the completion of each phase. Fusion is a method that integrates and extends earlier object-oriented methods, and it is widely used, especially at Hewlett-Packard where it was developed. Development of Fusion is continuing with recent activity integrating the new Unified Modeling Language (UML) [3] that aims to reduce the plethora of object-oriented notations.

The software testing component of the course emphasises systematic approaches to test definition and execution. Both 'black-box' (functional) and 'white-box' (structural) testing are defined and explained. A variety of test generation strategies are demonstrated and the different types of test scaffolding (harnesses and stubs) are introduced. The techniques are illustrated using Ada examples that show how the Ada package mechanism can be used to advantage.

The course is not project-based because that approach is used in subsequent semesters; instead, a case-study approach [5] is adopted to give students experience with multiple problems and multiple solution techniques.

Learning Activities

Lectures in this course are traditional, serving to introduce students to new software design and testing techniques, and to present examples that illustrate how these techniques can be applied. Lectures are presented to about 150 students. All lecture material is readily available for students so there is no need for students to transcribe the lectures. Lectures provide an opportunity in the future to increase active learning by students.

Tutorials are small group classes (ten to twenty students) subdivided into teams of three or four students who work together both in the tutorial class and on assignments. Within each tutorial, these teams prepare solutions to tutorial exercises and present their solutions to the rest of the tutorial class [1].

The students choose their teams themselves, subject to the requirement that all members of a team must attend the same tutorial class. The decision to allow students to form their own teams is deliberate. Since this is the first team activity for most of these students, choosing their fellow team members makes the exercise less threatening and the learning ultimately more effective. Lecture two is used to present information about teams, including factors that make them effective and some of the typical problems faced by student project teams. The first tutorial uses one of the classic decision-making exercises for teams from Johnson and Johnson [8] to give teams experience of working together.

Tutorial exercises on testing include developing black-box test cases from simple informal specifications and white-box tests from small program fragments. The black-

box exercise is interesting since it demonstrates the ambiguity problems that usually arise with specifications expressed in natural language. Having to derive concrete test cases highlights any ambiguities and emphasises that test case generation can also play a verification and validation role.

As well as exercises to help students become familiar with the Fusion notation and process, one exercise aims to encourage students develop a wider perspective on the challenges of software design. Thought-provoking articles, such as [10], [7] or the profiles in [12], are set as stimulus material. Each team of students summarises one article and leads a brief discussion on the main message in that article.

Assignments are the primary learning activity. One of the challenges is to make these suitable for small teams of students, that is, small enough to be manageable but large enough to create interdependence between team members. A poor assignment is one that could be more easily done by one person than by the team collectively.

Three assignments are used to give students practical experience. The first assignment presents the students with the specification of a program (in natural language and as a formal Z specification [11]) and three binary implementations. The required task is to plan a suite of (black-box) tests and execute them on each implementation to identify all discrepancies between the specification and the implementation. Students are encouraged to build simple test scaffolding (primarily test harnesses) to allow test execution to be automated. Automatic test tools are not introduced as I consider it better for students to gain experience with manual test execution and consequently then appreciate the role of automation. In 1997, the specified program was a dependency management system that managed a collection of dependency relationships.

The second assignment presents the students with the Fusion analysis models for a program. Their task is to develop corresponding Fusion design models. In 1997, the program was an interactive spelling checker, similar to the Unix `ispell`. Test suite design is also part of the assignment. The third assignment follows on from the second and requires the students to continue the development through to implementation. To help students who might have struggled with assignment two, a design solution is provided that can be used as the basis for assignment three without penalty. It also serves as a model answer. Students are required to execute their test suite from assignment two (it may be improved without penalty) and report the results.

When each assignment is handed in, the team of students must nominate the percentage contribution of each member. The mark for the assignment is then divided between the team members according to the percentage with the proviso that no individual can get more than full marks for an assignment. The proviso precludes, for example, a team of three choosing to tackle one assignment each thus avoiding any collaboration at all. The allocation of contribution is a team responsibility but it may be discussed with the teaching staff if necessary. I have chosen to have a public (rather than private, for example [2]) allocation because I believe it is important for teams to learn how to deal openly with important issues (for some students, it is the most important issue).

The major reflective activity that students in this course are required to do is associated with the monitoring of their study time. It is described elsewhere [4].

Outcomes

Overall, students find this a challenging and demanding course that introduces them to many new ideas. They are required to work cooperatively in teams of three or four with their final grade partly determined (40%) by their effectiveness as a team. Although some students have difficulty working in a team, the overall effect is very positive. In 1997, all 38 teams handed in all three assignments. Many students acknowledge that they do not appreciate the potential difficulties of working as part of a team until they experience them first hand.

Student feedback collected anonymously at the end of the course on standard University of Queensland questionnaires rates this course as only average. The main areas of complaint are the time monitoring exercise and the quality of computing equipment in the undergraduate Unix laboratories. Many students regard the time monitoring as "demeaning" and "a waste of time", although most agree that it is relatively easy to satisfy the assessment criteria. The Department acknowledges that the Unix laboratories need a substantial upgrade. Students would also like more material on the Fusion method, particularly examples, and more opportunities to consult academic staff about their assignments. While tutorials are the primary forum for consultation, the set exercises require most of the time.

The third assignment, which involves transforming a Fusion design into an implementation, proved to be very difficult for many student teams. One source of difficulty was that Ada-95 was the programming language of choice for most of the students. While Ada-95 supports object-orientation, translation of Fusion design models to Ada-95 is more involved than the equivalent translation to C++ or Eiffel, the two major languages used in the Fusion references. Although it is possible to produce a solution in about 600 lines of Ada code, students do not have sufficient experience to manage programs of this size. The program is designed and written as a team effort, so this assignment is probably their first experience with the integration problem.

One of the most pleasing aspects of this course is the significantly reduced incidence of plagiarism. In similar courses with individual assignments, plagiarism had always been a serious problem. Some students seem to be willing to take the risk of detection (which would lead to losing all associated marks and potentially to course failure), often because they feel they have no-one to help them do the

assignment properly. In 1997, standard checks to identify suspiciously similar implementations were applied to the electronic submissions for assignment three. Two pairs were identified but it turned out that each pair came from a single team as more than one team member had chosen to submit their team's final product.

The products of the assignments (software test suites, designs and implementations) are difficult and time-consuming to assess. An additional problem is maintaining consistency when multiple markers are required to handle the large class size. Having students work in teams helps with the assessment task by reducing the number of submissions and significantly improving the average quality.

Future Plans

Many of my objectives for cooperative learning in teams have been achieved, but improvements are required so that more students regard it as a positive experience. Having past students describe their experiences and provide advice may help get the message across about how to work effectively in a team.

For the implementation assignment, I need to compensate for the students' inexperience by providing more assistance, with explicit attention to the difficulties associated with software integration if the components do not conform to clearly defined interfaces. In that respect, it may help to force the teams to define explicit interfaces in their chosen implementation language as part of the design assignment. The advantage would be that these interfaces could be checked as part of the design assessment and feedback provided before implementation proceeds.

CASE tools for Fusion would help students construct and check at least the syntactic correctness of their designs. It would also reduce the load on human markers who could concentrate on verifying the appropriateness of the design relative to the specification. Of course, CASE tools are no panacea since they typically have a large learning curve and are normally not designed for novice use.

Each time the course is offered, I will develop more Fusion examples to meet the request from students for more variety in the examples used to illustrate the method.

Conclusions

Teaching software design and testing is an important and challenging task within the computer science and software engineering curriculum. It is important because of the central role of these activities in the software lifecycle. It is challenging because of the numerous methods and techniques that students should have some familiarity with. This paper presents one approach to this task based on a philosophy that emphases active learning for students in cooperative teams.

Acknowledgments

Planning for this course was my project for the Graduate Certificate in Education (Higher Education). I greatly appreciate the feedback and encouragement from my fellow students and Jim Butler as team leader. Teaching this course would have been much harder without the assistance and advice of Ian MacColl who also obliged by reading a draft of this paper and providing valuable feedback. I also thank Ian Hayes, Eric Salzman and Paul Strooper in their role as tutors; their efforts are very much appreciated.

Bibliography

[1] Bakker, P., Carrington, D., Goodchild, A., Hayes, I., Purchase, H. and Strooper, P., "The communicating technologist: an educational challenge", *Proc. 25th Frontiers in Education Conference*, 1995, pp. 4a4.1-4a4.4.

[2] Brown, R.W., "Autorating: Getting individual marks from team marks and enhancing teamwork", *Proc. 25th Frontiers in Education Conference*, 1995, pp. 3c2.15-3c2-18.

[3] Booch, G. Jacobson, I. and Rumbaugh, J., "The Unified Modeling Language", technical report, Rational Software Corporation, January 1997.

[4] Carrington, D., "Time monitoring for students", *Proc. 28th Frontiers in Education Conference*, 1998.

[5] Clancy, M. and Linn, M., "Case studies in the classroom", *Proc. 23rd SIGCSE Annual Conference*, 1992, pp. 220-224.

[6] Coleman, D. et al., *Object-oriented development: The Fusion method.* Prentice Hall, 1994.

[7] Glass, R., "Creativity and software design: the missing link", *Information Systems Management*, 9(3):38-43, 1992.

[8] Johnson, D. and Johnson, F., *Joining together: group theory and group skills*, Allyn & Bacon, 1994.

[9] Malan, R. et al., *Object-oriented development at work: Fusion in the real world*, Prentice Hall, 1996.

[10] Parnas, D. and Clements, P., "A rational design process: how and why to fake it", *IEEE Transactions on Software Engineering*, 12(2):251-257, 1986.

[11] Spivey, J. *The Z notation: a reference manual,* second edition, Prentice Hall, 1992.

[12] Winograd, T. et al., *Bringing design to software*, Addison-Wesley, 1996.