TEACHING SOFTWARE DESIGN WITH OPEN SOURCE SOFTWARE

David Carrington[1] and Soon-Kyeong Kim[2]

Abstract *– When an introductory course on software design and testing was revised, it was decided to use open source software tools as the major examples and objects of study. The goal was to expose students to realistic software systems and give them experience dealing with large quantities of code written by other people. Using open source software also has the beneficial effect of ensuring that students are aware of the open source software movement, and opens up opportunities to discuss topics such as software piracy and ethics.*

Index Terms – open source software, reverse engineering, software design, UML.

INTRODUCTION

A major challenge in teaching software engineering is helping students understand the differences between the small programs that they write for themselves and the large scale software products that they will deal with when they are working. One facet of this challenge is helping students appreciate the activity of software design where the structure and the behaviour of the software to be built are envisaged prior to its construction and modification.

This paper discusses how open source software engineering tools can be used as classroom materials in a software design course. Students gain experience with a product containing a large amount of source code. Because many open source projects do not provide design documentation, students can undertake reverse engineering activities to understand the structure of such tools. The benefits of abstract representations of a software design become apparent in this situation, and when modifications are required.

The paper reports our experiences revising a second year software engineering course on design and testing [2] for a class of more than 260 students. The practical work in the previous version of the course required students to design, implement and test a new software product in teams of three or four students. The practical work in the revised course requires similar student teams to study and modify one of ten different open source software engineering tools. By changing the emphasis from new development to reverse engineering and maintenance activities, we want students to gain experience reading and understanding other people's code, and with much larger products than they could construct on their own.

The concept of sharing software source code is not new and has been common in academic and research communities since computing began. Many open source projects started life in universities. However, the open source software movement has grown rapidly in the last decade with the expansion of the Internet and now involves many commercial and industrial participants.

While not a true open source project, the original distribution of UNIX to universities and research organisations in the mid 1970s included full source code and this had major consequences. Early UNIX adopters tended to share their extensions and adaptations. UNIX became widely used by students and gave rise to computer science's most famous "suppressed" (to protect AT&T's trade secrets) publication when John Lions [5] documented the UNIX kernel source code to help teach operating system principles.

Open source projects provide a wealth of materials for students to study. Not all are exemplars but it is valuable to expose students to a variety of programming styles and encourage their critical abilities. Hunt and Thomas [4] refer to this type of study as "software archaeology" but an alternative analogy is the medical student pathology laboratory.

Other educators are beginning to discuss how they are using open source software to attain educational goals. Andrews and Lutfiyya [1] used some GNU software products in a senior software engineering course focusing on software maintenance. At the SIGCSE conference in 2002, a panel reviewed the social and ethical responsibilities associated with using open source software [8]. O'Hara and Kay [7] provide an overview that includes a description of some of the different open source licences. Nelson and Ng [6] describe a course on computer networking that relies on multiple open source packages.

Using open source software also has the beneficial effect of ensuring that students are aware of the open source software movement, and opens up opportunities to discuss topics such as software piracy and ethics.

The remainder of this paper covers the context of the revised course, its goals, teaching mechanisms, and the results.

[1] David Carrington, School of Information Technology and Electrical Engineering, The University of Queensland, St Lucia 4072, Australia, davec@itee.uq.edu.au
[2] Soon-Kyeong Kim, School of Information Technology and Electrical Engineering, The University of Queensland, St Lucia 4072, Australia, soon@itee.uq.edu.au

0-7803-7961-6/03/$17.00 © 2003 IEEE

TEACHING CONTEXT

The software design and testing course is a core element of the second year program for both the software engineering and the information technology degrees offered by the School of Information Technology and Electrical Engineering at the University of Queensland. Incoming students have previously completed two programming courses using Java and are typically studying a data structures and algorithms course in parallel.

The one semester (13 teaching weeks) course is structured with three lectures and two tutorials per week. Each lecture is for the full class of over 260 students while the tutorials are for groups of up to thirty students (up to eight teams). Thus each tutorial class is offered ten times per week. Members of each student team are constrained to attend the same tutorials so they can work as a team during this time. Team presentations are also required in some tutorial classes.

The course introduces students to the practical aspects of configuration management using CVS (Concurrent Versions System - an open source version control system). Students have been introduced to UNIX in a prior course and are required to install their open source tool in our UNIX labs.

Assessment for the course consists of four team assignments, a time monitoring exercise [3], and a final open-book exam.

GOALS

The goals for the course are to:

- Demonstrate the concepts and practice of software design and testing, the UML notation, software design patterns, code refactoring and configuration management.
- Extend students' programming experience, particularly in terms of program size but also in the use of additional program structures and development methods.
- Provide students with positive experiences of collaborative learning and some appreciation of the need for life-long learning skills.
- Expose students to open source software tools and real world code written by other developers.

TEACHING MECHANISMS

Prior to the beginning of the teaching semester, we made a short list of open source software engineering tools written in Java that either we knew about or found using the search facilities at SourceForge (sourceforge.net). An objective was that the tool should be potentially relevant to the students in their software engineering activities. The initial list contained:

- ArgoUML (argouml.tigris.org): a UML modelling tool
- Eclipse (www.eclipse.org): an extensible IDE
- JEdit (www.jedit.org): a programmer's text editor
- JRefactory (jrefactory.sourceforge.net): a tool to perform refactoring on Java source code
- JUnit (www.junit.org): a regression testing framework
- NetBeans (www.netbeans.org): an extensible IDE
- Process DashBoard (processdash.sourceforge.net): a Personal Software Process (PSP) support tool

The list was intended to give students a core set of tools, but students were encouraged to find and suggest other suitable tools. Several student teams did so and nominated the following tools:

- Ant (jakarta.apache.org/ant): a build tool that uses XML configuration files
- DocWiz (www.mindspring.com/~chroma/docwiz): a GUI tool to add JavaDoc comments to Java source code
- PMD (pmd.sourceforge.net): a static checker for Java source code

The number of student teams studying each tool is summarised in the following table. Within each tutorial group, student teams were strongly encouraged to choose different tools to study.

TABLE I
NUMBER OF STUDENT TEAMS FOR EACH JAVA TOOL

Java Tool Name	Number of Teams
Ant	4
ArgoUML	10
DocWiz	1
Eclipse	10
JEdit	11
JRefactory	9
JUnit	5
NetBeans	12
PMD	5
Process Dashboard	3

To help students understand what we expected them to do, an early tutorial exercise used a small case study (460 non-comment source lines in eight classes) provided by the Distributed Systems Technology Centre (DSTC) at the University of Queensland. This case study was based on a small Java program that provided an automated test harness for Java (it predates JUnit). We asked the students to read the supplied source code to identify the classes and their relationships. To capture their understanding, teams built a UML class diagram with associations between classes based on references in the code. We encouraged students to focus on the most important aspects of the system rather than dealing with every method and field. A design rationale document was supplied by the DSTC developers to explain how the system architecture satisfied the product requirements.

The practical work for the course was structured as four team assignments that required an increasing depth of

knowledge and understanding about the selected open source tool.

Assignment 1

The first assignment asked each team to select a tool. Each team was required to install and use their tool prior to giving a five minute presentation to their tutorial class that explained:

- the purpose of the tool,
- the installation process, and
- how to use the tool, preferably with a simple example.

The primary purpose of this assignment was to get teams started and to overcome any difficulties with tool installation and use. A secondary goal was to display a range of software engineering tools to students so they were aware of the tools being studied by their peers, and so they could use them if they identified a need.

Assignment 2

The second assignment represented the first part of an intensive investigation of the source code for the selected Java tool. This assignment was intended to provide an interim milestone and offer an opportunity to receive feedback on the team's progress and plans for the additional investigation required for assignment 4. Assignment 2 had four deliverables:

1. source code descriptions,
2. CVS and UNIX build evidence,
3. a team learning journal, and
4. a plan for future research.

Using any existing documentation as a base, the team was to construct a design guide for the tool. The guide should incorporate an overview of the tool's structure including a complete list of classes and explain any structuring into packages. If the tool relies on external libraries or other systems, these should be recorded.

Since some tools are much larger than others, teams could choose to focus their detailed description on a coherent subset of the overall source code. A rationale for the selection of the subset was required. As a very rough guide to the size of a subset for assignment 2, we suggested focusing on about 12 to 16 classes or about 2,000 to 3,000 lines of code (not counting comments).

UML class diagrams were to be constructed to capture the classes and the relationships between them. Not all attributes and methods needed to be included for each class – inclusion was to be on the basis of assisting understanding. UML interaction diagrams (collaboration or sequence) were not required in this assignment but they could be used where they assisted the documentation goal.

The assignment had to include evidence of the team's use of CVS to maintain the Java tool source code and their ability to build the tool from source under UNIX. Since no source modifications were required for assignment 2, the CVS evidence was merely to confirm that each team had been able to put the source code under CVS control successfully. The CVS logs and status information provided the primary evidence.

Each team was required to keep a journal of their activities, experiences and ideas over the course of this assignment. The following questions were suggested as a guide for what to include in the team journal.

1. How, in your opinion, does the theory of software design help with the practical task of understanding the internal workings of a large software system?
2. What strategies are you using to understand the source code of your team's Java tool?
3. What resources have you found useful? (How and why).
4. What mistakes has the team made, and how will you avoid making the same mistakes in the future?
5. What do you now know or understand that you didn't know or understand at the beginning of the semester?
6. For your team, to what extent have the goals of this assignment been achieved?
7. What do you think you need to know to make further progress in understanding software design?

The final part of assignment 2 was a plan for the next stage of investigation of their tool. While this plan was not binding on the team, it represented an opportunity to prepare for assignment 4. The plan was to propose some extensions, improvements or refactorings to the Java tool that the team felt could be implemented for assignment 4. These tool enhancements were to be described as precisely as possible without getting into implementation details (i.e., what is to be extended, improved or refactored and why, but not how this is to be achieved). A test plan was also required for the parts of the tool to be affected by the proposed extensions, improvements or refactorings. The test plan was to describe the class(es) to be tested, the proposed test process including any required test scaffolding, and the general types of tests that the team proposed to execute.

Assignment 3

For the third assignment, each team had to prepare and deliver a 15 minute presentation about the internals of their Java tool covering:

1. the overall tool structure
2. the subset studied in detail
3. the UML diagrams developed
4. any patterns identified
5. extensions or refactorings planned or completed
6. planned or completed testing related to extensions or refactorings

Because the presentations for assignment 3 were longer than for assignment 1, the presentations in tutorials were spread over three weeks. This meant that teams that presented earlier emphasised their plans while teams that presented later reviewed their achievements.

0-7803-7961-6/03/$17.00 © 2003 IEEE

Assignment 4

The final assignment required each team to extend their tool in some way, either by adding functionality or by refactoring the existing code. The deliverables were similar to assignment 2:

1. source code changes and accompanying descriptions,
2. CVS and UNIX build evidence,
3. UML diagrams illustrating the source changes,
4. testing information describing how the changes to the Java tool had been verified, and
5. an extended team learning journal.

RESULTS AND DISCUSSION

Overall, the use of open source software helped achieve the goals of the course. Compared to earlier versions of the course, using open source software provided a reverse engineering and maintenance focus, rather than concentrating on new development. This change of focus is considered to provide a more realistic experience for students, although some students wanted more coding (only one assignment of the four required coding skills).

The selection of Java tools could be improved. Ideally the students should find it easy to grasp the purpose of the tool and how to use it. Large multi-purpose tools like NetBeans were generally found to be overwhelming without extra assistance. NetBeans comes with comprehensive documentation, but this is itself a challenge for students to absorb. The sheer size of the source code for NetBeans and Eclipse in particular was daunting. Student feedback at the end of the course indicated that they felt that the initial choice of a tool had unduly influenced their results: *"The tools should be equally difficult"* and *"It is obvious that some tools are much harder to understand and do the assignments on. I feel this disadvantages students who have to do a larger, more complicated tool (i.e. NetBeans)"*. JUnit was probably the easiest of the suggested tools to understand. In future, we plan to use JUnit as an exemplar in lectures and tutorials.

Some students found it difficult to imagine that they might be able to extend or improve the work of other people: *"It's hard for us to say that our design could possibly be better than the original programmers."*

Student teams generally did not report using other tools, for example, JRefactory (or ArgoUML) for reverse engineering, or JUnit for testing. The reason seems to be lack of time to learn and understand how these other tools might assist them.

We need to provide more assistance with CVS. We had anticipated that an overview lecture would be sufficient to get the teams started, but some teams struggled throughout the semester to master CVS. Our current plan is to provide some practical exercises at the beginning of the semester under supervision.

Reviewing the students' responses to the assignments, the first two were successful with most teams able to handle both of them. The mean mark for assignment 1 was 7.8/10 (sd = 1.3) and for assignment 2 was 12.7/20 (sd = 3.1). The timing of assignment 3 was problematic, falling as it did between assignments 2 and 4 so that it represented a progress report. Some student teams had difficulty presenting both plans and results, and feedback suggested that they would have preferred this presentation to be the final assessment item. Compared to assignment 2, the assignment 4 results were more varied. Some teams felt that assignment 4 was just "more of the same" even though the emphasis was intended to be on design and code extensions. Having similar assignment deliverables including the team journal probably contributed to this sense of "sameness". Weaker teams found it difficult to make meaningful extensions or refactorings. Also for some teams, their overall workload at the end of semester constrained the time available for this assignment. The mean mark for assignment 3 was 11.7/15 (sd = 2.2) and for assignment 4 was 27/40 (sd = 7.4).

An anonymous survey was administered in week 8 of the semester just after assignment 2 was submitted. The purpose of the survey was to get feedback on students' perceptions and feelings about the course. The response rate was good (149 out of 264) because the survey was administered in the tutorial classes.

The survey asked about how the team was performing, reactions to the team's Java tool, and course organisation. Responses were on a 5 point scale: strongly agree (SA), agree (A), neutral (N), disagree (D) and strongly disagree (SD). The questions about the Java tool were ("My Java tool is"):

1. easy to understand
2. giving me real-world software experience
3. helping me understand software design and construction
4. helping me improve my programming skills
5. helping me improve my knowledge of Java
6. an enjoyable way to learn this subject matter

The results for question 1 are shown in Figure 1. While the class mean was 3.0, there were some interesting tool-specific differences: PMD = 2.2, JUnit = 2.6, NetBeans = 3.3 and ArgoUML = 3.6.

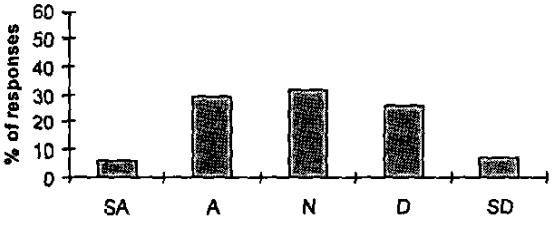

FIGURE 1
MY JAVA TOOL IS EASY TO UNDERSTAND

The mean for question 2 was 2.3, again with some variation between tools: Process Dashboard = 1.9, JUnit = 2.0 and Ant = 2.7.

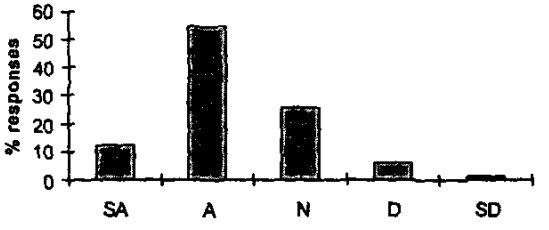

FIGURE 2
MY JAVA TOOL IS GIVING ME REAL-WORLD SOFTWARE EXPERIENCE

For question 3 the mean was 2.4.

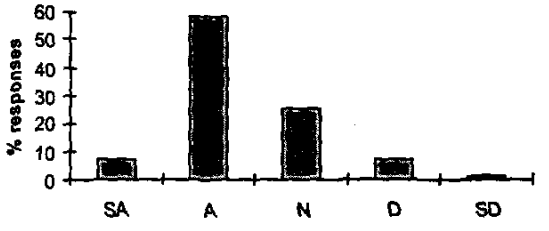

FIGURE 3
MY JAVA TOOL IS HELPING ME UNDERSTAND SOFTWARE DESIGN AND CONSTRUCTION

The class means for question 4 was 3.0 with JRefactory = 2.6 and NetBeans = 3.3.

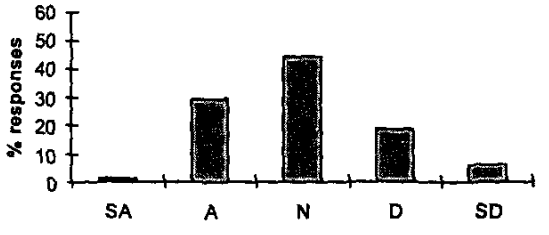

FIGURE 4
MY JAVA TOOL IS HELPING ME IMPROVE MY PROGRAMMING SKILLS

The mean for question 5 was 2.7 with JUnit = 2.2, ArgoUML = 3.0 and NetBeans = 3.2.

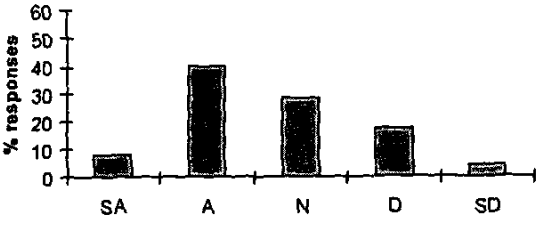

FIGURE 5
MY JAVA TOOL IS HELPING ME IMPROVE MY KNOWLEDGE OF JAVA

The question 6 mean was 2.9 (Ant and Eclipse = 2.6).

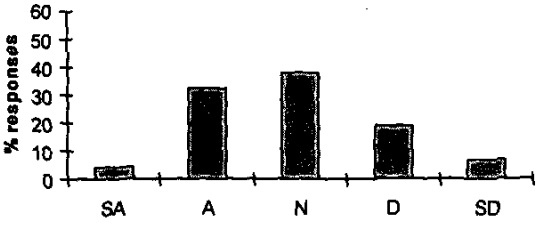

FIGURE 6
MY JAVA TOOL IS AN ENJOYABLE WAY TO LEARN THIS SUBJECT MATTER

Our general conclusion is that students appreciate the benefits of this approach to studying software design and testing but at this point in the course, they were not convinced that it was easy or enjoyable. Since for assignment 1 and 2 no programming was required, the response for question 4 about developing programming skills is understandable.

CONCLUSIONS

The open source software movement provides wonderful resources for teaching software engineering. Our experience using open source Java tools in this course on software design and testing was positive, although there is room for improvement in the future. Students appreciated the opportunity to use and explore real world software and to study its internal construction.

For teaching staff, the challenge is to provide adequate support and scaffolding for open source learning since they are unlikely to be familiar with all the details of the tools. They need to be able to demonstrate how to reverse engineer software by exploring and extracting critical information from the source code.

There are some open issues which we have not yet considered:

- Can students' work contribute to the open source community through feedback, software changes or documentation?
- Can the open source community interact with the students other than by providing their software?

ACKNOWLEDGMENTS

David would like to acknowledge the influence of John Lions for his pioneering effort demonstrating the value of studying the UNIX operating system through its source code.

We acknowledge the significant efforts of our tutors (teaching fellows) for this course who each took responsibility for one or more open source tools and became the primary point of contact for student queries. The contributions by Daniel Jarrott, Matthew McGill, Penny Sweetser, Tim Cederman and Tim McComb are very much appreciated. Ron Chernich and Liz Armstrong from the Distributed Systems Technology Centre were most helpful in providing the automated test harness case study. We also acknowledge the efforts of the students in this course who accepted the challenge to study large and complex software systems.

REFERENCES

[1] J.H. Andrews and H.L. Lutfiyya. Experiences with a Software Maintenance Project Course, *IEEE Trans. Education*, November 2000, 43, 4, pp. 383-388.

[2] D. Carrington. Teaching software design and testing. *Proc. 28th Annual Frontiers in Education Conference (FIE'98)*, 1998, pp. 547-550.

[3] D. Carrington. Time monitoring for students. *Proc. 28th Annual Frontiers in Education Conference (FIE'98*, 1998*)*, pp. 8-13.

[4] A. Hunt and D. Thomas. Software Archaeology, *IEEE Software*, March/April 2002, 19, 2, pp. 20-22.

[5] J. Lions. *Lions' Commentary on UNIX 6th Edition, with Source Code*, Peer-to-Peer Communications, 1996.

[6] D. Nelson and Y.M. Ng. Teaching Computer Networking using Open Source Software, *Proc. ITiCSE 2000*, pp. 13-16.

[7] K.J. O'Hara and J. S. Kay. Open Source software and Computer Science Education, *Journal of Computing Sciences in Colleges*, 2000, 18, 3.

[8] M.J. Wolf (Moderator). Open Source Software: Intellectual Challenges to the Status Quo, *Proc. SIGCSE Conference*, 2002, pp. 317-318.