Assessing Undergraduate Experience of Continuous Integration and Test-Driven Development

Jon Bowyer
Applied Computing
University of Dundee
Dundee DD1 4HN, UK
+44 1382 385195
jbowyer@computing.dundee.ac.uk

Janet Hughes
Applied Computing
University of Dundee
Dundee DD1 4HN, UK
+44 1382 385195
jhughes@computing.dundee.ac.uk

ABSTRACT
A number of agile practices are included in software engineering curricula, including test-driven development. Continuous integration often is not included, despite it becoming increasingly common in industry to code, test, and integrate at the same time. This paper describes a study whereby software engineering undergraduates were given a short intensive experience of test-driven development with continuous integration using an environment that imitated a typical industrial circumstance. Assessment was made of students' agile experience rather than of project deliverables, using a novel set of process measures that examined students' participation and performance in agile testing. Results showed good participation by student pairs, and clear understanding of agile processes and configuration management. Future work will investigate automation of the assessment of continuous integration and configuration management server data.

Categories and Subject Descriptors
K.3.2 [**Computers & Education**]: Computer and Information Science Education - *Computer science education*

General Terms
Design, Measurement, Verification, Human Factors.

Keywords
Agile practices; test driven development; JUnit; continuous integration; software engineering education; assessment process.

1. INTRODUCTION
Extreme Programming (XP) is a key component of the set of "relatively light" adaptive software development methods known as agile practices. Agile practices have prompted an amount of excitement and debate in industry and education recently, e.g., [6]. The component practice advocated most frequently is pair programming, whereby code is co-authored by two programmers who work side-by-side at the same machine. Other practices have been given somewhat less attention, notably test-driven development (TDD) with continuous integration (CI). Continuous integration is used in most industrial projects that are developed using agile methods.

ICSE'06, May 20-28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

In such a system, developers keep their code and accompanying unit tests in a version control server, which is continuously monitored for changes by a continuous integration server. When changes are detected, the continuous integration server executes a build script for the project. Typically the build script retrieves the latest versions of all the code and test classes, compiles the code and tests, then runs the tests. If code fails to compile or a test fails, the build is said to have failed, otherwise it is said to have succeeded. This build result is then published to the developers – usually sent by email and/or via a build results intranet webpage.

TDD and pair programming were introduced into a software engineering module at the University of Dundee in 2004 but without student assessment. In 2005 and 2006 the experience was revised: (a) continuous integration was included; (b) the three agile practices (CI, TDD, and pair programming) were made the focus of an assessed project that contributed to a student's degree award; and (c) a reflective component was introduced. To maximize the real-world relevance of the project, the lab environment was set to mimic the use of agile practices in industry. To maximize the use of agile practices, the project was designed to assess the *process* of TDD and CI. Students were to be given credit for taking an agile testing approach, and reflecting upon that approach, rather than for production of a deliverable. A key question to be addressed was how to assess their approach*:* how can (i) participation and (ii) performance in agile testing be measured? The remainder of this paper describes related work (section 2), the agile testing project (section 3), the assessment scheme adopted (section 4), lessons learned (section 5), and future work (section 6).

2. RELATED WORK
A number of positive and enthusiastic experience reports over several years have encouraged academics to apply agile practices to education e.g., [16]. Extreme programming by student teams has been found to produce as good results, in terms of external and internal quality, as those of teams following a design-led approach [13]. Various recommendations have been made to make the agile experience as valuable and as real as possible. These include keeping courses short and providing adequate amounts of lab time for students [5]. One attempt to simulate production programming in the classroom using XP [1] was judged successful, although improvements may have resulted from a less flexible attitude to

deadlines, and the requirement to spend a minimum amount of time per week on the assignment.

Pair programming has featured most regularly and positively: relative to that, continuous integration has been given less attention. Some reports have described problems implementing test-driven development, for example as a result of technical issues [12]. Where a suite of XP practices was applied, test-driven development was identified as the most difficult for students to adopt [14]. However, there is incentive for researchers to persevere: a study by [4] suggests that integrating software testing (not specifically XP) with software design can be beneficial for students, for example enhancing learning in design quality, test planning, and testing & quality assurance. Similarly and more recently, integration of testing and the JUnit framework across the curriculum was found to increase students' understanding and appreciation of the overall value of testing in software development [15]. One review of the use of a test-driven approach in the classroom that summarised four experience reports and three experimental design studies [10] indicates a number of potential benefits, with improvements in collaboration, software quality, programmer productivity, student enthusiasm and completion rates for assignments.

An interesting development was to combine test-driven development experiences for students with an automated assessment tool that provides them with immediate feedback [2]. The "Web-CAT Grader" tool assess students' tests on the basis of validity (accurate and consistent with the problem assignment), completeness (coverage of the problem), and correctness (how many of a student's own tests his or her code will pass). Edwards reports a comparison of the programs produced by a 2003 student cohort, which had experienced the test-driven development with automated assessment, with those produced by a 2001 student cohort that had *not* used a test-driven development approach. The former were found to have fewer test case failures than the latter, and fewer defects per thousand lines of code [2]. Edwards speculates that it would be possible to "infuse continual practice and development of comprehension, analysis and hypothesis-testing skills" via the use of test-driven development across programming assignments in the computer science curriculum [3], and suggests that testing can promote a "Reflection in action" approach by students. This report examines how work such as [14] can be progressed by including another agile practice – continuous integration – in the curriculum. The approach taken is detailed in the next section.

3. AGILE TESTING PROJECT

3.1 Tools

A pedagogic aim of the work was to give students practical experience of agile practices, particularly test-driven development and continuous integration. It was judged that the students' effort should be focused on the experience of TDD within a CI environment, rather than on background technical procedures to set up a continuous integration server. The configuration of the continuous integration system was therefore carried out for the students prior to the start of their assignment. Java was selected as the programming language with which the students had greatest familiarity and most recent experience. The toolset used for the assignment was chosen to closely approximate that used in an industrial setting employing agile practices. The most commonly used continuous integration servers are CruiseControl [8] and Anthill [9] – both use the Java-based tool Apache Ant [7] for build scripts, and both support a variety of version control servers. CruiseControl was used in this instance, with Microsoft's Visual SourceSafe as the configuration management (CM) version control server holding the code repositories which CruiseControl was to monitor.

For each pair of students, the configuration required the creation of a CruiseControl project, a SourceSafe repository and an Ant build file, and the linking of these together. The CruiseControl project configuration regularly scanned each pair's SourceSafe repository for code changes. If changes were detected, CruiseControl invoked the corresponding Ant build file. The build file (i) retrieved the latest versions of all files from the pair's SourceSafe repository and stored them in a temporary local directory; (ii) executed the Java compiler to compile any Java classes; then (iii) launched the JUnit test runner to execute any Java unit tests. The CruiseControl project configuration then emailed the build results to each student in the pair, and also sent copies to the module tutors. Principal information in the email was identification of the student pair, the build ID, its date and time, when it was last changed, the result of unit tests run, and a summary of the modifications made since the last build. Results were also automatically posted to an internal CruiseControl website running on a Tomcat web server. Students were given an overview of the system and its processes, and then instructed to devote their practical time to writing their unit tests and code.

3.2 Instruction

Participants were two cohorts of students taking a third year module in software engineering as part of their undergraduate degree program in 2005 and 2006. The sixty nine students received a small number of lectures on the principles and experiences of test driven development and continuous integration. They were given two laboratory classes to gain practice (a) writing unit tests for Java classes and (b) with the operation of the continuous integration server. The assignment was to develop a simple application for the storage and management of information. The 2005 version was to manage a DVD library. The 2006 version was to manage an e-commerce system. Requirements were intentionally limited to encourage the students to concentrate on the development process rather than to focus on achieving specific functionality. The students were required to adopt a test-driven development and pair programming approach.

Following the recommendation in [5], the time for the assignment was strictly limited: students were required to stop working upon the assignment seven working days later, with a recommended development time of twenty hours per person. Each student was asked to record at the end of each session what his or her pair had done for that session's development work, including: (i) what work had been intended; (ii) what was checked in to SourceSafe; (iii) what the result of the work was; (iv) what each person did individually during the work; (v) a personal reflection upon the value of the session test-driven development work; (vi) any other observations about the learning achieved or personal development made during the assignment period. This information was recorded using e-portfolios, an electronic log book system provided via the

University's virtual learning environment. The students were advised that there were no "final" deliverables: assessment of their work would be on the basis of the development performed, not the final product. Work performed was to be examined via the CruiseControl electronic log of what was submitted and its success or otherwise, the SourceSafe information (such as code, configuration management comments, and history), and the e-portfolio. Students organized themselves into self-selected pairs.

4. ASSESSMENT MEASURES

This section outlines the measures selected to assess the students' (i) participation, (ii) ability to perform agile testing, and (iii) their reflection upon the process. Data was collected from three sources: CruiseControl (by examination of a pair's electronic log of build attempts), e-portfolios (by examination of each student's record of work done), and SourceSafe (by examination of the version control history of a pair's code).

4.1 Participation

Participation was assessed by the number of builds per session of work, and the number of hours of development work. The former provided a measure of the regularity associated with good CI practice. The latter measured the extent of participation in terms of number of hours of effort, as indicated by the CruiseControl logs. Both were confirmed by examination of each student's e-portfolio, which record also provided evidence of pair programming.

4.2 Performance

A number of items were identified as indirect measures of students' ability to perform TDD using CI. Most measures of performance were quantitative, the data being directly and reliably obtained from the electronic artifacts of the CI and CM environment. These included the proportion of development time with a failed build; number of overnight failures, i.e. when the students left their project in a "failed build" state from one day to the next; use of the JUnit setUp and tearDown methods; number of instances of deletion of tests, i.e. when the students chose to delete a test rather than to improve their code and make it pass; number of types of assertions used; attempt to use stub classes and mock objects; number of requirements implemented that had corresponding tests; number of methods not tested.

Where qualitative measures were required, nominal scales were used to encode items, e.g. "absent", "present but very limited", "satisfactory", or "good". These included quality of test and method naming; suitability of version control comments; quality of assertion explanations; quality of refactoring; and quality of tests. Samples of these measures were double-marked to confirm reliability. Baseline values or acceptable ranges were established for the majority of quantitative measures. As an example, for the number of builds per session the expected range was 1 – 5, and tutor feedback to the students described the good practice in industry of regular builds. In some instances, student work was sampled for assessment in preference to complete but time-consuming examination of every component of their work. As an example, for instances of test deletion, only the deletions of entire tests were noted; ideally there would be examination of all tests to uncover any instances of partial deletions, such as portions being commented out. Interestingly, items requiring qualitative judgments prompted tutors to give the most detailed and informative feedback to students.

4.3 Reflection

E-portfolios were examined for evidence of reflection upon TDD and CI, with respect to personal experience, the context of the given problem, and the wider context of systems development. These were assessed on a nominal scale, and a sample re-marked to confirm the stability of the measure.

5. LESSONS LEARNED

Did the approach achieve the goals of integrating CI and TDD into the software engineering module?

- Yes, all pairs successfully performed TDD and CI using the industry-styled environment. All students obtained pass marks for the assignment, with about one quarter obtaining a first-class mark. A guest lecturer from the software engineering industry endorsed the project, and student appraisals of the project were extremely positive.

Was there considered or deep reflection about agile practices?

- Not always: reflection was often superficial, with many students scoring second-class marks for this component. The majority of attempts to reflect upon agile processes were cursory, self-centered, and limited to the present project. Generally students did not consider wider issues or contemplate its use in large-scale developments. This may reflect strategic behavior, whereby students put most effort into work that is credited with most marks or work that has industrial relevance, which they appreciate will enhance interview prospects. If reflection upon learning is deemed by tutors to be important, perhaps it needs to be given credit equal to practical work in the assessment scheme. Alternatively, it may reflect students' poor time management, whereby they left insufficient time to reflect upon extreme programming at the end of the project. However students did reflect well upon the assignment process itself, and made helpful suggestions to make the assignment more realistic and valuable. For example, there were suggestions that for 2007, student pairs should work in small groups of six or eight to integrate their user story developments; this would better simulate an industrial project team.

What did the students learn?

- Students learned about the process of continuous integration and the test-driven approach. Their work demonstrated understanding of the use of the CI server (regular check-ins, few overnight failures, and suitable ratios of successful builds to failed builds). Students' code showed evidence of TDD good practice: appropriate use of assertions, setUp and tearDown, refactoring, suitable choice of tests, and meaningful naming.

- Students obtained a deep understanding of configuration management. In previous years, configuration management had been taught, rather than learned. Students' appreciation of CM issues was raised enormously by the use of the CI server to detect code changes. Their CM comments were meaningful, and they made responsible and regular use of the CM server. A number of e-portfolio comments confirmed this observation.

What did the tutors learn?

- Firstly, to set a suitably challenging task. The too-trivial nature of the 2005 task produced a number of implementations that were so simplistic that there was insufficient incentive or opportunity to master refactoring and use mock objects.

- Secondly, that the metrics selected *did* provide a suitable basis for assessing students' participation and performance. In 2006 the students were involved with the design of the assessment, and negotiated with the tutors the final choice of metrics, which further confirmed their appropriateness. The data gathering was very time-consuming, however: - automated analysis of the SourceSafe repositories and CruiseControl logs data would be essential if the assessment process is to be efficient in terms of tutor time. Furthermore, a number of additional but desirable measures had to be excluded, and some sampling of data was necessary, for reasons of practicality: automated analysis would provide for a more thorough assessment.

- Thirdly, to support students in their choice of partner for the pair-programming process. Students' e-portfolios reflected their appreciation of the pair-programming process: almost all found it to be a valuable experience and/or could discern the benefits. However in 2005 five students did *not* have a good experience, as a result of the operation of the pairs. One pair in particular was dysfunctional, evidently because of a mismatch of abilities. The student whose programming ability was less good found the pair programming experience to be a particularly unhappy one: "*My self esteem is very low …I feel like a total failure, as if I have let down both myself and my team mate."* This appears to concur with recent findings [11] about compatibility in pair programming. In 2006 students were urged to consider this when agreeing pairs, and the problem of dysfunctional pairs did not recur.

- Fourthly, to build into the assignment schedule a period of post-project reflection time that would encourage students to review the agile processes in greater depth.

- Finally, to involve students in the design of the assessment's marking scheme. In 2006 the identification of assessment measures was a class exercise in which discussion led to agreement about which measures should attract credit. As an illustration of this process, the 2006 class voted NOT to give credit for meaningful comments, agreeing that code should be self-documenting: "if you feel the need to write a comment, return to the code to see if it deserves to be refactored instead".

6. FURTHER WORK

A number of the measures of performance were identified as desirable but impractical: a disproportionate amount of time would have been required to gather and process the data by hand, particularly with a large student cohort. Plans have been made to extend this work by devising ways in which the essential data is extracted from the CruiseControl and SourceSafe log files and processed automatically. This will significantly reduce the assessment effort for tutors.

7. REFERENCES

[1] Allen, E., Cartwright R., and Reis, C. Production programming in the classroom. In *Proceedings of the 34th SIGCSE Technical Symposium Computer Science Education* (Reno, NV, February 19-23, 2003).

[2] Edwards, S.H. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *ACM Journal of Educational Resources in Computing, 3,* **3** (Sept. 2003), 1-24.

[3] Edwards, S.H. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proceedings of the 35th SIGCSE Technical Symposium Computer Science Education* (Norfolk, Virginia, March 3-7, 2004).

[4] Frezza, S. Integrating Testing and Design Methods for Undergraduates: Teaching Software Testing in the Context of Software Design. In *Proceedings of the 32nd ASEE/IEEE Frontiers in Education Conference* (Boston, MA, November 6-9, 2002).

[5] Goldman, A., Kon F., Silva P.J.S., and Yoder J.W., Being Extreme in the Classroom: Experiences Teaching XP. *Journal of the Brazilian Computer Society, 10,* **2** (2004), 1-17.

[6] Gotterburn, D. UML and Agile Methods: In support of Irresponsible Development. *Inroads – The SIGCSE Bulletin, 36,* **2** (June 2004), 11-13.

[7] http://ant.apache.org/

[8] http://cruisecontrol.sourceforge.net

[9] http://www.urbancode.com/projects/anthill/default.jsp

[10] Jones, C.G. Test-Driven development goes to school. *Journal of Computing Sciences in Colleges, 20,* **1** (2004), 220-231.

[11] Katira, N., Williams, L., and Osborne, J. Towards Increasing the Compatibility of Student Pair Programmers. *27th International Conference of Software Engineering (ICSE 2005)* (St Louis, Missouri, 15-21 May 2005).

[12] Keefe K. and Dick M. Using Extreme Programming in a Capstone Project. In *Proceedings of Sixth Australasian Computing Education Conference (ACE 2004)* (Dunedin, New Zealand, January 18-22, 2004).

[13] Macias, F., Holcolmbe, M., and Gheorghe, M. Design-led & Design-less: One Experiment and Two Approaches. *Lecture Notes in Computer Science, 2675,* (2003), 394-401.

[14] Muller, M.M., Link J., Sand R., and Mahlpohl, G. Extreme Programming in Curriculum: Experiences from Academia and Industry. In *International Conference on Extreme Programming and Agile Processes in Software Engineering (XP2004)* (Garmisch-Partenkirchen, Germany, June 2004).

[15] Wick, M., Stevenson D., and Wagner, P. Using Testing and JUnit Across the Curriculum. In *Proceedings of the SIGCSE Technical Symposium on Computer Science Education* (St Louis, Missouri, February 23-27, 2005).

[16] Williams, L. and Upchurch, R. Extreme Programming for Software Engineering Education? In *Proceedings of the 31st ASEE/IEEE Frontiers in Education Conference* (Reno, NV, October 10-13, 2001).