Testing Test-Driven Development

Hussein Suleman, Stephan Jamieson, Maria Keet

Department of Computer Science
University of Cape Town
Private Bag X3, Rondebosch
South Africa
hussein@cs.uct.ac.za, sjamieson@cs.uct.ac.za, mkeet@cs.uct.ac.za

Abstract. Test-driven development is often taught as a software engineering technique in an advanced course rather than a core programming technique taught in an introductory course. As a result, student programmers resist changing their habits and seldom switch over to designing of tests before code. This paper reports on the early stages of an experimental intervention to teach test-driven development in an introductory programming course, with the expectation that earlier incorporation of this concept will improve acceptance. Incorporation into an introductory course, with large numbers of students, means that mechanisms are needed to be put into place to enable automation, essentially to test the test-driven development. Initial results from a pilot study have surfaced numerous lessons and challenges, especially related to mixed reactions from students and the limitations of existing automation approaches.

Keywords: programming, test-driven development, unit testing, automatic marking

1 Background and Motivation

In a Computer Science (CS) degree, students are taught how to develop software systems, in addition to other skills and techniques relevant to the discipline. Due to the rapidly-changing nature of CS as a young discipline, the recommended techniques and methodologies to develop software (and therefore what is taught) change over time, for example, from the Waterfall methodology in the 1980s to Iterative in the 1990s to Agile in the 2000s.

Traditionally, CS courses would mirror the general academic practice whereby students submit work for assessment by tutors and lecturers. Many CS programmes have now shifted the assessment of computer programs to computer-based assessment - termed Automatic Marking at the authors' institution [7]. While this assists in teaching the development of computer programs, it does not help students to develop their own mechanisms to assess their solutions. However, when students graduate, assessment is a key skill required of them by their employers, as many software development companies expect assessment of software, as a quality control mechanism, to occur hand-in-hand with software development. Frequently, however, new graduates either do not have experience or exposure to such software development skills.

While there are numerous approaches to software testing, Test-Driven Development (TDD) [1] has emerged as a popular software development approach, where the tests of software are produced first and drive the software creation process. TDD has been shown to result in programmers learning to be more productive [3, 4], while generating code that is more modular and extensible [5] and is of better quality [6]. Some early experiments have been conducted to show that test-driven development can be incorporated into the kinds of automated testing that is used in many large teaching institutions. Edwards [2] opted to measure the code coverage of tests and the adherence of student code to student tests as indicators of appropriate use of TDD. He found that students appreciated the technique and there was a clear reduction in error rates in code. However, testing required the provision of a model solution for coverage analysis, which is not always possible or accurate.

This paper describes a first attempt to incorporate TDD approaches into an automated marking system, using the alternative method of testing the tests. Where students have, in the past, only encountered advanced software development techniques in later courses, TDD was designed into an early programming course to avoid resistance from students who had already learnt a traditional approach. It was expected that this would improve the skills of students while better meeting the needs of industry. We present the initial integration in a 1st year CS course, observations on that, and the results of a survey among students.

2 Implementation

As a pilot study, elements of TDD were incorporated into specific parts of two assignments of a second semester course on object-oriented programming. Students had already learnt imperative programming and were exposed to the general notions of program testing in the previous semester.

Assignments 2 and 3 of the course were selected because they marked the start of OO content proper. It was felt that TDD could assist with mastery of the concepts; with thinking about objects and behaviour.

(Assignment 1 concerned knowledge transfer - constructing imperative programs in the new language, Java.)

2.1 Assignment 2

In Assignment 2, software testing was introduced as a means of guiding problem analysis, solution design and evaluation. Students were: (i) given tests and asked to develop code; and (ii) given code and asked to develop tests.

- Question 1 asked the students to develop an implementation of a specified Student class. Three test requirements were described and a corresponding test suite provided.

- Question 2 asked the students to develop an implementation of a specified Uber class. Twenty-three test requirements were described and a corresponding test suite provided.
- Question 3 asked the students to develop test requirements and a corresponding test suite for a given Collator class (specification and code provided).

In order to reduce cognitive load, it was decided that tests should be presented using plain old Java rather than JUnit. However, to support automated testing of tests, a specific structure was designed for their specification.

```
// Test <number>
// Test purpose
System.out.println("Test <number>");
// Set up test fixture
// Perform mutation and/or observation of objects
// Check result, printing 'Pass" or 'Fail' as appropriate.

Example:
// Test 1
// Check setNames sets name and and getFullName returns name.
System.out.println("Test 1");
Student student = new Student();
student.setNames("Patricia", "Nombuyiselo", "Noah");
if (student.getFullName().equals("Patricia N. Noah")) {
System.out.println("Pass");
}
else {
System.out.println("Fail");
}
```

2.2 Assignment 3

Assignment 3 builds upon Assignment 2, aimed to have the students work on both components of TTD, being the red-green-refactor approach: they were asked to develop tests and then develop code.

- Question 1 asked the students to identify tests requirements and, subsequently, to develop a test suite for a specified JumpRecord class.
- Question 2 asked the students to construct a JumpRecord class.

Each student was given a single opportunity to submit their answer for Question 2 to the automatic marker. They were told that getting 100% for Question 1 should assure that an implementation that passed their tests would also pass the automatic marker's tests.

2.3 Testing the Tests

Question 3 of Assignment 2 (A2.3) and Question 1 of Assignment 3 (A3.1) required that the automatic marker be used to evaluate a set of tests to assess how well they discriminate between a correct implementation of the class under test and a faulty implementation.

For each question, based on the given specification, a set of test requirements was drawn up and then, from these, a correct implementation (gold standard), and a set of faulty implementations (mutations) were developed.

Mutations were developed by hand by taking each requirement in turn and identifying ways in which the correct implementation could be modified such that it would fail.

In all, 19 test requirements were identified for A2.3, and 22 test requirements were identified for A3.1. From these, 23 mutations were developed for A2.3, and 54 mutations were developed for A3.1.

The automatic marker was set up for a question such that each mutation served as the basis of a trial. A trial was conducted by: (i) compiling and running the student test suite against the given mutation; (ii) capturing the output; and (iii) searching for the presence of 'Fail'. The test suite passed a trial if it generated a 'Fail'. As a final trial, the test suite was run against the gold standard implementation of the class under test. Successful completion of the trial required, naturally, that the suite did not generate a 'Fail'. This final trial was implemented using a special penalty feature, whereby all awarded marks were deducted in the event that it was not successfully completed.

3 Observations

3.1 Automatic marker

In the normal course of affairs, an assignment requires that a student submit programs to the automatic marker. Each program is compiled once, and run may times, i.e., trials are conducted in which inputs are applied to the program and the actual output compared to the expected output. The student is informed of the outcome of each trial. Where the program fails a trial, typically, they are shown the inputs, the expected output and the actual output.

Automatic marking of a typical 1st year program usually involves about 10-20 trials. In the case of testing the tests of question 1 in assignment 3, there were 56 trials, resulting in a very large amount of feedback that was hard to process. A different approach to providing feedback on tests of tests needs to be considered.

The user experience was impacted by a decision to have the automatic marker compile mutations rather than precompile them. This made configuration easier, but at the expense of submission response times.

3.2 Mutation Development

Mutations appear to make for effective characterisation of programming problem requirements, and thus serve well as a means of assessing student tests that purport to do the same. However, the effort involved in devising mutations by hand is not trivial. While offset by the fact that materials are reusable, using this approach to roll out TDD across a whole course would be a sizable task.

One way of reducing the workload may be through a more discriminating application. For example, rather than developing mutations for all requirements, identify those that possess genuine complexity, or those that are most important in the context of the learning outcomes for the assignment.

3.3 Testing Can Only Reveal the Presence of Faults, Never Their Absence

In Assignment 3, the students were assured that, should their test suite be passed by the automatic marker, and their implementation be passed by their test suite, then their implementation would also be passed by the automatic marker. They were given one opportunity to submit their class implementation. The intention was to encourage students to focus on tests first and then code later.

A large quantity of mutations was developed to ensure that the assurance held. However, there were still some problems. A number students made a mistake in the implementation of a particular method that resulted in a double being returned instead of an integer; unfortunately, this was not picked up when testing their tests because of automatic typecasting. One of the students created an off-by-one fault in their code that should have been accounted for in the mutation set but was not.

These problems were fixed by adding two additional trials, one involving compiling the student test suite against an implementation of the class with the wrong return type, and the other by generating an additional mutation.

In future applications, it would be wise simply to limit the number of submissions to a small number, as even test harnesses are seldom perfect.

4 Student attainment

Table 1 depicts the distribution of marks attained by the students on the TDD questions.

Table 1. Student mark distribution

	A2.1	A2.2	A2.3	A3.1	A3.2
$mark = 0$	6%	8%	19%	33%	25%
$0 < mark \leq 50$	0%	0%	3%	1%	4%
$50 < mark \leq 90$	0%	6%	33%	10%	5%
$90 < mark \leq 100$	0%	0%	38%	21%	2%
$mark = 100$	94%	86%	7%	35%	64%

A mark of zero generally indicates that the student did not make a submission. Thus, for A2.1, for example, 6% of the cohort fall into this category.

Students were permitted to make multiple submissions for the questions of assignment 2 and for the first question of assignment 3. The mark spread for

A2.1 indicates that the students largely found it unproblematic. Students were able to perfect their answers. The same generally applies to A2.2.

The spreads for A2.3 and A3.1 suggest that the students found designing test requirements and tests to be challenging. A significant number appear to have accepted a 'good enough' mark once past 50%. It is possible that that they were not equipped to perfect their solutions.

It is not uncommon in an assignment for later questions to score lower, possibly due to time pressure. However, the spreads for A3.1 and A3.2 indicate that assignment 3 was challenging from the start. A high percentage did not submit a test suite for A3.1, or could not gain a mark past zero, while a relatively small percentage achieved perfection.

Due to the issues noted in section 3.3, the restriction for A3.2 was actually relaxed, and the number of permitted code submissions was raised from 1 to 5. Given that the percentage attaining a mark of zero goes down from A3.1 to A3.2, it would suggest some students simply skipped the first question.

The average number of submissions made by the students was 1 for A2.1, 2 for A2.2, 5 for A2.3 and 12 for A3.1. These figures seem to confirm that designing tests and test requirements was challenging. Also, the slight increase in the average between A2.1 and A2.2 suggests that some students did not fully utilise the supplied test suite.

5 Student Feedback

Students were asked to comment on their experiences on TDD in Assignments 2 and 3, as well as unit testing that they were exposed to in later assignments. 23 students responded to a short survey at the beginning of the following semester from a class that comprises about half the number of students enrolled in the 1st year course. In summary, the responses were as follows:

- When asked "how do you feel about having to do test-driven development", students had mixed reactions. About half were intimidated or unexcited by the prospect, some of whom felt that it was inappropriate or unnecessary for the sizes of projects. Other students, in contrast, felt that these were industry-preparation skills that were necessary and useful for programmers.
- Most students felt that using TDD took more time, but some correctly reasoned that this was because of the shift in mindset and setting up of appropriate code frameworks.
- Most students thought that TDD did not improve their code quality, with many unsure of the impact because of small projects. Only 4 students thought TDD definitely improved their code quality.
- When asked "was TDD too much unnecessary work", 14/23 indicated that it was.
- About 50% of respondents understood how TDD worked in the assignments.
- Almost all respondents understood why TDD is important.
- Students were asked how the assignments could be improved. Answers included: better instructions/teaching; less emphasis on pedantic tests; more

examples; a checklist for what could go wrong; more practice exercises; and less ambiguity.
- When asked for general comments, many students agreed that the principle of TDD was important but the implementation could be improved in various ways.

6 Discussion, Conclusions, and Future Work

The results from the early pilot study are mixed. Students appear to appreciate the importance of TDD but they are not necessarily experiencing the immediate benefits and it is perceived to be more of a hindrance than a help to them. Staff observations also suggest that more effort needs to go into the design of assessments and the design of assessment tools to support teaching using a TDD approach. Traditional automated testing environments are based on feeding input values or parameters to code and observing output/parameters. TDD may need more sophisticated or different techniques to assess tests rather than code.

Instruction was mentioned by many students as an inadequacy and may have to include a better explanation on the TDD methodology [1] and aforementioned benefits [3, 4, 6]. This also extends to the training of those who design assessment of TDD assignments. Automated assessment of programs is often considered to be a black art, where experienced teachers encode their value systems into programs to enforce those values on students. Automated assessment of tests takes this one step further and not much has been written on formalisms, guidelines or best practices for those designing the assessments.

Given the many issues and mixed results, the next step is a refinement and possibly a second pilot study. Technology enhancements, mode of teaching, timing of TDD in the curriculum and many other factors need to be determined before the next group of students encounters TDD, with hopefully a much more positive experience than that of their predecessors. It may also be of use to assess the pattern of submissions to the automatic marker, in particular on whether the undesirable "testing one's code through the automarker" also holds for submitting tests, rather than the intended red-green-refactor of TDD.

Acknowledgement. This research was partially funded by the National Research Foundation of South Africa (Grant numbers: 85470 and 88209) and University of Cape Town.

References

1. Beck, K.: Test-Driven Development by Example, Addison Wesley, Vaseem (2003).
2. Edwards, S. H.: Improving student performance by evaluating how well students test their own programs, Journal on Educational Resources in Computing (JERIC) 3(3) (2003). doi: 10.1145/1029994.1029995

3. Erdogmus, H., Morisio, T.: On the Effectiveness of Test-first Approach to Programming. IEEE Transactions on Software Engineering, 31(1) (2005). doi: 10.1109/TSE.2005.37
4. Janzen, D. S.: Software architecture improvement through test-driven development. In Companion to 20th ACM SIGPLAN Conference 2005, pp. 240–245, ACM Proceedings (2005). doi: 10.1145/1094855.1094954
5. Madeyski, L.: Test-Driven Development - An Empirical Evaluation of Agile Practice, Springer, ISBN 978-3-642-04287-4, pp. 1–245 (2010). doi: 10.1007/978-3-642-04288-1
6. Rafique, Y., Mišić, V.B. The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis, IEEE Transactions on Software Engineering, 39(6): 835–856 (2013). doi: 10.1109/TSE.2012.28
7. Suleman, H.: Automatic Marking with Sakai. In Proceedings of SAICSIT 2008, 6-8 October 2008, Wilderness, South Africa, ACM (2008). Availablehttp://pubs.cs.uct.ac.za/archive/00000465/