What vs. How: Comparing Students' Testing and Coding Skills

Colin Fidge[1] **Jim Hogan**[1] **Ray Lister**[2]

[1]School of Electrical Engineering and Computer Science,
Queensland University of Technology, Brisbane, Qld, Australia

[2]Faculty of Engineering and Information Technology,
University of Technology Sydney, Sydney, NSW, Australia

{c.fidge, j.hogan}@qut.edu.au, Raymond.Lister@uts.edu.au

Abstract

The well-known difficulties students exhibit when learning to program are often characterised as either difficulties in understanding the problem to be solved or difficulties in devising and coding a computational solution. It would therefore be helpful to understand which of these gives students the greatest trouble. Unit testing is a mainstay of large-scale software development and maintenance. A unit test suite serves not only for acceptance testing, but is also a form of requirements specification, as exemplified by agile programming methodologies in which the tests are developed *before* the corresponding program code. In order to better understand students' conceptual difficulties with programming, we conducted a series of experiments in which students were required to write both unit tests and program code for non-trivial problems. Their code and tests were then assessed separately for correctness and 'coverage', respectively. The results allowed us to directly compare students' abilities to characterise a computational problem, as a unit test suite, and develop a corresponding solution, as executable code. Since understanding a problem is a pre-requisite to solving it, we expected students' unit testing skills to be a strong predictor of their ability to successfully implement the corresponding program. Instead, however, we found that students' testing abilities lag well behind their coding skills.

Keywords: Learning to program; unit testing; object-oriented programming; program specification

1 Introduction

Failure and attrition rates in tertiary programming subjects are notoriously high. Many reasons have been suggested for poor performance on programming assignments, including an inability to fully understand the problem to be solved and an inability to express a solution in the target programming language.

To help determine whether students do poorly on programming tasks due to an incomplete understanding of the problem or an inability to write a solution, we conducted a series of experiments in which these two aspects of programming were evaluated separately. Students in second and third-year programming classes were given assessable assignments which required them to:

1. unambiguously and completely characterise the problem to be solved, expressing the requirements as a unit test suite; and then
2. produce a fully-functional computational solution to the problem, expressed as an object-oriented program.

Both parts of the assignments carried equal weight. By directly comparing how well each student performed on each of these tasks we aimed to see if there was a clear relationship between students' ability to say *what* must be done versus their ability to say *how* to do it.

Since understanding a problem is a necessary prerequisite to solving it, our expectation was that students would inevitably need to do well on problem definition before succeeding in coding a solution. We therefore designed assignments which allowed us to produce independent marks for code functionality and test coverage, enabling students' skills in these two distinct stages of large-scale object-oriented programming to be contrasted.

2 Related and previous work

Our overall goal is to help expose the underlying reasons for students' difficulties in learning how to develop program code. In particular, we aimed to distinguish students' abilities to both describe and solve the same computational problem, using 'test-first' programming as a basis. We did this with classes of students who had already completed two previous programming subjects, so they had advanced beyond the basic problems of developing imperative program code. As a clearly-defined test-first programming paradigm we used 'test-driven development' (Beck 2003), in which unit tests are developed first and the program code is then extended and refactored in order to pass the tests.

The academic role of unit testing in general, and test-first programming in particular, has already received considerable attention. For instance, an early study by Barriocanal et al. (2002) attempted to integrate unit testing into a first-year programming subject. They made unit testing optional to assess its take-up rate and found that only around 10% of students voluntarily adopted the approach (although those who did so reported that they liked it).

In another early study, Edwards (2004) advocated the introduction of testing into student assignments as a way of preventing them from following an ad hoc, trial-and-error coding process. He describes a marking tool called Web-CAT which assesses both program code correctness and unit test coverage. (We do the same thing, but developed our own UNIX scripts for this purpose.) However, where we kept students' code correctness and test coverage marks separate, so that we could perform a correlation analysis on them, Edwards (2004) combined the marks to produce a single composite score for return to the students.

In a more recent study of unit testing for first-year students, Whalley & Philpott (2011) assessed the advantages of supplying unit tests to students. (We do likewise in our first-year programming subject, giving students unit tests but not expecting them to write their own.) Using a questionnaire they concluded that although it was generally beneficial for students to apply the 'test-early' principle, a minority of students still struggle to understand the concept.

In an earlier study, Melnick & Maurer (2005) surveyed students' perceptions of agile programming practices, and found that students failed to see the benefits of testing and believed that it requires too much work. In our experience, there is no doubt that test-first programming requires considerable discipline from the programmer and can be more time-consuming than 'test-last' programming. Nevertheless, we have found that this extra effort is compensated for by a better quality product, as have many other academics (Desai et al. 2008).

Given students' reluctance to put effort into unit testing, Spacco & Pugh (2006) argued that testing must be taught throughout the curriculum and that students must be encouraged to "test early". Like us they did this by assessing students' test coverage and by keeping some unit tests used in marking from the students. (In fact, in our experiments we did not provide the students with any unit tests at all. Instead we gave them an Application Programming Interface to satisfy, so that they had to develop all the unit tests themselves.)

Keefe et al. (2006) aimed to introduce not just test-driven development into first-year programming, but also other 'extreme' programming concepts such as pair programming and refactoring. (These principles are also introduced in our overall curriculum, but not all during first year.) Their survey of students found a unanimous dislike of test-driven development, and they concluded that students find it a "difficult" concept to fully appreciate. (In our course we introduce students to unit testing in first year, but don't get them to write their own test suites until the second and third-year subject described herein. At all levels, however, our experience is also that there is considerable resistance to the concept from novice programmers.)

Like many other researchers, Janzen & Saiedian (2007) found that mature programmers were more likely to see the benefits of test-driven development. Given a choice they found that novice programmers picked test-last programming in general. In a subsequent study they also found that students using test-first programming produced more unit tests, and hence better test coverage, than those using test-last programming, but that students nevertheless still needed to be coerced to follow test-first programming (Janzen & Saiedian 2008). (Both of these findings are entirely consistent with our experiences. Although our own academic staff can clearly see the benefits of test-first programming, our students must be coerced into adopting it.)

Most recently, using attitudinal surveys, Buffardi & Edwards (2012) again found that students did not readily accept test-driven development, despite the fact that those students who followed the approach produced higher quality code. (They report that they taught test-driven development but did not assess it directly. By contrast we directly assessed our students' unit testing skills.)

Thus, while there have been many relevant studies, none has presented a direct comparison of unit testing and program coding skills as we do below. Furthermore, while we focussed on students with some programming experience, most previous studies have considered first-year students only.

3 Programming versus unit testing

In traditional "waterfall" programming methodologies, program code is developed first, followed by a set of acceptance tests. Modern "agile" software development approaches reverse this sequence (Schuh 2005). Here the tests are written first, in order to define what the program code is required to do, and then corresponding program code is developed which passes the tests. In the most extreme form, *test-driven development* involves iteratively writing individual unit tests immediately followed by extending and refactoring the corresponding program code to pass the new test (Beck 2003). This is usually done in object-oriented programming, where the "unit" of testing is one or more methods. Advantages claimed for this approach to software development include the fact that a "working" version of the system is available at all times (even if all the desired functionality has not yet been implemented), that it minimises administrative overheads, and that it responds rapidly to changing customer requirements.

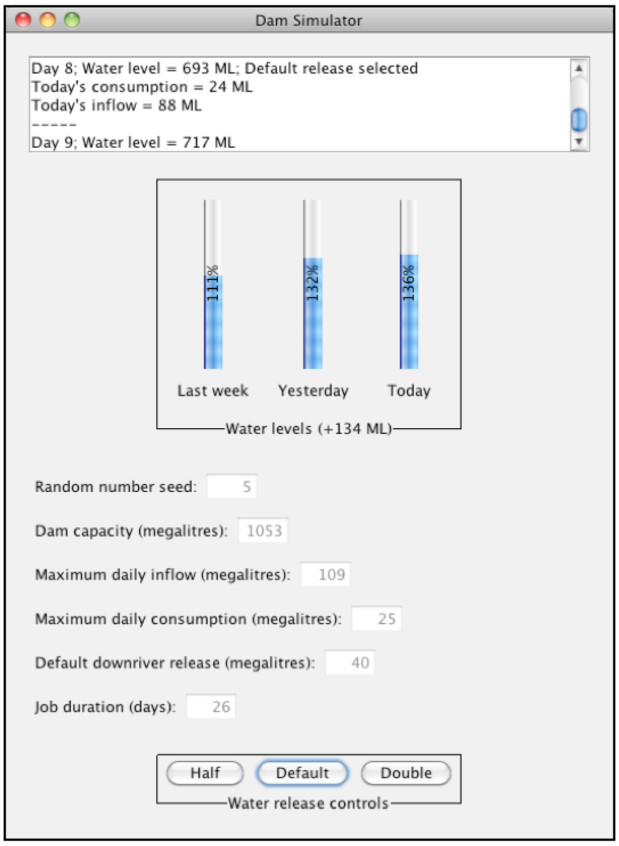
Most importantly for our purposes, a suite of unit tests can be viewed as a *specification* of what the corresponding program is required to do. It defines, via concrete examples, what output or effect each method in the program must produce in response to specific inputs. For each method in the program, a well-designed unit test suite will include several representative examples of 'typical' cases, which validate the method's 'normal' functionality, as well as 'extreme' or 'boundary' cases, which confirm the method's robustness in unusual situations (Schach 2005, Astels 2003). Overall, a unit test suite must provide good *coverage* of the various scenarios the individual methods are expected to encounter. (However, unit testing does not consider how separate program modules work together, so it is usually followed by *integration* testing.)

Test-driven software development thus produces two distinct artefacts, a unit test suite and corresponding program code. The first defines *what* needs to be done and the second describes *how* this requirement is achieved computationally. Having taught object-oriented programming and test-driven development for several years, we realised that these two artefacts can be assessed separately, giving independent insights into students' programming practices.

4 The experiments

To explore the relationship between students' testing (specification) and coding (programming) ability, we conducted a series of four experiments over two semesters. This was done in the context of a *Software Development* subject for second and third-year IT students. (The classes also contained a handful of postgraduate students, but too few to have a significant bearing on the results.) The students enrolled have typically completed two previous programming subjects, meaning that they have already advanced beyond simple imperative coding skills and are now concerned with large-scale, object-oriented programming.

The *Software Development* subject focuses on tools and techniques for large-scale program development and long-term code maintenance. Topics covered include version control, interfacing to databases, software metrics, Application Programming Interfaces, refactoring, automated builds, etc. In particu-



```
Day 8; Water level = 693 ML; Default release selected
Today's consumption = 24 ML
Today's inflow = 88 ML
-----
Day 9; Water level = 717 ML
```

Random number seed: 5

Dam capacity (megalitres): 1053

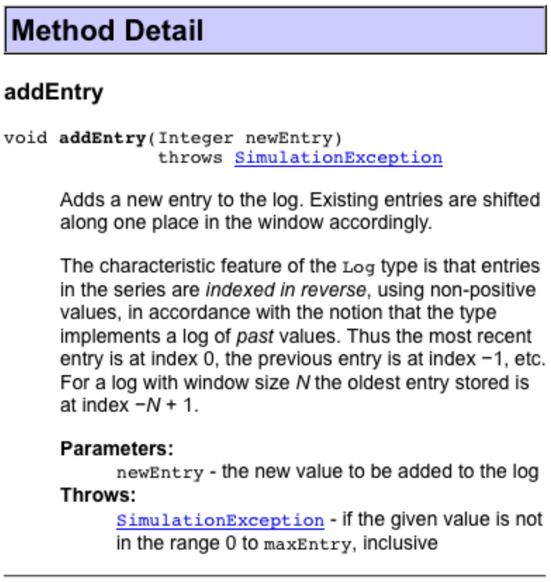
Maximum daily inflow (megalitres): 109

Maximum daily consumption (megalitres): 25

Default downriver release (megalitres): 40

Job duration (days): 26

Figure 1: User interface for the solution to Assignment 2a



Method Detail

addEntry

```
void addEntry(Integer newEntry)
         throws SimulationException
```

Adds a new entry to the log. Existing entries are shifted along one place in the window accordingly.

The characteristic feature of the `Log` type is that entries in the series are *indexed in reverse*, using non-positive values, in accordance with the notion that the type implements a log of *past* values. Thus the most recent entry is at index 0, the previous entry is at index −1, etc. For a log with window size *N* the oldest entry stored is at index −*N* + 1.

Parameters:
`newEntry` - the new value to be added to the log

Throws:
`SimulationException` - if the given value is not in the range 0 to `maxEntry`, inclusive

Figure 2: Example Javadoc specification of a method to be implemented in Assignment 2a

lar, the role of unit testing is introduced early and is used throughout the semester. Test-driven development is also introduced early as a consistent methodology for creating unit test suites. The illustrative programming language for the subject is Java, although the concepts of interest are not Java-specific.

Each semester's assessment includes two non-trivial programming assignments. Both involve developing a unit test suite and a corresponding object-oriented program, each worth approximately equal marks. The first assignment is individual and the second is larger and conducted in pairs. In the first assignment the students are given a front-end Graphical User Interface and must develop the back-end classes needed to support it. In the second assignment the student pairs are required to develop both the GUI and the back-end code.

For example, one of the individual assignments involved developing classes to complete the implementation of a 'Dam Simulator' which models the actions involved in controlling water levels in a dam. (This topical assignment was introduced shortly after the January 2011 Brisbane floods, which were exacerbated by the overflow of the Wivenhoe dam.) The simulator models the effects on water levels of randomly-generated inflows and user-controlled outflows over a period of time. The user acts as the dam's operator and can choose how much water to release into the downstream spillway each day. The simulation ends if the dam overflows or runs dry.

(This assignment, and the 'Warehouse Simulator' described below, are examples of 'optimal stopping' problems, in which the challenge is to optimise the value of a certain variable, such as a dam's water level, despite one or more influencing factors, such as rainfall, being out of the user's control (Ferguson 2010). We find that these problems make excellent assignment topics because they can form the basis of game-like simulations which are popular with the students.)

The students were given the code for the user interface (Figure 1), minus the back-end calculations. (As per civil engineering convention, the dam's 'normal' water level is half of its capacity, which is why the meters in Figure 1 go to 200%.) To complete the simulator they were required to develop two classes and their corresponding unit tests. One class is used to keep a daily log of water levels in the dam and the other implements the effects of the user's inputs.

The necessary classes and methods to be implemented were defined via Java 'interface' classes, described in standard 'Javadoc' style. For instance, Figure 2 contains an extract from this specification, showing the requirements for one of the Java methods that must be implemented for the assignment. In this case the method adds a new entry to the log of daily water levels. It must check the validity of its given arguments, add the new entry to the (finite) log, and keep track of the number of log entries made to date. Our anticipated solution is the Java method shown in Figure 3.

Each of these Java functions must be accompanied by a corresponding suite of 'JUnit' tests (Link 2003), to ensure that the method has been implemented correctly and to document its required functionality. For instance, Figure 4 shows one such unit test which confirms that this method correctly maintains the number of log entries made to date. The test does this by instantiating a new log object, adding a fixed number of entries into it, and asserting at each step that the number of entries added to the log so far equals the number of entries reported by the log itself. In general there will be several such unit tests for each method implemented—it is typically the case that a comprehensive unit test suite will be much larger than the program code itself. In our own solution we had seven distinct unit tests like the one in Figure 4 to fully define the required properties of the method in Figure 3.

Since the deliverables for these assignments included both unit tests and code, we determined to exploit the opportunity to directly compare how well students performed on unit testing and coding. We considered only those parts of the assignments where both unit tests and code must be produced. (The GUI code in the two pair-programming assignments

```
public void addEntry(Integer newValue) throws SimulationException {
    // Sanity checks
    if (newValue < 0)
        throw new SimulationException("New log entry " + newValue + " too small");
    if (newValue > maxEntry)
        throw new SimulationException("New log entry " + newValue + " too big");
    // Remove oldest entry, if window is full
    if (waterLog.size() == windowSize)
        waterLog.remove(windowSize - 1);
    // Add new entry and keep count
    waterLog.add(0, newValue);
    numberOfEntries += 1;
}
```

Figure 3: Example of a Java method to be implemented in Assignment 2a

```
@Test(timeout = maxWait)
public void NumEntriesIsCorrect() throws SimulationException {
    testLog = new WaterLog(windowSizeSmall, maxEntryLarge);
    // Add entries up to, but not exceeding, the window size
    for (Integer numEntered = 1; numEntered <= windowSizeSmall; numEntered++) {
        // Add an entry
        testLog.addEntry(smallEntryOne);
        // Confirm that number entered equals number reported by the log
        assertEquals(numEntered, testLog.numEntries());
    }
}
```

Figure 4: Example of a JUnit test for the method in Figure 3

Table 1: Statistics for the four assignments

	Assignment			
	1a	1b	2a	2b
Number of classes to implement (excluding GUIs)	2	10	2	6
Number of methods to implement (excluding GUIs)	16	25	12	26
Number of unit tests in our 'ideal' solution	66	63	45	53
Number of assignments submitted and marked	170	83	217	113

was not accompanied by unit tests and was marked manually. Nor did we consider marks awarded for 'code quality' in the experiment.) At the time the experiment was conducted we had four assignments' worth of results to analyse, two from each semester (Table 1). In all four cases the students were required to implement both program code and unit tests and it was emphasised that these two parts were worth equal marks.

- **Assignment 1a:** First semester, individual assignment. This assignment involved completing the back-end code for a 'Warehouse Simulator' which models stock levels in a warehouse. A Graphical User Interface was supplied and students needed to complete a 'ledger' class, to track expenditure, and a 'transactions' class, to implement user-controlled stock buying and randomly-generated customer supply actions.

- **Assignment 1b:** First semester, paired assignment. This assignment involved developing the GUI and back-end code for a 'Container Ship Management System' which models loading and unloading of cargo containers on the deck of a ship, subject to certain safety and capacity constraints. Classes were needed for different container types (refrigerated, dry goods, hazardous, etc) and for maintaining the ship's manifest. (There were a large number of classes and methods in this assignment, but most were trivial subclasses just containing a constructor and one or two additional methods.)

- **Assignment 2a:** Second semester, individual assignment. This was the 'Dam Simulator' assignment described above.

- **Assignment 2b:** Second semester, paired assignment. This assignment involved developing the GUI and back-end code for a 'Departing Train Management System' which modelled the assembly and boarding of a long-distance passenger train. Classes were needed to model the shunting of individual items of rolling stock (including an engine, passenger cars, freight cars, etc) to assemble a train and then simulate boarding of passengers.

In each case the students were expected to follow the test-driven development discipline. For each functional requirement they were expected to develop a JUnit test (such as that in Figure 4) and then extend and refactor their Java program code (like that in Figure 3) until all the system requirements were satisfied. During individual Assignments 1a and 2a the lone student was expected to alternate roles as 'tester' and 'coder'. Pair-programming Assignments 1b and 2b allowed each student to adopt one of these roles at a time.

UNIX shell test scripts were developed to automatically mark the submitted assignments. This was done in two stages, to separately assess the students' program code and unit test suites.

- **Code functionality:** The students' classes were compiled together with our own 'ideal' unit test suite. (As explained below, this often exposed students' failures to match the specified API.) Our unit tests were then executed to determine how well the students had implemented the required functionality in their program code. The proportion of tests passed was used to calculate a 'code functionality' mark and a report was generated automatically for feedback to the students.
- **Test coverage:** For each of our own unit tests we developed a corresponding 'broken' program which exhibited the flaw being tested for. To assess the students' unit test suite against these programs, their tests were first applied to our own 'ideal' solution program to provide a benchmark for the number of tests passed on a correct solution. Then the students' unit tests were applied to each of our broken programs. If fewer tests were passed than the benchmark our marking script interpreted this to mean that the students' unit tests had detected the bug in the program. (This process is not infallible since it can't tell *which* of the students' tests failed. Nevertheless, we have found over several semesters that it gives a good, broad assessment of the quality of the students' unit test suites.) The proportion of bugs found was used to calculate a 'test coverage' mark and a feedback report was generated automatically.

These marking scripts needed to be quite elaborate to cater for the complexity of the assignments and to allow for various problems caused by students failing to follow the assignments' instructions. The marking scripts ultimately consisted of well over 400 lines of (commented) Unix bash code (excluding our own ideal solution and the broken programs needed to assess the students' tests).

A particularly exasperating problem encountered during the marking process was the failure of a large proportion of students to accurately implement the specified Application Programming Interface and file formats, typically due to misspelling the names of classes and methods, failing to throw required exceptions, adding unexpected public attributes, or using the wrong types for numeric parameters. This was despite the teaching staff repeatedly emphasising the need to precisely match the specified API as an important aspect of professional software development. In particular, during marking of Assignment 2a it was found that fully half of the submissions failed to match the API specification, and therefore could not be compiled and assessed. In order to salvage some marks for these defective assignments, those that could be easily corrected by, for example, changing the class and method signatures, were fixed manually. In the end 97 assignments, which accounted for 45% of all submissions, were corrected manually and re-marked. Penalities were applied proportional to the extent of correction needed. Ultimately, however, this large amount of effort produced little difference since the re-marking penalties sometimes outweighed the additional marks gained!

Another practical issue noted by Spacco & Pugh (2006), and confirmed by our own experiences, is the difficulty of developing unit tests that uniquely identify a program bug. In practice a single program bug is likely to cause multiple tests to fail. We found when setting up our automatic marking environment, including our own unit test suite (for assessing the students' code) and the suite of 'broken' programs (for

Table 2: Measures of central tendency, spread and correlation for code functionality and test coverage

	Assignment			
	1a	**1b**	**2a**	**2b**
Code functionality mean	82	84	79	91
Code functionality median	86	93	86	96
Code functionality standard deviation	20	20	21	12
Test coverage mean	69	79	45	63
Test coverage median	67	92	52	69
Test coverage standard deviation	23	25	29	24
Functionality versus coverage correlation	0.64	0.76	0.70	0.55

assessing the students' unit tests), that it was impossible to achieve a precise one-to-one correspondence between code bugs and unit tests. While frustrating for us, this did not invalidate the marking process, however.

Moreover, one of the risks associated with this kind of study is threats to 'construct validity' (Arisholm & Sjøberg 2004), i.e., the danger that the outcomes are sensitive to different choices of code functionality and test coverage measures. Nevertheless, we believe our analysis is quite robust in this regard because each of the individual tests in our 'ideal' unit test suite was directly developed from a specific functional requirement clearly described in the Javadoc API specifications, and each of the broken programs was introduced to ensure that a particular unit test was exercised. This very close functional relationship between requirements, code features and unit tests left little scope for arbitrary choices of unit tests or broken programs against which to assess the students' assignments.

5 Experimental results

The marks awarded for code functionality and test coverage were normalised to percentages and are summarised in Table 2. (These values exclude marks awarded for 'code presentation' and for the front-end GUIs in Assignments 1b and 2b.) It is obvious that the average marks for test coverage are well below those for code functionality, which immediately casts doubt on our assumption that students' unit testing abilities would be comparable to their programming skills. Furthermore, while a standard correlation measure (last row of Table 2) shows some correlation between the two sets of marks, it is not very strong.

Further insight can be gained by plotting both sets of marks together as in Figures 5 to 8. Here we have sorted the pairs of marks firstly by code functionality, in blue, followed by test coverage, in red. Assignments 1a and 1b used a coarser marking scheme than was used for Assignments 2a and 2b, thereby accounting for their charts' 'chunkier' appearance, but the overall pattern is essentially the same in all four cases. At each level of achievement for code functionality there is a wide range of results for test coverage. This pattern is especially pronounced in Figure 7.

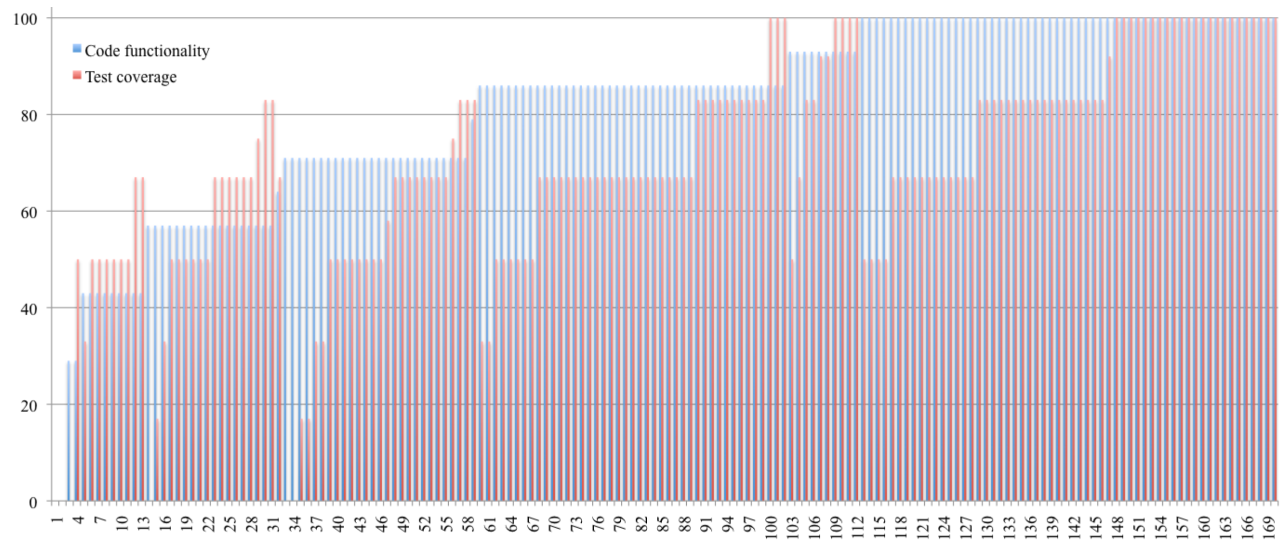


Figure 5: Comparison of marks for code functionality and test coverage, Assignment 1a, 170 submissions

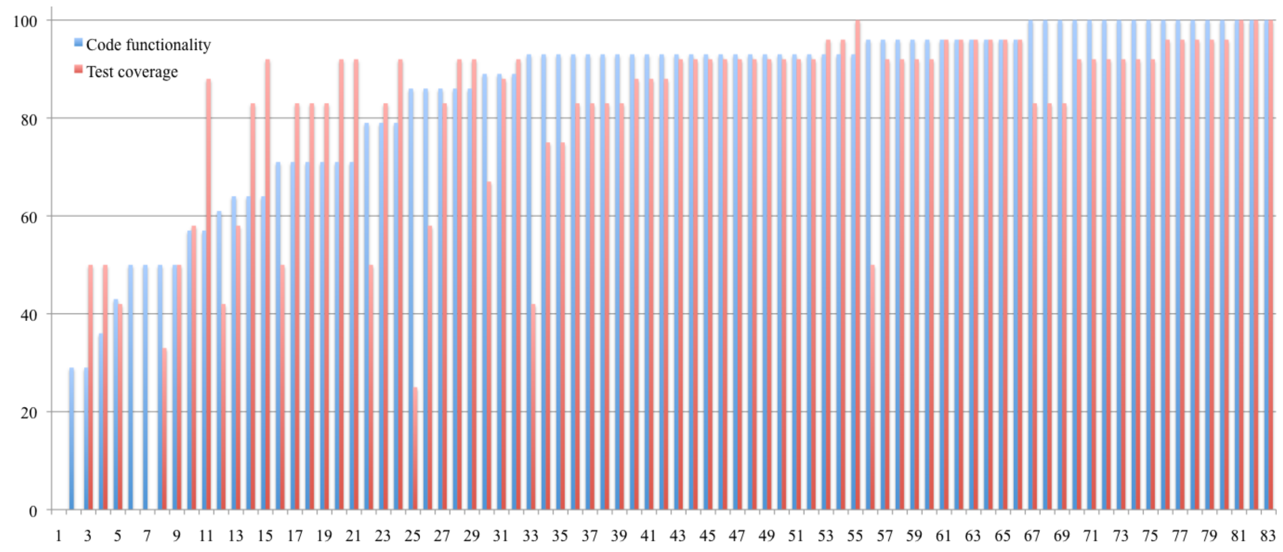


Figure 6: Comparison of marks for code functionality and test coverage, Assignment 1b, 83 submissions

Recall that Assignments 1a and 2a were individual, so we can assume that the same person developed both the unit tests and program code. How pairs of students divided their workload in Assignments 1b and 2b is difficult to say based purely on the submitted artefacts, although if they followed the assignments' instructions they would have swapped 'tester' and 'coder' roles regularly. Figures 6 and 8 are for the pair programming assignments and both show an improvement in the test coverage marks, compared to the preceding individual assignments in Figures 5 and 7. Regardless of how they divided up the task, this suggests that the students took the unit testing parts of the assignment more seriously in their second assessment. Nevertheless, the test coverage results still lag well behind those for code functionality throughout.

Inspection of the submitted assignments suggests that many students *didn't* follow the test-driven development process which, in practice, can be time consuming and requires considerable discipline. Often students developed their program code and their unit tests separately, rather than letting the latter motivate the former. However, even if students did not follow the test-driven development process, this does not invalidate our comparision of their unit testing and program coding abilities, as these are two distinct skills.

In all four sets of results there are numerous examples of students scoring well for code functionality but very poorly for test coverage, meaning that they could implement a solution to a problem that they couldn't (or merely chose not to) characterise in the form of a unit test suite, the exact opposite of what we would expect if they had strictly followed the test-driven development process.

To explore this phenomenon further we conducted a conventional correlation analysis (Griffiths et al. 1998) to see if there was a clear relationship between students' testing and coding skills, irrespective of their absolute marks for each. We began by checking the normality of all the marks' distributions. This was done by inspecting quantile-quantile plots generated by the *qqnorm* function from the statistics package *R* (R Core Team 2012). These plots (not shown) provide good evidence of normality for seven of the eight data sets, albeit with some overrepresentation of high scores in the coding result distributions. The quantile-quantile plots were skewed somewhat by discretisation of the marks; evidence for normality was especially strong for Assignments 2a and 2b, for which we had the most fine-grained marks available.

The marks were then used to create correlation scatterplots (Griffiths et al. 1998), comparing stan-

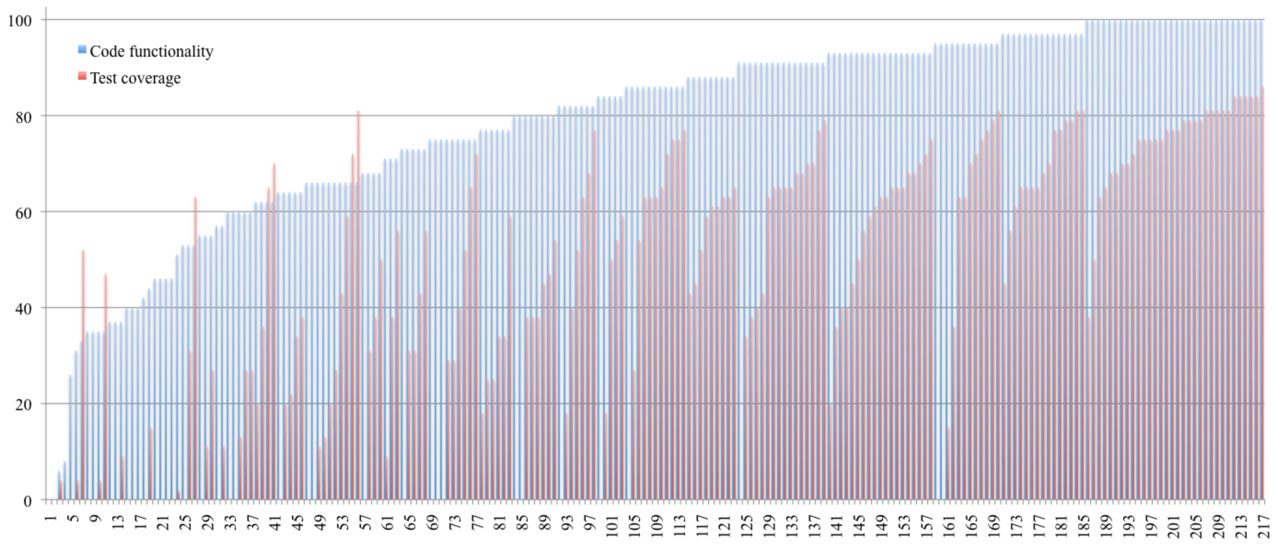


Figure 7: Comparison of marks for code functionality and test coverage, Assignment 2a, 217 submissions

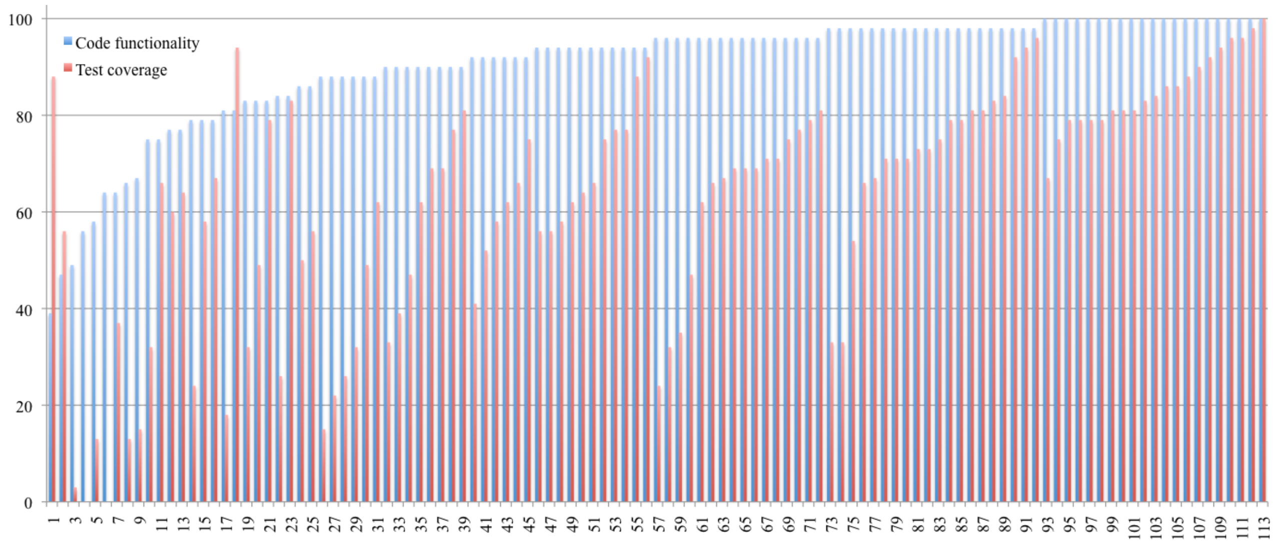


Figure 8: Comparison of marks for code functionality and test coverage, Assignment 2b, 113 submissions

dardised marks for each of the four assignments, as shown in Figures 9 to 12. As usual, dots appearing in the top-right and bottom-left quadrants suggest that there is a positive linear relationship between the variables of interest, in this case students' code functionality and test coverage results. (In the interests of clarity a handful of extreme outliers, resulting from some students receiving near-zero marks for both tests and code, have been omitted from the figures.)

All four figures exhibit good evidence of a linear relationship, especially due to the numbers of students who did well on both measures (top-right quadrant). Overall, though, the pattern is not as strong as we expected.

Undoubtedly many students put less effort into the unit tests, despite them being worth equal marks to the program code. It is clear from the averages in Table 2 and the charts in Figures 7 and 8 that even the best students did not do as well on the unit tests as the program code, and it certainly wasn't the case that successfully defining the test cases to be passed was a pre-requisite to implementing a solution as we had originally assumed. Even more obvious is the line of dots along the bottom of Figure 11 which was caused by students who received zero marks for test coverage. Evidently these students learned the importance of the unit tests for their overall grade by the time they did their second assignment because no such pattern is evident in Figure 12.

Overall, therefore, contrary to our expectations, computer programming students' performance at expressing *what* needs to be done proved to be a poor predictor of their ability to define *how* to do it.

6 Conclusion

The students' poor performance in unit testing compared to program coding surprised us. Nonetheless, the pattern of results in Figures 5 to 8 is remarkably consistent. On the left of each chart are a few examples of students who could characterise the problem to be solved but couldn't complete a solution (i.e., their test coverage results in red are better than their code functionality results in blue), which is what we would expect to see if the students applied test-driven development. However, this was far outweighed in each experiment by the dominance of results in which students produced a good program but achieved poor test coverage (i.e., their blue code functionality results were better than their red test coverage results). We thus have clear empirical evidence that students struggle to a greater extent defining test suites than

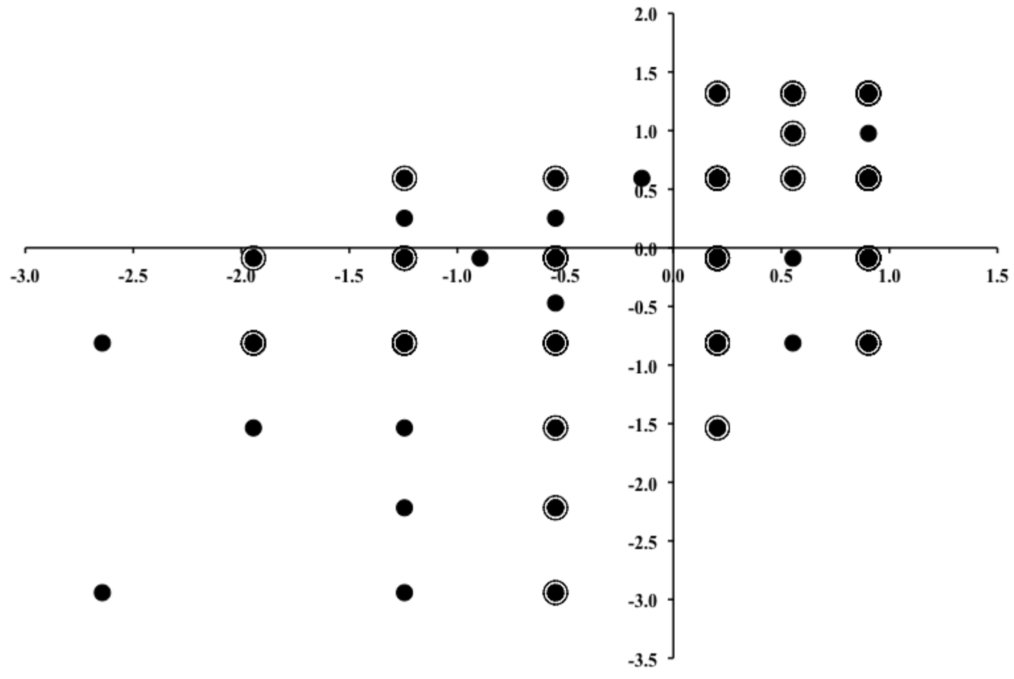


Figure 9: Correlation scatterplot, code functionality (x-axis) versus test coverage (y-axis), with circled dots denoting multiple data points with the same value, Assignment 1a

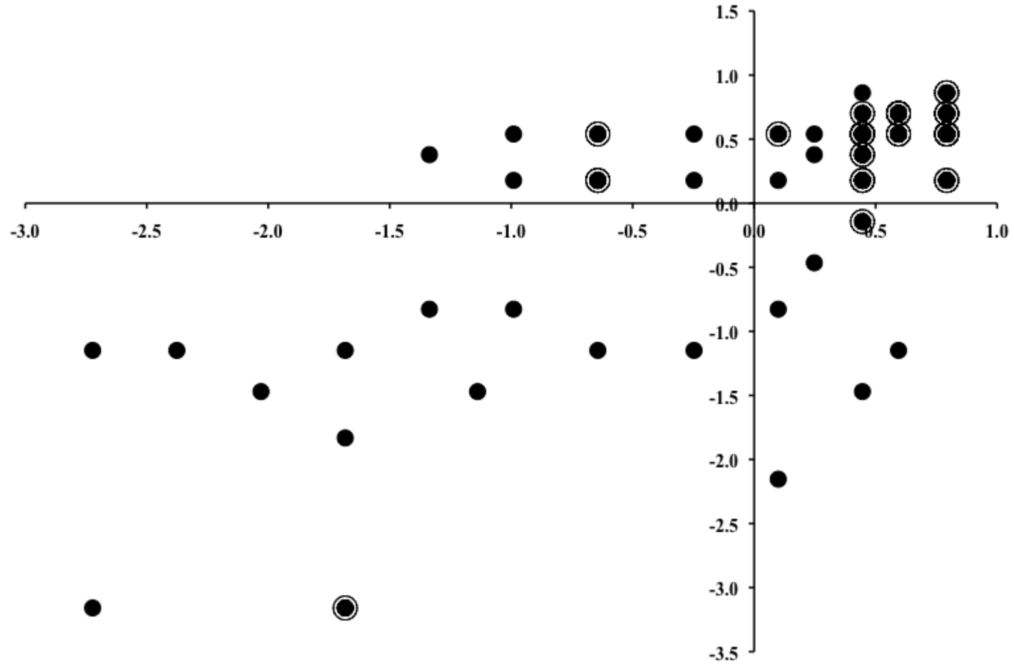


Figure 10: Correlation scatterplot, Assignment 1b

they do implementing solutions, for the same programming problem.

The outstanding question from this study is whether this unexpected result reflects students' abilities or motivations. From our inspection of the submitted assignments we can make the following observations.

- The results can be explained in part by many students' obvious apathy towards the unit testing part of the assignments. Despite their regular exposure to the principles of test-driven development in class, and despite being well aware that half of their marks for the assignment were for their tests, it was clear in many cases that students did not follow the necessary discipline and instead wrote the program code first, seeing it as more "important". Their unit tests were then completed hastily just before the assignment's deadline. As noted in Section 2 above, this student behaviour has been observed in many prior studies. For instance, Buffardi & Edwards (2012) found that students procrastinate when it comes to unit testing, even when a test-first programming paradigm is advocated, which typically leads to poor test coverage in the submitted assignments.

- Another partial explanation is simply that many students had poor unit testing skills. Test-driven development emphasises the construction of a large number of small and independent tests,

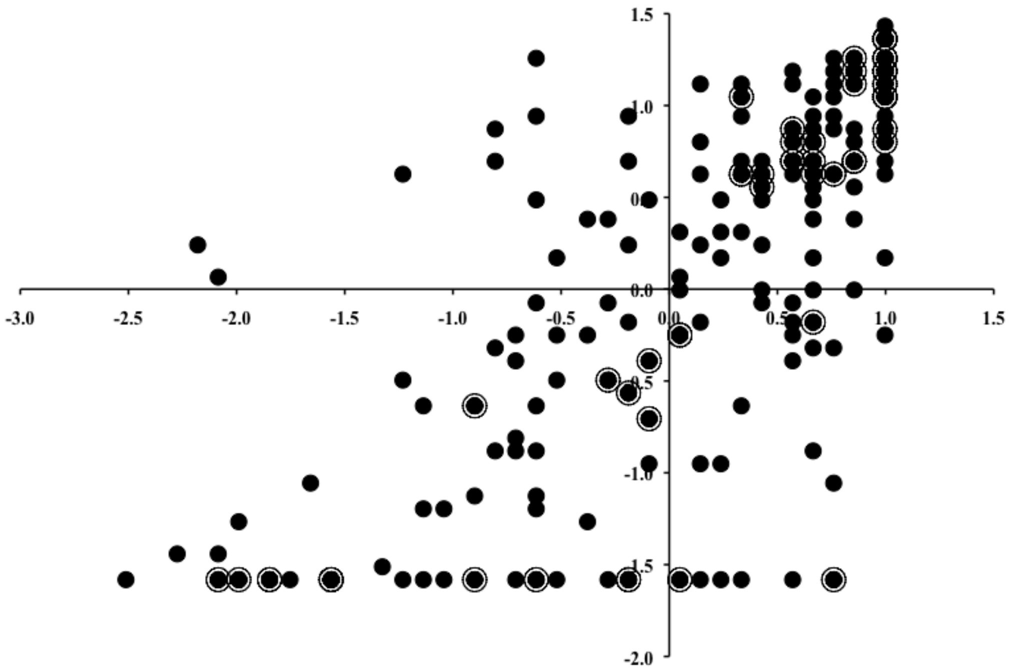


Figure 11: Correlation scatterplot, Assignment 2a

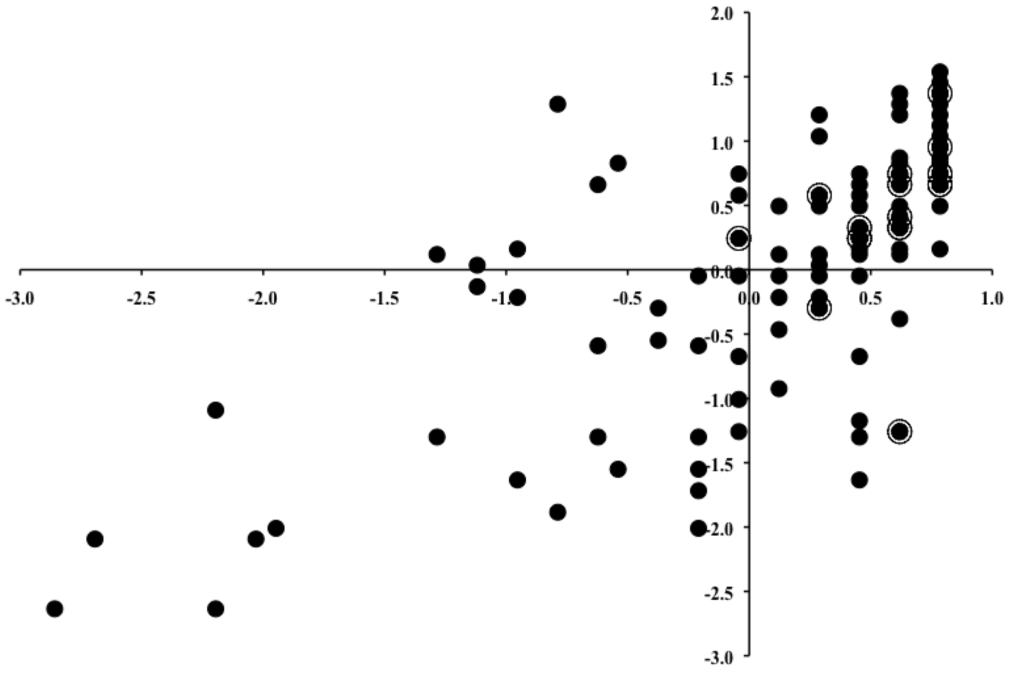


Figure 12: Correlation scatterplot, Assignment 2b

each highlighting a distinct requirement. However, inspection of some assignments with low testing scores showed that they consisted of a small number of large and complicated tests, each attempting to do several things at once. This is evidence that the students could not (or chose not to) follow the test-driven development discipline. Small sets of complex unit tests are characteristic of test-last programming and typically produce poor test coverage because several distinct coding errors may all cause the same test to fail, without the reason for the failure being obvious. At the other extreme there were also a few examples of students producing a very large number of tests, often over twice as many as in our own solution, but still achieving poor coverage because their tests did not check distinct problems and so many were redundant. This undirected, ‘shotgun’ strategy is again evidence of a failure to apply, or understand, test-driven development, which avoids redundancy by only introducing tests that expose new bugs.

- It is also noteworthy that writing unit tests is a cognitively more abstract activity than writing the corresponding program code, thus making it more challenging for students still coming to grips with the fundamentals of programming. Whether or not this was the cause of students’ poor test coverage marks is impossible to tell from the submitted artefacts alone. Anecdotally, our discussions with students while they were working on the assignments left us with the impression that they understood the principles of unit testing well enough. By far the most common question asked by students while they were working on their assignment was "How many unit tests should I produce?" rather than "How do I write a unit test?"

Ultimately, therefore, further research is still required. Although we have demonstrated that programming students' testing and coding skills can be analysed separately, and that they consistently perform better at coding than testing, more work is required to conclusively explain why this is so.

Acknowledgements

We wish to thank Dr Andrew Craik for developing the assignment marking scripts used in the first two experiments, and the anonymous reviewers for their many helpful suggestions for improving the correlation analysis. Support for this project was provided by the Office of Learning and Teaching, an initiative of the Australian Government Department of Industry, Innovation, Science, Research and Tertiary Education. The views expressed in this publication do not necessarily reflect the views of the Office of Learning and Teaching or the Australian Government.

References

Arisholm, E. & Sjøberg, D. (2004), 'Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software', *IEEE Transactions on Software Engineering* **30**(8), 521–534.

Astels, D. (2003), *Test-Driven Development: A Practical Guide*, Prentice-Hall.

Barriocanal, E., Urbán, M.-A., Cuevas, I. & Pérez, P. (2002), 'An experience in integrating automated unit testing practices in an introductory programming course', *SIGCSE Bulletin* **34**(4), 125–128.

Beck, K. (2003), *Test-Driven Development: By Example*, Addison-Wesley.

Buffardi, K. & Edwards, S. (2012), Exploring influences on student adherence to test-driven development, *in* T. Lapidot, J. Gal-Ezer, M. Caspersen & O. Hazzan, eds, 'Proceedings of the Seventeenth Conference on Innovation Technology in Computer Science Education (ITiCSE'12), Israel, July 3–5', ACM, pp. 105–110.

Desai, C., Janzen, D. & Savage, K. (2008), 'A survey of evidence for test-driven development in academia', *SIGCSE Bulletin* **40**(2), 97–101.

Edwards, S. (2004), Using software testing to move students from trial-and-error to reflection-in-action, *in* D. Joyce, D. Knox, W. Dann & T. Naps, eds, 'Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04), USA, March 3–7', ACM, pp. 26–30.

Ferguson, T. S. (2010), 'Optimal stopping and applications'. `http://www.math.ucla.edu/~tom/Stopping/Contents`.

Griffiths, D., Stirling, W. D. & Weldon, K. L. (1998), *Understanding Data: Principles and Practice of Statistics*, Wiley.

Janzen, D. & Saiedian, H. (2007), A leveled examination of test-driven development acceptance, *in* 'Proceedings of the 29th International Conference on Software Engineering (ICSE'07), USA, May 20–26', IEEE Computer Society, pp. 719–722.

Janzen, D. & Saiedian, H. (2008), Test-driven learning in early programming courses, *in* S. Fitzgerald & M. Guzdial, eds, 'Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'08), USA, March 12–15', ACM, pp. 532–536.

Keefe, K., Sheard, J. & Dick, M. (2006), Adopting XP practices for teaching object oriented programming, *in* D. Tolhurst & S. Mann, eds, 'Proceedings of the Eighth Australasian Computing Education Conference (ACE2006), Hobart', Vol. 52 of *Conferences in Research in Practice in Information Technology*, Australian Computer Society, pp. 91–100.

Link, J. (2003), *Unit Testing in Java*, Morgan Kaufmann.

Melnick, G. & Maurer, F. (2005), A cross-program investigation of students perceptions of agile methods, *in* 'Proceedings of the 27th International Conference on Software Engineering (ICSE05), USA, May 15-21', ACM, pp. 481–488.

R Core Team (2012), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.
URL: *http://www.R-project.org*

Schach, S. (2005), *Object-Oriented and Classical Software Engineering*, McGraw-Hill, USA. Sixth edition.

Schuh, P. (2005), *Integrating Agile Development in the Real World*, Thomson.

Spacco, J. & Pugh, W. (2006), Helping students appreciate Test-Driven Development (TDD), *in* W. Cook, R. Biddle & R. Gabriel, eds, 'Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA06), USA, October 22-26', ACM, pp. 907–913.

Whalley, J. & Philpott, A. (2011), A unit testing approach to building novice programmers skills and confidence, *in* J. Hamer & M. de Raadt, eds, 'Proceedings of the Thirteenth Australasian Computer Education Conference (ACE 2011), Perth', Vol. 114 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society, pp. 113–118.