Formative Test-Driven Development for Programming Practicals[*]

Peadar F Grant

Dundalk Institute of Technology

Abstract

Sub-optimal performance in computer programming practicals may be associated with a lack of contemporaneous formative feedback in the laboratory session. Although automated testing is a key feature of industrial software development, its routine use in computer programming education remains uncommon. Additionally, written or verbal specifications of desired functionality can lack precision, leading to uncertainty amongst students regarding what is expected in a particular exercise. Literature reports of automated testing centre mainly on student-written tests as an end product and on summative grading systems utilising automated testing after assignment submission.

This study therefore examines the effect of adopting a signature pedagogy (Shulman, 2005) of test-driven development that utilizes formative automated testing in applied programming laboratories. Laboratory sessions were re-designed to incorporate automated formative feedback that combined lecturer-supplied test cases with industry-standard software testing frameworks. The approach was informed by best practices in formative feedback (Nicol and MacFarlane-Dick, 2006). Furthermore, technology choices minimised student effort whilst remaining tolerant of student-specific technology choices. Tools for staff use were additionally developed to tabulate and aggregate test results to guide subsequent sessions.

A mixed-method evaluation combines qualitative learner views and practitioner reflections, supported by ongoing test performance collection. The approach discussed is shown to provide improved certainty of completion and correctness. Student feedback particularly noted the easy penalty-free access to formative feedback within familiar programming environments. Utilization of industry-standard tools and a common project layout promoted and focused student discussion with peers and staff. It is concluded that a test-driven signature pedagogy utilizing lecturer-composed formative test cases could improve outcomes in advanced applied programming courses.

Keywords

Automated Formative Feedback, Signature Pedagogy, Automated Testing, Assessing Computer Programming, Test-Driven Development

* URL: http://ojs.aishe.org/index.php/aishe-j/article/view/302

1. Introduction

Computer programming, commonly called coding, forms a key component of contemporary computing education. Coding is the process by which a human expresses instructions in a form that a computer can carry out, or execute. These instructions are written in one of a number of programming languages, often Java and Python at introductory undergraduate level (Mason, 2012) A piece of software known as a compiler converts the human-readable instructions into machine-language instructions that the computer can execute.

Practical classes, designed primarily to develop fundamental and applied coding skills, are a key delivery method used in computing courses (Azemi, 1995). Within the Irish Institutes of Technology, it is also common for academic staff to deliver practical classes without the aid of teaching assistants (Lalor, 2010). A key benefit of the lab environment in general is the immediate feedback that can be delivered by staff making a quick visual inspection of the work in progress. However, when teaching computer programming, visual code reviews are of more limited use, because a given piece of code can take different paths when the computer actually executes it. Against this backdrop, there can be insufficient time for a lecturer to critique code manually and in sufficient depth, particulary when students experience difficulties.

Over several years at Dundalk Institute of Technology, the author observed a number of undesired emergent symptoms when delivering modules that involved programming. Many students were failing to complete the exercises in full before leaving the laboratory. They then regularly arrived to subsequent sessions without working code from previous sessions. There was little evidence that students checked whether their code could be compiled, that is, understood by the computer, until they had it written in full. Similarly, they did not routinely test that their written programs performed correctly when executed. When incremental development did occur, previously working code often had faults, or bugs, introduced as students worked on additional functionality. Specifications supplied to students were often vague, causing diversion of attention towards discretionary extras, such as user interface, whilst core functionality remained weak. For many students, the ultimate arbiter of success seemed to be whether the code compiled without error, and ran without crashing, rather than its functional correctness.

This final observation interestingly suggests a possible improvement to the learning environment for programming. That is, the author observed that students seemed to react positively to simple binary feedback denoting success or failure. Furthermore, positive engagement with repeatable, automated, penalty-free automated feedback has been previously reported (Nicol, 2007). Could specifications for programming tasks be encoded in such a way that the student can check their work continuously as they progress, without lecturer intervention? Within the Irish Institutes of Technology, in particular, the capacity issues and resulting constraints (Lalor, 2010) are unlikely to change in the near future (Clarke, 2015). Therefore, any means to increase in-lab feedback needs to be workload negative or at least neutral for it to be sustainable (Brown, 2004), which effectively mandates automation and self-administration.

Up until two decades ago, similar problems regarding correctness were mirrored in the software industry. The so-called waterfall development model (Royce,1970) where testing is a separate process undertaken once coding is complete, had similar issues of mismatch between delivered code and requirements (Kennedy,1998) Responding to these deficiencies, agile development practices (Beck, 2001, Beck, 2004, Humble, 2011) have gained in popularity. These methodologies heavily leverage ongoing user feedback whilst the software is being written (Beck, 1999). Automated testing tools are now central to the development process to provide feedback to coders, not just to avoid bugs being introduced when additional functionality is added, but more interestingly to encode required functionality prior to development, (Beck, 2001). Software tools are now available for almost all commonly used programming languages to simplify the process of testing, such as the JUnit framework for Java (Tahchiev et al., 2011).

Despite its clear industrial benefits, the use of automated testing as a feedback tool has not universally spread to the classroom. Reported experiences of automated test usage in the classroom setting (Devedžić and Milenković, 2011, Carlson, 2008, Wick, 2005, Olan, 2003, Girard and Wellington, 2006) do mirror the perceived positive impact by industrial user (Beck, 2001, Beck,1999, Beck, 2004). However, most case studies examine introductory programming labs, and reports of routine automated testing in advanced applied programming modules, such as those dealing with web frameworks, parallel programming or database interaction, is almost non-existent. There remains a space to add to the body knowledge specifically looking at modules where programming is used rather than taught for its own sake. From the lecturer's perspective, laboratory teaching could be greatly enhanced by being able to monitor completion and to identify areas causing difficulties for the student group.

The aim of this study was, therefore, to determine the effect on student learning of adopting test-driven development automated testing to deliver formative feedback within applied programming laboratories.

Contextual considerations guided a number of a-priori assumptions. The student cohort available for this study were Year 3 students on the BSc (Hons) Computing programme who chose the Web Frameworks elective module. Reasonable competency of programming using the Java language, which is introduced in the first two years of the programme, could therefore be assumed. Many students bring their own laptops to college, and there is substantial diversity of operating systems and development environments used for Java programming by learners. As this work was undertaken as part of the author's studies towards the Master of Arts in Learning and Teaching, the study was time-bound to one semester. The remainder of the paper is structured as follows, Section 2 reviews literature relating to feedback, both traditional and automated, and briefly examines its use within computer programming.

Section 3 describes the development and implementation of a classroom intervention to deliver formative automated feedback on programming work. Section 4 evaluates the intervention's effect, describing a qualitative-leaning mixed-mode study with quantitative support, where its influence on student achievement and solicitation of feedback in laboratory sessions are investigated.

2. Literature Review

2.1 Formative feedback

The primary prerequisite to any educational endeavour is a well-designed set of desired learning outcomes (Biggs,2003). Attainment of the desired learning outcomes is ultimately summatively tested by tasks satisfying the varied criteria of validity (Brown, 2004). However, successful self-regulation of a student's own learning essentially depends on formative feedback being returned and acted upon (Nicol,2006). Gibbs (2010) directly identifies the provision of feedback as a distinct skill that a lecturer must develop. McCabe and O'Connor (2013) similarly charge the lecturer to regularly deliver feedback of sufficient quality and quantity to enable self-regulation of learning.

The constitution of good feedback eludes simple definition (Nicol and MacFarlane-Dick, 2006), however numerous works offer opinions on its necessary attributes. The seminal piece by Nicol (2006), concisely identifies seven principles necessary for the provision of high-quality feedback, justifying each from theory. Chickering (1987) places an aligned practical lens on good practices in undergraduate education. These views are supported by Evans (2013), a review that explores many aspects in greater depth and specificity.

However, Gibbs (2004), asserts most succinctly the basic requirement that: “Sufficient feedback is provided, both often enough and in enough detail".

The essence of ideal feedback presents as a duplex dialogue between the learner and teacher. Gibbs and Simpson (2004), note a general decline in learner-specific feedback, citing the formative tutorial systems of Oxford and Cambridge as exemplars of good practice, where face-to-face dialogue coalesces around regular written work. As a two way process, the feedback dialogue requires proactive learner effort, not just reactive or passive reception (Nicol and MacFarlane-Dick, 2006). This requirement suggests an emotional dimension to feedback, and the consequential need to empathize with learners' evoked emotional responses to feedback (Dowden et al, 2013). Gibbs and Simpson (2004) echoes this concern, cautioning that feedback should concern attributes of the delivered work, not the student delivering it.

Almost unanimously, the literature recognizes that much delivered feedback accompanies summative assessments. Accordingly, the primary concern of the student is recognized as the achieved numerical grade (Cooper, 2000, Gibbs and Simpson, 2004). Indeed, Nicol and MacFarlane-Dick, (2006), asserts that the very existence of a numerical score negatively impacts on feedback efficacy. Higgins (2001) amplifies this criticism, arguing that the resultant consumerist view favours surface rather than deep learning approaches. Yet, Nicol and MacFarlane-Dick (2006), also states that feedback must help to close the gap between actual and ideal performance. Later work showed that multiple low-stakes summative assessments provide a good formative effect (Nicol,2007)}, but advise pre-drafted feedback of sufficient quality and a setting promoting further dialogue are provided if following this approach. More generally, to positively impact learning, a numerical score must be accompanied by specific diagnoses and actionable recommendations (Gibbs and Simpson, 2004).

Furthermore, an opportunity must be given for the feedback to be actively put to use, (Cooper, 2000, Gibbs and Simpson, 2004). This can often take the form of penalty-free opportunities to repeat (Nicol, 2007)}, particularly when automated electronic feedback is supplied. Daku, (2009), similarly reports positive outcomes from this approach in hardware-based practical laboratories. Finally, the initial “feed forward" must specify basic expectations before work is commenced (Chickering and Gamson,1987).

Assuming that a dialogue is set up with a high-quality exchange of actionable information, how often should it occur? The literature unanimously agrees that feedback must be routine (Brown,2004, Gibbs and Simpson, 2004) and delivered promptly following submission (Nicol, 2007, Chickering and Gamson,1987). Brown (2004) asserts that the feedback must be directly integral to the learning process. Not only are education outcomes improved, but the additional burden on academic staff can be kept to a minimum, which is a key consideration in the constrained Irish third-level sector (Lalor, 2010).

From our brief examination, feedback provision must be interactive, provide actionable recommendations, and be a routine continuous activity that improves objective performance. These ideals of feedback are not at all confined to the education space. The software industry leapfrogs the educational space here, as the value of rapid and high-quality feedback within the software process was seized upon and incorporated into standard practice nearly two decades ago.

2.2 Industry perspective

By the late 1990's, many deficiencies of educational feedback practice were mirrored in industrial settings. Traditional software engineering followed a strict pipeline named the ``waterfall cycle'' almost 40 years ago (Royce,1970). Superfically a logical choice, the well-documented disadvantage of this approach is that no customer feedback is available until the project is technically complete, by which time modifications are costly and time-consuming to implement (Kennedy,1998). The person who wrote the code is not even responsible for testing their code at the point of writing in some cases.

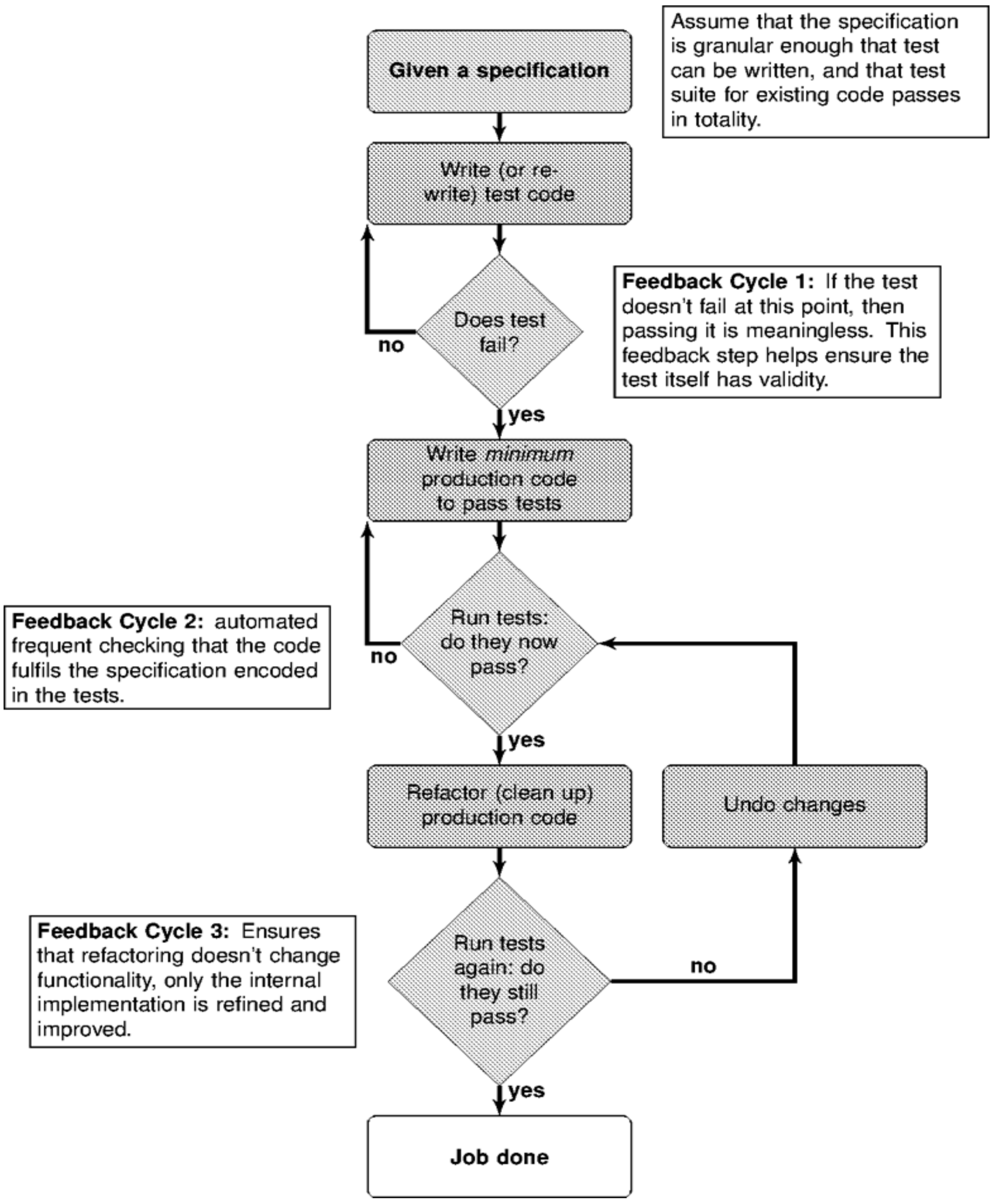
In 2001, a self-selected group of software developers coalesced to publish a contrarian view, the so-called *Agile Manifesto* (Beck et al, 2001), which leveraged feedback to improve product quality.

Instead of a waterfall cycle, short iterations culminating in demonstration of new functionality to the customer ensured that feedback was received regularly and acted upon (Beck and Andres, 2004, Gibbs and Simpson, 2004). Even more radically, a customer or user representative was seconded permanently to the development team, opening a rich and continuous dialogue.

Implementation of the Agile Manifesto's aspirations primarily fell upon new iterative process methodologies, aided by a number of specific working practices and technical interventions (Beck and Andres, 2004). Many complex and time-consuming manual processes were neutered by a deep commitment to automation, succinctly put as a key maxim: “If It Hurts, Do It More Frequently, and Bring the Pain Forward” (Humble and Farely, 2011). Perhaps the most significant impact of the Agile Manifesto was on the process of testing.

Previously, testing was a manual, laborious and time-consuming endeavour performed once all code was written. An overriding preference for automated testing of code was espoused by the agile methodology, so that the code could in essence test itself once written. This meant that testing could now take place as a regular and repeatable activity (Beck and Andres, 2004), providing routine feedback. More fundamentally, testing moved both temporally and conceptually from its validation role following development to an integral cyclical feedback component of writing code, coining the moniker of Test Driven Development (TDD) as a key cornerstone of today's software industry (Beck, 2001). TDD inverts the role of automated testing, placing it at the heart of a feedback-driven development workflow. The key change was that automated tests describing new functionality were written before the code to implement that functionality (Beck, 2001). Given a particular new functional requirement, tests are written in a machine-readable format, which serve as an explicit checklist of functionality that we use the computer to check has been completed. A typical TDD workflow contains three feedback loops. Firstly, the developer or another team member writes a test from a functional specification. The first feedback loop is that the programmer verifies that this test fails, since no production code has been written to satisfy it. The minimum amount of production code is then written to satisfy the test, which is run repeatedly by the programmer to verify their work, forming the second feedback cycle Finally, once a working solution has been arrived at, it may need a certain amount of refactoring, or internal reorganisation. The same test that verified the new functionality ensures that no errors enter during refactoring, forming the third feedback cycle.

Most programming languages have a defacto testing framework, such as JUnit for the Java language (Tahchiev, 2011), that enables basic pieces of code to be tested. More advanced testing scenarios, such as simulating user interaction with graphical user interfaces or web applications, can be scaffolded on to these frameworks.


Given a specification
Assume that the specification is granular enough that test can be written, and that test suite for existing code passes in totality.
Write (or re-write) test code
Does test fail?
no
yes
Feedback Cycle 1: If the test doesn't fail at this point, then passing it is meaningless. This feedback step helps ensure the test itself has validity.
Write *minimum* production code to pass tests
Feedback Cycle 2: automated frequent checking that the code fulfils the specification encoded in the tests.
Run tests: do they now pass?
no
yes
Refactor (clean up) production code
Undo changes
Feedback Cycle 3: Ensures that refactoring doesn't change functionality, only the internal implementation is refined and improved.
Run tests again: do they still pass?
no
yes
Job done

2.3 Classroom adoption

The documented disadvantages of waterfall development (Kennedy,1998) mirror problematic experiences from educational programming laboratories. (Shulman,2005), introduces the concept of a signature pedagogy, where elements of professional practice are mirrored in the learning methods employed in the classroom, such as the clinical round in medicine or the debate in law. In software development, similarities exist between industry and the classroom. The lecturer parallels the customer or specifier role (Beck et al, 2001, Beck and Andres, 2004), setting requirements and being available during the lab session to clarify expectations and offer coaching. However, there is limited evidence of universal adoption of automated testing in the classroom, despite its widespread industrial application. There is a confluence of technical and educational literature around the potential benefits of formative automated testing tools that provide penalty-free self-administered feedback during development work (Nicol, 2006, Cooper,2000) which suggests that their incorporation may further complete the signature pedagogy of computer programming. Both Christensen (2003) and Jones (2001) suggest that automated testing of this nature should pervade computing education at programme rather than module-level. Desai et al (2008) catalogues a number of approaches used to date, likening automated testing more to a programming technique than an assessment method.

Automated testing has been reported in numerous guises within computing education as Douce et al, 2005catalogues. Multiple direct references to TDD practices within the classroom exist, with Barriocanal et al (2002), one of the earliest accounts, providing an organizational blueprint. Further reassurance is offered by Girard and Wellington (2006), and Wick (2005), who report positive outcomes of a test-first approach at both introductory and more advanced level modules. Desai et al, 2009, similarly offers sensible guidance for TDD in the class setting, noting that some reordering of certain introductory topics benefit TDD, and crucially also reports no significant increase in student workload as a result of this approach.

Edwards (2003) crucially identifies three conditions for testing to have an effect on student's overall learning, testing must be integral to the programming activity, frequent useful feedback is required and the testing activity must be seen to have value. In practice, these three conditions reflect the key principles of good feedback practice (Nicol, 2006). Expanding on this, (Edwards,2004), cites the introduction of automated testing as enabling a move “from trial-and-error to reflection in action". Successful regular use of automated testing promotes students to refactor, or clean up, their code once working (Carlson,2008).

TDD will unavoidably require specialist supporting software tools to be employed. It is essential, however, that the tools used do not cognitively overload students (Reis and Cartwright, 2004), because testing will then simply be neglected (Humble and Farely, 2011). Particular difficulties are unavoidable when testing work that involves graphical user interfaces (Thornton et al,2008), and indeed simple keyboard-driven interactive programs (Proulx and Rasala, 2004), as Douce et al, (2005), acknowledge.

The literature includes numerous accounts of development in the area of centralized and later web-based online summative grading systems that could test submitted code to varying degrees (Hitchner,1999, Spacco et al (2006), Edwards,2003, Higgins et al, 2005). In this vein, the Web-CAT system described by Edwards (2003), emphasized the combination of instructor and learner-written automated tests. For formative work, it would be preferable to have testing tools that do not require time-consuming steps of submitting code to an online service (Clarke et al, 2014).

The three mainstream Java Development environments (Eclipse, Netbeans and IntelliJ) all provide support for running automated tests and viewing the results. In addition, a number of pedagogically focused Java development environments have been released, that support testing to varying degrees, such as DrJava (Allen et al, 2002) and Java Programming Laboratory (JPL) (Pullan et al, 2013). Interestingly, JPL was designed to promote mastery of certain threshold programming concepts through composition of a large number of small, targeted pieces of code, and included a relatively painless testing and submission system. Whilst pedagogically focused development tools no doubt provide value, they are inescapably contradictory to the idea that the signature pedagogy should be authentic to practice (Shulman, 2005). It could therefore be argued that for mid-level courses involving students who are soon entering the workplace that pedagogy-specific tools should be eschewed in favour of industrially utilized tools.

The innate feelings of students regarding TDD as a feedback method must also be considered, since it should foster positive motivational beliefs (Nicol and MacFarlane-Dick, 2006), and be used in a way that is emotionally sensitive (Dowden et al, 2013). Analysis by Janzen and Saiedian, (2007), found that positive experiences of automated testing in educational settings led to continued usage. Similarly, an account of broad introduction of agile software development practices in an undergraduate course directly noted an excessive use of testing amongst student concerns Devedžić and Milenković, 2011). In particular, the critical analysis by Kollanus and Isomöttönen (2008), points to cognitive overloading when applying TDD in its most orthodox form.

Whilst it is clear that various forms of test-driven development have been reported to be successful, their use has been confined to introductory level courses, and often required non-standard development software. There is scope for further examination of the effects of adopting TDD within the signature pedagogy used in more applied programming settings, such as signal processing or web development, whilst retaining its automated nature promoting frequent usage.

1. Intervention Methodology

Test-driven development was adopted as a signature pedagogy (Shulman, 2005) for the *Web Frameworks* module, noting the need for authenticity to industrial practice. This extends the existing natural parallels between the lab environment and agile development methodologies, namely short iterations and customer-presence. Given that the identified problem was one of feedback, the seminal seven principles of quality feedback proposed by Nicol and MacFarlane-Dick, (2006), were used to guide the intervention's design process. Secondly, the constraint was imposed that any technologies adopted must be industry vernacular (Shulman, 2005). Additionally, the diverse choices of operating systems (Windows, Mac, GNU/Linux) and integrated development environments (Netbeans, Eclipse, IntelliJ) used by students to write and run Java code must continue to be respected. For these reasons, the approach taken involved minimal technological intervention and avoided classroom-specific tools

3.1 Laboratory exercise design

Each lab session began with a statement of the learning goals for the session to align the laboratory activity (Biggs, 2003), following a brief retrospective of the previous session. Required theory was presented, often including demonstration of finished-product or intermediate functionality. Students then completed a lab exercise where the specifications were supplied in both natural language and as automated tests. They could use the automated tests supplied to gain immediate feedback during the session on how their code performed in response to various valid and erroneous inputs.

Feedback must first “clarify what good performance is” (Nicol and MacFarlane-Dick, 2006). Noting the observation by (Sadler,1989) that requirements can be ambiguous and poorly articulated, the first design decision is that all principal functional requirements will be articulated in automated tests. Those automated tests will be supplied by the lecturer in addition to the written exercise brief. This directly mirrors industrial practice (Beck and Andres, 2004, Beck,2001), and previous similar interventions (Wick et al, 2005).Test cases will not only exercise “positive” functionality but will check how code deals with erroneous or ambiguous inputs (Olan,2003). In contrast to some previously reported studies (Edwards, 2003), the tests are in no way hidden from students, who can review the test's Java code in the normal way. A typical test case for code that produces factorials is presented in Figure 2.

```
 package  ie.dkit.webframeworks.unittestinglab;

 import  org.junit.After;
import  static  org.junit.Assert.assertEquals; import  org.junit.Before;
 import  org.junit.Test;
     public     class     FactorialCalculatorTest     {
     FactorialCalculator  calculator;
     @Before
public void setup() {
calculator = new FactorialCalculator();
     }
     @After
public void tearDown() { calculator = null;
     }
     // test a "normal" usage case
@Test
public   void   computesPositiveFactorial()   throws   Exception   {   assertEquals(6,
calculator.factorial(3));
     }
     // test an edge case
@Test
public    void    computesFactorialZero()    throws    Exception    {    assertEquals(1,
calculator.factorial(0));
     }
     // test that an erroneous input is caught and rejected
@Test(expected=ArithmeticException.class)
public void throwsExceptionOnNegativeFactorial() { calculator.factorial(-1);

     }
 }
```

Figure 2: Sample test case

The second principle requires good feedback to promote self assessment and reflection on the learner's part (Nicol and MacFarlane-Dick, 2006). This implies that the suite of tests be available to learners and easy to run on a regular basis, echoing the recommendation that feedback be routine (Brown, 2004, Gibbs and Simpson, 2004), and delivered promptly (Nicol, 2007). Therefore, students received objective progress measure by running tests regularly during the practical session using the testing features of their standard development environment. Providing the tests and feedback using the standard development environment avoided additional work on the part of the student, ensuring that they would regularly run the tests while working, in contrast to previously reported automatic grading solutions where students submitted code to a separate online service (Edwards,2003).

Nicol and MacFarlane-Dick's, (2006) third principle requires feedback to deliver information of sufficient quality to learners. By adopting the standard JUnit test framework, the industrially-employed Integrated Development Environments' abilities to show test results in a colour-coded easy-to-understand pass/fail format are leveraged fully. Furthermore, each test examined a single aspect of functionality (Tahchiev,2011), allowing easy identification of a failing test's possible cause. A failing test therefore directly lead to so-called “reflection-in-action” (Edwards,2004).

Fourthly, and critically, feedback should encourage a learner to enter dialogue with their peers and instructors around learning (Nicol and MacFarlane-Dick, 2006). To faciliate focused discussion, tests were individually well-named and contextualised (Gibbs and Simpson, , 2004, Dowden et al, 2013).

Furthermore, each student started from a common project template, as with the previously described JPL system (Pullan et al, 2013), providing an identical context and layout enabling discussion without having to review a peer's setup.

In this implementation, a common project template was distributed using the industry-standard Git version control system, giving the same benefits. Additionally, this project template used the industry-standard Maven Java project management tool (Miller et al, 2010), that directly and transparently integrates with all main Java development environments, thus ensuring that students did not need to install or configure test frameworks manually.

Fifth, feedback must also bolster a learner's motivational beliefs, and their self-esteem ((Nicol and MacFarlane-Dick, 2006). With the approach taken, a student ideally leaves the classroom knowing that they have passed a given number of tests, and ideally that no tests remain as failing. Test names and assertion failure messages were framed positively (Dowden et al, 2013). By using the Git version control system and the Maven project builder system, difficulties relating to setting up the test environment were avoided, thus avoiding any negative perceptions on the part of the student.

Sixth, feedback must facilitate closing the gap between actual and desired performance ((Nicol and MacFarlane-Dick, 2006). Tests were therefore written with sufficient granularity and hints in comments to allow students to meaningfully focus effort on failing tests (Gibbs and Simpson, 2004, Cooper, 2000).

The seventh and final principle is concerned with feedback of information to the teacher and is dealt with separately in Section 3.2.

Undeniably, the use of automated testing introduces new and unfamiliar software tools and attendant technical challenges (Kollanus and Isomöttönen, 2008). Before starting the lab series, a practical tutorial session on automated testing was provided, to the extent that it was used in the module. This included how the JUnit system deployed worked (Tahchiev), and why we were using it from a pedagogical perspective (Wick et al, 2005). The software industry context (Beck,2001) was explained as the rationale for choosing this signature pedagogy (Shulman,2005). In particular, the ability for testing to clarify expectations and help close the gap ((Nicol and MacFarlane-Dick, 2006) was discussed. An integrated lab exercise gave students the opportunity to set up the test framework for themselves and to experience using it on a functionally-trivial example.

3.2 Performance analysis

The teacher's input to the feedback process is heavily dependent on the 7th principle of good feedback practice, namely the availability of information that shapes teaching ((Nicol and MacFarlane-Dick, 2006). Looked at more closely, principles 4 and 6 depend on this if the teacher is going to have any useful role in the process.

In-class participation in the module accounted for 20% of the grade. For laboratory work, the participation grade was assigned for the submission of the test report files to the relevant Assignment on Moodle. The approach taken was similar to (Pullan et al, 2013), in replacing manual reporting (Clarke et al, 2014) with semi-automated reporting that required little student effort.

Each time that a learner runs the tests, JUnit produces a report file, which is in the XML machine-readable structured data format. This report lists the specific tests passed and failed, and is easy to analyse programatically.

The reports were downloaded in bulk from Moodle immediately after each lab session. Custom scripting to analyse these files was developed by the author using Python with the Pandas data analysis library (McKinney,2010). Output for a simulated group of report files is presented in Figure 3.

```
n_registered = 15
#submitted = 15

TESTS FOUND:
T_1: DefaultPropertyReaderTest.readsPropertyCorrectly[0: site.title=Cou
T_2: DefaultPropertyReaderTest.readsPropertyCorrectly[1: site.subtitle=
T_3: DefaultPropertyReaderTest.readsPropertyCorrectly[2: site.country=I
T_4: DefaultPropertyReaderTest.readsPropertyCorrectly[3: contact.number
T_5: PropertyReaderTest.readsPropertyCorrectly[0: site.title=Course web
T_6: PropertyReaderTest.readsPropertyCorrectly[1: site.subtitle=Dundalk
T_7: PropertyReaderTest.readsPropertyCorrectly[2: site.country=Ireland]
T_8: PropertyReaderTest.readsPropertyCorrectly[3: contact.number=null]

STUDENT-TEST RESULT BREAKDOWN
     T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8
11c7  P   P   P   P   P   P   P   P
2259  P   P   P   P   P   P   P   P
2c50  P   P   P   P   P   P   P   P
3821  P   P   P   P   P   P   P   P
3f22  P   P   S   P   P   F   P   P
64a4  P   P   P   P   P   P   P   P
8313  P   P   P   P   P   P   P   P
9e25  P   P   P   P   P   P   P   P
b88c  P   P   P   P   P   P   P   P
bb87  P   P   P   P   P   P   P   P
cdef  P   P   P   P   P   P   P   P
cefa  P   P   P   P   P   P   P   P
d664  P   P   P   P   P   P   P   P
eaca  P   P   P   P   P   P   P   P
ff82  P   P   P   P   P   P   P   P

Key: P=PASS, F=FAIL, E=ERROR, S=SKIPPED
SUMMARY:
submission rate: 100%
mean pass rate: 98%
100pc pass rate: 93%
```

Figure 3: Performance tracking script output

These reports were used to inform in-class activities and provide targeted assistance, in a face-to-face analogue of the approach that (Pullan, 2013) suggests. Additionally, the following summary statistics were generated for each lab exercise:

> **Submission rate** -- the proportion of students who attempted the lab, demonstrated by making a submission at or before the end of the lab. This measures basic engagement.
>
> **Mean pass rate** -- the mean of the percentage of tests passing for all students who submitted data for the particular lab exercise. This measures the overall attainment of the learning goals.
>
> **All pass** -- the percentage of students who achieved all tests passing during the laboratory session.

2. Evaluation Methodology

The effect of adopting TDD as a signature pedagogy on learning was evaluated primarily in a qualitative fashion informed in part by quantitative performance data. The main summative qualitative data collection tools for consideration here include interviews, focus groups and surveys (Cohen et al ,2007). Surveys were discounted immediately due to concerns over the small sample size. Whilst personal interviews have the potential for a rich dialogue between the researcher and individual participant, they have the potential to become intrusive and to present scheduling difficulties (Cohen et al ,2007).

The focus group, by its nature, removes the personal spotlight from the individual participant, and crucially allows group discussions to develop (Cohen et al, 2007, McLafferty,2004). For these reasons, the focus group was selected as the primary qualitative input of the learner experience. Focus groups sessions were scheduled prior to the end of term, recognising the possible impact of competing personal commitments on turnout (McLafferty,2004).

Ethical protections required close consideration given that participants were also students taking the module for credit. Institutional ethical policies were fully complied with. Activities were appropriately sequenced to ensure direct participant input occurred only after a significant proportion of the available grade had been returned Cohen et al, 2007). At the beginning of the module, students were fully informed on the relation of the research activity to the module delivery. They were free to contact the author or other named individuals within the institution if they had any concerns.

2.1 Focus group

Design and conduct of the focus group sessions was closely informed by Krueger (2002) McLafferty (2004) and Kidd (2000). Although both Krueger (2002,) Cohen (2007) suggest more than one group, it is felt here that since the sample size is small and there is *a priori* commonality amongst the participants that a single group will suffice.

The session was time bound to a strict maximum of one hour. Light refreshments were provided at the beginning of the session to create a warm and relaxing atmosphere (McLafferty,2004). The room was chosen and arranged to promote discussion, utilizing circular seating (Krueger,2002). Questions were designed to stimulate discussion and allow additional probing, avoiding serendipitous dead ends (Krueger,2002). Appendix A provides the adopted question outline.

Audio recording was used to capture the output of the focus group whilst allowing free-flowing discussion. The faciltator additionally kept contemporaneous notes (Krueger,2002).

Particular ethical safeguards were employed regarding the running of focus groups. Students had already received 50% of the possible marks prior to the focus group session. An information session was held that refreshed the overall study aims and summarised the research progress to date. An invitation was issued to participate in a focus group. Full details on the areas to be covered and session conduct were provided. Learners received hardcopy information sheets and participant consent forms to take home and peruse at their leisure, in compliance with the institutional ethics policies. Participants were free to withdraw from the focus group at any time before or during the session without giving a reason, and were reminded of this in the information session and on the day. Additionally, students were reminded that they could contact the researcher or other named individuals at any time for clarification or to express concerns.

Due to conflicting commitments on the scheduled day, the first focus group attracted a single participant, who expressed a wish to proceed in any event, it is separately reported under Section 5.2 as an in-depth interview. A number of potential participants who were unable to make the first focus group made contact with the facilitator and a second session was held, which is is reported under Section 5.1. Analyses were conducted separately for both sessions, with commonalities and differences identified in the discussion.

After 100% of module marks had been assigned to students, the facilitator released notes and audio recordings of both sessions to the researcher. The audio recordings were professionally transcribed and the transcriptions stored securely along with the digital audio files. Initial review of transcripts was carried out, supported by careful review of the original audio, including the correction of some programming-specific terms.

Kidd and Parshall (2000) notes the lack of specific standard analysis techniques for focus group data, although a number of catalogued qualitative analysis methods would initially appear applicable (Cohen et al, 2007). Thematic analysis (Braun and Clarke, 2006) was selected as the primary formal qualitative analysis tool, noting its easy application and summary abilities. Additionally, its capability to identify unexpected outcomes and to inform ongoing practice by easy generation of recommendations were noted. Iterative review of the transcripts facilitated distillation of the discussion to a small number of themes and sub-themes. The process was inductive, rather than theoretical, seeking to identify emergent changes to learning rather than answers to a specific question (Braun and Clarke,2006). Quotes were used on occasion to illustrate specific indicative views.

2.2 Practitioner reflection

To provide a secondary aspect, the researcher maintained a longitudinal reflective account during the course of the module delivery, noting the reported internalised personal benefits to the teacher and researcher of journalling their progress (Ortlipp,2008, Borg,2001). Much of the advice regarding observations given by (Cohen, 2007) can be equally applied to the keeping of a reflective diary, particularly around framing the structure of each entry. These reflections focused specifically on how the adoption of test-driven development appeared to influence learning. No student names or other identifying information were placed in any reflective accounts. The reflective diary was held in secure electronic storage.

2.3 Lab performance tracking

Lab performance data was collected from students in the form of the JUnit report files in XML format via Moodle and were turned in at the conclusion of each lab session. Quantitative data reporting is purely in aggregated for, and used to further illustrate some key points identified from the qualitative sources. Specific individual consent was not required to utilize this data since it was normally collected in the course of delivering this module and no individual tracking of students was performed. Following conclusion of the module, files for analysis presented here were programatically stripped of any possible identifying information, such as login name and home directory path.

3. Findings

3.1 Focus group findings

The focus group session attracted five participants, and lasted approximately 40 minutes. Subsequent thematic analysis identified three primary themes, unambiguous clarification of completion, facilitation of discussion and the need for justification.

The primary theme identified was that of certainty of completion. Students reported that the automated tests directly clarified whether the code had the desired result. However, participants also felt that it gave them freedom of interpretation by checking that final answer was correct. The use of testing gave a definite meaning to ``done'', avoiding doubt as to whether something was working.

Participants reported a changed mental attitude to lab work, where the immediate goal was now to pass the provided tests, sequentially passing each one rather than becoming paralysed by totality of task, with one participant noting to “Look at the green bar going up” referring to the visual onscreen indication of the each test sequentially passing. Students reported running the tests regularly, particularly after a change was made. This certainty of completion extended to the lab exercise as a complete unit. Although designed for in-class usage, the utility of the automated tests were mentioned for work finished at home also. Participants appeared to be positive towards the idea of submitting test report files for a small amount of credit.

The utility of unit testing appeared to extend out of the lab intervention and into summative assessments, where unit tests were provided in one case. Students expressed satisfaction with their provision, as they felt that they could make informed predictions about completion time and reasonably self-assess their own progress:

> it was really good to know where you stood in the project throughout, like oh I got 50% working, that's a good chunk of my marks, I can now get the rest of them. I can now work on the other files which are the rest of the tests instead of working on one file over and over

Additionally, they reported that in many cases, the unit tests provided much greater clarity than the narrative descriptions in the assignment briefs, one learner eloquently noted that “reading the JUnit was far easier than mulling through the English of the brief."

There appeared to be reasonably satisfaction with the amount and granularity of tests, although some participants expressed a preference for a small number of more challenging tests for those who finish early. Recognition that passing a particular test did not necessarily guarantee correctness was evident, with participants cogniscent of the effect of false passes. It was also appreciated that the test suite included one or two low-bar tests that were easy to pass. Students noted a jump in test complexity when transitioning from desktop to client-server programming exercises, suggesting more scaffolding may be required.

The second theme identified was the automated test as a facilitator of discussion with the lecturer and with their peers. The ability of the automated test to localise a problem for discussion, by framing it around a specific failing test was directly voiced: “instead of kind of going, you know, I've a null pointer exception, you can kind of say I've a problem with such and such a test." Participants identified the provision of identical starting files as a second pillar to enable this discussion to occur, since it was no longer necessary for peers to structurally familiarise themselves with each others' code.

The final and arguably most important theme was that of student disposition towards the approach. Students reported previous brief exposure to automated testing for grading assignments in other modules, but lacked motivation and experienced challenges with the software tools employed:

> kind of thrown at us an additional framework, we didn't have something like Maven to integrate the tests for us automatically so even getting them running in the first place was very, very difficult

By contrast, the fact that the testing was directly supported by the Maven project builder for Java was directly mentioned here, where tests are automatically discovered and executed during the project building process. More generally, participants mentioned on a number of occasions the importance that running tests be easy and routine, and that "buy in" must exist for the process to be successful.

Further refinement identified consistency of the signature pedagogy as a key sub-theme, through familiarity and ubiquity. The introduction of the tests at the beginning and consistency of application throughout the module was greatly valued. One direct benefit identified by participants was that of time-management, where "a lot of times as well he would finish class maybe a couple of minutes early as well and you would still have the better learning from it." In fact, students reported missing the tests where they weren't provided, such as in the final more open-ended assignment.

The sub-theme of an industrially relevant signature pedagogy emerged without direct exploration. Usage of standard workplace tooling was positively identified by one student visiting a prospective employer, who observed the use of the employed tooling. Specific mention was made of the need for adequate time to be set aside at the beginning of the module for students to set up their development environments and verify that the test framework was working. Furthermore, a mental association is clearly evident among a number of separate tools introduced in the module, the unit tests, the Maven project builder and the Git source code control system.

3.2 In-depth interview

The first scheduled focus group sessions attracted only one participant, who expressed a desire to the facilitator to go ahead with the session. Following ethical guidelines, the same protocol was followed as with the multi-participant focus group. The session lasted approximately 14 minutes. The transcript was separately coded and separately analyzed for emerging themes, independent of the multi-participant focus group that was previously reported in Section 5.1

The main identified theme was that of clarification of expectation. The tests were said to be helpful regarding staying within a reasonable timeframe, since they suggested where to start, and the ultimate goal to be achieved, and in doing so avoided wasting time.

> Definitely a whole lot easier because you sort of just had an idea of where to start and where you wanted to get to so you weren't wasting time on just like writing a whole lot of gibberish that won't work.

In particular, it emerged that they were able to provide rich information when something didn't work:

> If it worked the test ... it may or may not have passed but it would come up on the list of errors what was wrong and you could sort of go off that ... if you were just running it and it would just break or something you just wouldn't have a clue where you were.

Regular use of the test suite was reported by the participant, and the number of tests provided was said to be adequate. In terms of quality, the supplied tests were judged to be sufficient.

The second identified theme was the need for learners to internalise the relevance of the signature pedagogy. Previously, the participant reported not really knowing what testing was about. In particular, they reported a change in their own view on unit testing as a result of this experience, noting that: “I thought it was just a load of rubbish last year because I didn't really have any time for it really." However, asked if they would continue to write their own unit tests, as opposed to using supplied tests as in this module, they felt unsure. Yet, a desire was expressed to use them in future employment.

Furthermore, the provision of the tests encouraged compliance with naming guidelines for files and other program units. It was interesting in this regard that it was mentioned that at no stage was their any attempt to “cheat" on participant's behalf by changing the tests.

Asked if the approach of using unit tests was a good idea, it was recommended that this method be continued. A preference for tests to be supplied rather than having to write them was expressed. In particular, the value of lecturer-written tests was expressed in the recommendation: “Definitely don't just throw your students into doing JUnit. I'd definitely go for the approach of giving out JUnit tests and having to pass them." As previously noted, reliance was evident on the automated tests rather than the narrative descriptions of the assignment brief.

3.3 Practitioner experience

Practitioner experience was recorded by the researcher during the module at varying intervals throughout the semester. Following initial unfamiliarity with the use of automated tests to assist coding effort, their effect in the lab seemed overwhelmingly positive. Assisting individual students with difficulties became a less time-intensive and more targeted process, since failing tests could in most cases isolate particular portions of code causing problems. Most students appeared to attempt to solve a problem themselves, solicit peer consultation and only then ask the lecturer for assistance. Having an overview of test performance available when circulating throughout the room was helpful to gauge ongoing group progress.

The automated tests were also felt to provide much richer information to the lecturer than previously available. The grade was simply awarded for their submission, which limited any additional burden and avoided the appearance of the exercise as a summative process. At a group level, the available performance data allowed optimisation of the class workload, giving an overall indication of whether the group had as a whole completed the previous session's work. It enabled simple visual identification of particular tests that were failing across the group and also particular students who may be experiencing difficulties. The initial time investment in developing the tabulation and analysis scripting was definitely judged to be beneficial.

Contrary to inital fears, the introduction of automated testing did not cause too many technical issues. This good fortune seemed largely attributable to the adoption of the Maven project build tool, which offered an easy way to identically build and test Java code regardless of operating system or development environment. However, this tool was in many ways unnoticeable by students since most of its other features apart than integration automated testing were not discussed. By comparison, the Git version control system caused many more problems and resultant queries, yet these were still very manageable.

3.4 Lab performance tracking results

Summary data from nine laboratory sessions are presented in Table 1. Taking the official enrolment count as canonical, *n* = 15

Lab	Submission rate (%)	Mean pass rate (%)	All pass (%)
1	93	61	27
2	100	100	100
3	100	88	13
4	93	100	93
5	13	45	0
6	67	70	33
7	67	68	20
8	80	73	0
9	100	100	100

Table 1: Lab performance tracking metrics (n=15)

Given the small sample size and variable number and difficulty of tests in each tracked lab exercise, there are few global features readily identifiable from the presented data. Lab~5 was somewhat anomalous, since it was the first lab that introduced client/server programming, and unforeseen circumstances resulted in many students not being able to complete this session. session, causing a zero ``All pass'' rate. Students were advised to skip the offending test by simply annotating it for exclusion, but some did submit it as failing.

4. Discussion

The aim of this study was to evaluate the effects on student learning of adopting test-driven development as a signature pedagogy within an applied programming module. Laboratory exercises were augmented with lecturer-written automated test cases to provide formative feedback to students whilst coding. A primarily qualitative examination of its effects was undertaken utilizing learner input and practitioner reflection, with illustration from quantitative performance metrics. Independent inductive thematic analyses identified three key benefits of this signature pedagogy, completion, facilitation of discussion and the use of vernacular tools.

There was a clear concurrence amongst the student and practitioner views that the use of automated testing promoted completion. The lecturer-written tests did not require test composition on the part of the student, in contrast to other reported studies (Wick et al, 2005, Girard and Wellimgton, 2006, Edwards,2003). Instead, the tests objectively assessed the functionality to be delivered (Nicol and Mac Farlane, 2006), enabling self-regulation. The students identified positive motivational beliefs' being engendered by visually seeing the results of checking their code against the tests whilst working. The results would therefore suggest that the adopted approach improved laboratory completion and achievement in this group.

It was expressed in the focus group that students might welcome more challenging, or indeed optional tests. However, the literature would suggest that automated tests should rarely, if ever, be optional in industrial settings. In particular, testing theory states that the normal state of a system should be that all tests pass (Humble and Farley, 2011). This avoids the so-called ``broken window syndrome'' where failure of the suite as a whole becomes normal. There is, however, scope to improve exercises by perhaps giving the opportunity for students to derive new automated test cases by example.

Practitioner and student views both cited testing as an aid to problem resolution, noting that a student could articulate their difficulties in terms of a failing test. Targeted self-diagnosis as well as discussion with peers and staff was better informed through identification of the particular functionality that was lacking, and the conditions under which failures occurred. However, this key contribution depends on lecturer-supplied tests to ensure commonality, and a lightweight toolset that promotes frequent test usage, differing from other reported studies that emphasized test composition (Wick et al, 2005, Edwards, 2003).

Both student input and practitioner reflection cited the provision of a common stating project as a key facilitator of discussion amongst students themselves. This was particular relevant, given that students used various different development environments and operating systems in their coding activities. The availablility of software project management tools such as Maven, which transparently integrates automated testing, was a key enabler in this process.

The issue of alignment between in-class practice and continuous assessments was highlighted within the focus group. Constructive alignment theory would suggest that there be a visible alignment between class practice and assessment activities (Biggs,1999). Whilst this does not mandate that every assessment is accompanied by automated tests, the rationale for their provision or non-provision may require clearer explanation.

Although the Java programming language includes a high degree of support for automated testing, the pedagogical methods explored do not depend on the particular technologies employed in this case study. Automated testing is routinely employed when other programming in other languages, such as C++, Python and C\#. Opportunities exist to extend the use of formative automated testing to programming exercises in non-computing disciplines, such as engineering computation, quantitative finance and theoretical science subjects, where numerical programming environments such as SciLab (Scilab-enterprises, 2012), Octave (Eaton et al, 2008) and R (R Core Team 2014) are regularly employed.

As presented, this study contains a number of direct limitations in both the intervention and evaluation methodologies. By its nature, the subject matter in the *Web Frameworks* module lent itself to examination by automated tests. More generally, this may present problems with certain interactive software that students commonly are asked to build. However, solutions have been proposed in the education literature regarding programs that use keyboard interaction (Proulx and Rasala, 2004,java} and graphical user interfaces (Thornton et al, 2008).

Although the tests do positively assess required functionality, they neither measure nor discourage the development of unrequested code. Assessment of code coverage by tests as a proxy for code-on-task may be helpful, as recognized industrially (Tahchiev, 2011). There are a number of free industrial code coverage tools that can be easily integrated into Java projects using the Maven builder. Students would require a reasonably brief tutorial in interpretation of code-coverage results to maximise its benefit.

A key auxiliary output was the development of a program to tabulate test passes per student and aggregate statistics. This proved to be a key aid to the lecturer, by providing information to inform teaching, a key requirement of quality feedback (Nicol and MacFarlane, 2006). The interface design and reporting are rather basic and require further development. Although not overly onerous, the possibility exists to replace Moodle submission by developing new Maven plugins to automatically submit test results in real time without student intervention. Such a facility would allow the lecturer to view test results in real time during the lab session, and would offer a major complementary advance on educational software testing tools (Pullan et al, 2013, Edwards, 2003).

There were additionally some limiting factors in the evaluation methodology that require brief consideration. The principal consideration relates to the sample size of 15, and caution against undue generalizations must be maintained (Cohen et al, 2007). Also, as both the small sample size and ethical considerations prevented the formation of an otherwise-comparable control group, there is a lack of contemporaneous comparison. These limitations were partly due to the time-bound nature of the study, which was performed as part of the author's studies towards the MA in Learning and Teaching.

Whilst the results of this study provide a certain level of support for the use of formative test-driven development as a signature pedagogy in computer programming laboratories, it cannot unconditionally recommend the implemented approach. Further trials of the developed pedagogical method on larger student cohorts in both the university and Institute of Technology sectors may provide additional insight. The evaluation presented occurred in an entirely face-to-face setting, and a similar evaluation in fully online or blended delivery modes may yield useful additional insight. Many programmes now incorporate an industrial work placement, and the effectiveness of classroom-based test-driven development might vary depending on whether the student has yet completed their placement. Similarly, the suitability of the employed teaching methods in emerging apprenticeship-based software development programmes remains untested.

5. Conclusion

This study describes the adoption of test-driven development as a signature pedagogy in applied programming practicals. Lecturer-supplied automated test cases were used as a formative assessment tool to supply specific rapid feedback on-demand to students completing laboratory work. This intervention was achieved using only free industry-standard software testing tools that the students already had access to.

The efficacy of the test-driven signature pedagogy was deemed successful within its delivered context by learners in a qualitative study. Students positively identified the instantaneous nature of the feedback and the continuing motivation to close the gap between actual and desired performance.

The ability to explicitly identify the test failing opened up peer discussion and increased the quality of the dialogue between peers and with the lecturer. The lecturer noted that the self-diagnosis provided to students by the automated tests permitted a more targeted level of assistance from the lecturer during the limited lab time available.

Developed artifacts included basic scripting to collate reports from in-lab test runs to guide lecturer assistance in subsequent sessions. Future technical work includes provision of real-time visibility of test pass/fail status to the lecturer. The applicability of the employed teaching methods in larger groups sizes, different institutional contexts and varied learning modes also remains to be investigated. Opportunities also exist to apply elements of the described pedagogical practice within other disciplines such as engineering, quantitative finance and theoretical sciences.

Acknowledgements

The assistance of Dr Bernadette Brereton, Dr Moira Maguire and Mr Gerry Gallagher from the Centre for Excellence in Learning and Teaching at Dundalk Institute of Technology throughout this study is sincerely acknowledged.

Focus groups were facilitated by the author's colleague Dr Caroline Sheedy, whose generous help is greatly appreciated.

6. References

Allen, E., Cartwright, R., and Stoler, B. (2002). DrJava: a lightweight pedagogic environment for java. *SIGCSE Bull.*, 34 (1), pp. 137–141.

Azemi, A. (1995). Teaching computer programming courses in a computer laboratory environment. In *Frontiers in Education Conference, 1995. Proceedings., 1995*, volume 1. pp. 2a5.18–2a5.20 vol.1.

Barriocanal, E.G., Urbán, M.A.S., Cuevas, I.A., and Pérez, P.D. (2002). An experience in integrating automated unit testing practices in an introductory programming course. *SIGCSE Bull.*, 34 (4), pp. 125–128.

Beck, K. (1999). Embracing change with extreme programming. *IEEE Computer*

Magazine. Beck, K. (2001). Aim, fire [test-first coding]. *Software, IEEE*, 18 (5), pp.

87–89.

Beck, K. and Andres, C. (2004). *Extreme Programming Explained: Embrace Change*. Addison- Wesley Professional, 2nd edition.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development. http://www.agilemanifesto.org.

Biggs, J. (1999). What the student does: teaching for enhanced learning. *Higher Education Research & Development*, 18 (1), pp. 57–75.

Biggs, J. (2003). Aligning teaching and assessing to course objectives. In *Teaching and Learning in HIgher Education: New Trends and Innovations*. University of Aveiro, pp. 1–7.

Borg, S. (2001). The research journal: a tool for promoting and understanding researcher develop- ment. *Learning Teaching Research*, 5 (2), pp. 156–177.

Braun, V. and Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3 (2), pp. 77–101.

Brown, S. (2004). Assessment for learning. *Learning and Teaching in Higher Education*.

Carlson, B. (2008). An agile classroom experience: Teaching TDD and refactoring. In *Agile, 2008. AGILE '08. Conference*. pp. 465–469.

Chickering, A.W. and Gamson, Z.F. (1987). Seven principles for good practice in undergraduate education. *AAHE Bulletin*, 40 (7), pp. 3–7.

Christensen, H.B. (2003). Systematic testing should not be a topic in the computer science curricu- lum! *SIGCSE Bull.*, 35 (3), pp. 7–10.

Clarke, M., Kenny, A., and Loxley, A. (2015). *Creating a Supportive Working Environment for Academics in Higher Education: Country Report Ireland*. Irish Federation of University Teachers and Teachers' Union of Ireland.

Clarke, P.J., Davis, D., King, T.M., Pava, J., and Jones, E.L. (2014). Integrating testing into software engineering courses supported by a collaborative learning environment. *Trans. Comput. Educ.*, 14 (3), pp. 18:1–18:33.

Cohen, L., Manion, L., and Morrison, K. (2007). *Research methods in education*. Routledge, 6th edition.

Cooper, N.J. (2000). Facilitating learning from formative feedback in level 3 assessment. *Assessment*
& Evaluation in Higher Education, 25 (3), pp. 279–291.

Daku, B. (2009). Individualized laboratory using moodle. In *Frontiers in Education Conference, 2009. FIE '09. 39th IEEE*. pp. 1–5.

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

Desai, C., Janzen, D.S., and Clements, J. (2009). Implications of integrating test-driven development into cs1/cs2 curricula. *SIGCSE Bull.*, 41 (1), pp. 148–152.

Devedžić, V. and Milenković, S. (2011). Teaching agile software development: A case study. *IEEE Transactions on Education*, 54 (2), pp. 273–278.

Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5 (3).

Dowden, T., Pittaway, S., Yost, H., and McCathy, R. (2013). Students' perceptions of written feedback in teacher education: ideally feedback is a continuing two-way communication that encourages progress. *Assessment & Evaluation in Higher Education*, 38 (3), pp. 349–362.

Edwards, S.H. (2003). Improving student performance by evaluating how well students test their own programs. *ACM Journal of Educational Resources in Computing*, 3 (3).

Edwards, S.H. (2004). Using software testing to move students from trial-and-error to reflection-in- action. *SIGCSE Bull.*, 36 (1), pp. 26–30.

Evans, C. (2013). Making sense of assessment feedback in higher education. *Review of Educational Research*, 83 (1), pp. 70–120.

Gibbs, G. (2010). *Using assessment to support student learning*. Leeds Metropolitan Press.

Gibbs, G. and Simpson, C. (2004). Conditions under which assessment supports students' learning. *Learning and Teaching in Higher Education*, 1.

Girard, C. and Wellington, C. (2006). Work in progress: A test-first approach to teaching CS1. In *Frontiers in Education Conference, 36th Annual*. pp. 19–20.

Higgins, C.A., Gray, G., Symeonidis, P., and Tsintsifas, A. (2005). Automated assessment and expe- riences of teaching programming. *J. Educ. Resour. Comput.*, 5 (3).

Higgins, R., Hartley, P., and Skelton, A. (2001). Getting the message across: The problem of com- municating assessment feedback. *Teaching in Higher Education*, 6 (2), pp. 269–274.

Hitchner, L.E. (1999). An automatic testing and grading method for a C++ list class. *SIGCSE Bull.*, 31 (2), pp. 48–50.

Humble, J. and Farley, D. (2011). *Continuous delivery*. Addison-Wesley.

Janzen, D.S. and Saiedian, H. (2007). A leveled examination of test-driven development acceptance. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, pp. 719–722.

Jones, E.L. (2001). Integrating testing into the curriculum - arsenic in small doses. *SIGCSE Bull.*, 33 (1), pp. 337–341.

Kennedy, D.M. (1998). Software development teams in higher education: An educator's view. *Flexi- bilITy: The next wave*, pp. 373–385.

Kidd, P.S. and Parshall, M.B. (2000). Getting the focus and the group: Enhancing analytical rigor in focus group research. *Qualitative Health Research*, 10 (3), pp. 293–308.

Kollanus, S. and Isomöttönen, V. (2008). Test-driven development in education: Experiences with critical viewpoints. *SIGCSE Bull.*, 40 (3), pp. 124–127.

Krueger, R.A. (2002). Designing and conducting focus group interviews. http://www.eiu.edu/~ihec/Krueger-FocusGroupInterviews.pdf.

Lalor, K. (2010). 'the same, but different': Salary scales, progression arrangements and duties in Institutes of Technology (IoTs) and universities. *Administration*, 58 (3), pp. 79–105.

McCabe, A. and O'Connor, U. (2013). Student-centred learning: the role and responsibility of the lecturer. *Teaching in Higher Education*.

McKinney, W. (2010). Data structures for statistical computing in python. In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference*. pp. 51 – 56.

McLafferty, I. (2004). Focus group interviews as a data collecting strategy. *J Adv Nurs*, 48 (2), pp. 187–94.

Miller, F.P., Vandome, A.F., and McBrewster, J. (2010). *Apache Maven*. Alpha Press.

Nicol, D. (2006). Increasing success in first year courses: Assessment re-design, self-regulation and learning technologies. In *Proceedings of the 23rd annual ascilite conference: Who's learning? Whose technology?* University of Surrey, pp. 589–598.

Nicol, D. (2007). E-assessment by design: using multiple-choice tests to good effect. *Journal of Further and Higher Education*, 31 (1), pp. 53–64.

Nicol, D.J. and Macfarlane-Dick, D. (2006). Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in higher education*, 31 (2), pp. 199–218.

Olan, M. (2003). Unit testing: Test early, test often. *J. Comput. Sci. Coll.*, 19 (2), pp. 319–328.

Ortlipp, M. (2008). Keeping and using reflective journals in the qualitative research process. *The Qualitative Report*, 13 (4), pp. 695–705.

Proulx, V.K. and Rasala, R. (2004). Java IO and testing made simple. *SIGCSE Bull.*, 36 (1), pp. 161–165.

Pullan, W., Drew, S., and Tucker, S. (2013). An integrated approach to teaching introductory pro- gramming. In *e-Learning and e-Technologies in Education (ICEEE), 2013 Second International Conference on*. pp. 81–86.

Reis, C. and Cartwright, R. (2004). Taming a professional IDE for the classroom. *SIGCSE Bull.*, 36 (1), pp. 156–160.

Royce, W. (1970). Managing the development of large software systems. In *IEEE WESTCON*. Los Angeles, pp. 328–338.

Sadler, D.R. (1989). Formative assessment and the design of instructional systems. *Instructional Science*, 18 (2), pp. 119–144.

Shulman, L.S. (2005). Signature pedagogies in the professions. *Daedalus*, 134 (3), pp. 52–59.

Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J.K., and Padua-Perez, N. (2006). Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. *SIGCSE Bull.*, 38 (3), pp. 13–17.

Tahchiev, P., Leme, F., Massol, V., and Gregory, G. (2011). *JUnit in Action*. Manning.

Thornton, M., Edwards, S.H., Tan, R.P., and Pérez-Quiñones, M.A. (2008). Supporting student- written tests of GUI programs. *SIGCSE Bull.*, 40 (1), pp. 537–541.

Wick, M., Stevenson, D., and Wagner, P. (2005). Using testing and JUnit across the curriculum. *SIGCSE Bull.*, 37 (1), pp. 236–240.

Appendix A Focus group questions

The following questions were used to guide discussion in focus group sessions,

1. How, if at all, was your approach to a given problem in the lab influenced by the provision of automated tests?

2. On average, how often did you run the test suite in full during the lab session?

3. Did you experience any technical impediments with the automated testing process? If so, what were they?

4. How would you rate the quantity of tests for each lab exercise, would you have preferred significantly more or less?

5. What changes did the provision of automated tests make to your interactions with other peers and or staff in the lab?

6. What inhibitions did the use of automated testing place on you in the lab?
7. Were you in any way hindered / limited in your work by their presence?

8. Was your ability to complete the lab session within the alloted class time influenced any way by the use of the automated tests?

9. Did your approach to coding outside of the lab environment change as a result of this intervention?

10. If this module were to be delivered a second time using the same method, what changes would you make?