

Tool Support for Test Generation in Test-Driven Development

Boh Sui Jimm

*This report is submitted as partial fulfilment
of the requirements for the Honours Programme of the
School of Computer Science and Software Engineering,
The University of Western Australia,
2007*

Abstract

Test-Driven Development (TDD) of software systems is characterised by incremental development and frequent testing. Attempts to teach TDD in computer science and software engineering degrees has garnered some mixed reactions from students. Problems encountered by students include finding the test-first concept difficult to adapt to, and not seeing the short-term benefits of TDD because of the amount of effort required to develop tests during development.

In my project, I first investigated some of the problems encountered by UWA software engineering students in learning TDD, by evaluating programming exercises using TTD. A unit test case generation tool, UnitGen, and accompanying tutorial, was then developed to address these problems. UnitGen guides students in the building of test cases and also reduces the amount of effort required to write and execute the test cases. UnitGen addresses deficiencies for teaching TTD that were observed in existing automated test tools such as JUnit. Firstly, it helps generate unit test cases based on inputs provided by the user. Secondly, it provides a mechanism to facilitate the testing of object states without having to modify the source code under test. This is achieved by providing access to private object fields using Java reflection. The UnitGen tutorial, provides guidelines for generating black-box unit test cases as well as a user-manual on how to use UnitTestGen.

Keywords: Test-Driven Development, tdd, tool, unitgen

CR Categories: D.2 Software Engineering, D.2.5 Testing and Debugging, K.3.2 Computer and Information Science Education

Acknowledgements

I would like to thank my supervisor, Dr. Rachel Cardell-Oliver, for her guidance, with my thesis writing and experiment setups, especially for allowing me to demonstrate my tool during her SRPM workshops. Dr. Gordon Royle for helping me with Java reflection. My family and friends for their constant support. Participants of my experiments for taking their time to help me.

Contents

Abstract	ii
Acknowledgements	iii
1 Introduction	1
1.1 Test Driven Development	1
1.1.1 Three Aspects of TDD	2
1.2 Why Introduce TDD to School	3
1.3 Problem	4
1.4 My Hypothesis	6
1.5 Related Research	7
1.5.1 Experiments and Case Studies on TDD	7
1.5.2 Comparison of Experiment Setups	7
1.5.3 Comparison of Experiment Results	8
1.5.4 Survey Results on TDD	12
1.5.5 Conclusion on TDD Research	13
2 Tool Support for TDD	14
2.1 JUnit - An Automated Testing Framework	14
2.2 Scaling up JUnit - JNuke Project	17
2.3 Scaling up JUnit - JML Project	18
2.4 Conclusion on Automatic Unit Test Tools	22
3 Test-Driven Development Case Studies	23
3.1 Overview	23
3.2 Experiment 1 - Build a JUnit Testclass using TDD Approach . . .	24
3.2.1 Experiment 1 - Setup	24

3.2.2	Experiment 1 - Results	24
3.2.3	Experiment 1 - Conclusion	26
3.3	Experiment 2 - Evaluate First Year Software Engineering (SE) Students' Attempts with TDD	26
3.3.1	Experiment 2 - Setup	26
3.3.2	Experiment 2 - Results	28
3.3.3	Experiment 2 - Conclusion	30
3.4	Experiment 3 - Interview of Industrial and Academic Professionals	31
3.4.1	Experiment 3 - Setup	31
3.4.2	Experiment 3 - Results	32
3.4.3	Experiment 3 - Conclusion	33
4	UnitGen Implementation	34
4.1	Rationale	34
4.1.1	Tutorial	34
4.1.2	UnitGen	35
4.2	Tool Design	35
4.2.1	Tutorial	35
4.2.2	UnitGen	36
5	Tool Evaluation	43
5.1	Experiment 4 - Two workshop sessions on TDD and UnitGen . .	43
5.1.1	Experiment 4 - Setup	43
5.1.2	Experiment 4 - Results	45
5.1.3	Experiment 4 - Conclusion	47
5.2	Experiment 5 - Further Experiment on UnitGen versus. JUnit . .	48
5.2.1	Experiment 5 - Setup	48
5.2.2	Experiment 5 - Results	49
5.2.3	Experiment 5 - Conclusion	49
6	Conclusion	50
6.1	Future Work	51

A Original Honours Proposal	53
B Tutorial	61

List of Tables

3.1	Analysis of Errors Found	29
5.1	Survey Feedback Analysis	47
5.2	Results of UnitGen vs. JUnit on Eclipse	49

List of Figures

1.1	Summary of Experiment Setups	8
1.2	Summary of Experiment Results	9
2.1	A Translation Scheme for Runtime Assertion Checking of JML Specifications[6]	20
2.2	An Example of skeleton test code generated from JML specifications[6]	21
3.1	Analysis of 29 Students' Passing Rate for 39 Tests	28
4.1	Overview of UnitGen Process Flow	38
4.2	Overview of genTestMethods Process Flow	41
5.1	Survey Questions Rating System	44
5.2	Survey Results on Programmer Experience	45
5.3	Survey Results on UnitGen and my Tutorial	46

CHAPTER 1

Introduction

1.1 Test Driven Development

Test-Driven Development (TDD), a test-first approach to software development, gained popularity with the rise of agile process models such as Extreme Programming (XP). Although, TDD gained visibility only in the recent years with the introduction of Extreme Programming in 1998, it has been practiced informally for decades with one of the earliest references to its use in the late 1950s in the NASA Project Mercury [12].

TDD, also commonly referred to by various names such as test-first programming, test-driven design or test-first design [12], adopts a novel approach to software engineering by generating test cases before writing the source code, and using the test cases to drive the development and design of the system.

In traditional programming practices, unit tests are constructed after code is written, which could be anytime after the code is written, be it a few minutes or even months later. In addition, unit tests might be written and conducted by either software testing teams or the same programmers who written the source code. In the case of TDD, unit test cases are generated by the programmers before writing the source code, and testing is conducted by the programmers immediately after the unit source codes are written.

TDD, however, is more than just a test-first approach, as mentioned by Kent Beck, creator of Extreme Programming and co-founder of Agile Manifesto and JUnit unit testing framework[16]. TDD takes the test-first approach to the extreme by always writing tests before code, making tests as small as possible , and never letting code degrade [12]. Scott Ambler[1], in his article “Introduction to Test-Driven Development (TDD)”, described TDD with a simple formula, “TDD = TFD + Refactoring”. Refactoring, a key aspect of TDD, refers to changing the structure of a code without changing its external behaviour. That is, improving the code structure yet ensuring it still passes the tests [12].

1.1.1 Three Aspects of TDD

A common misconception about TDD is that it is a testing technique, in truth; its nature is far more complex. There are three aspects to TDD, as its name suggests; Test, Driven and Development.

Test Aspect which most people are familiar with, involves designing automated tests for each individual unit of a program. A unit in this case, refers to the smallest possible testable software component, such as a method or a procedure or in some cases a class in an object-oriented context. While tests may or may not need to be automated, automated testing reduces the effort required to conduct testing, particularly regression testing of large software systems. In addition, TDD assumes the existence of automated testing, as it requires frequent execution of tests and regression testing, for its iterative development cycles[12].

Without, automated testing, TDD would be too much of a hassle to practice. In fact, the existence of automated testing tools such as the xUnit family of frameworks, contributed to the rising popularity of TDD. Testing in TDD, as mentioned earlier in the text, refers to designing and writing automated unit tests before the code, and executing them immediately after the code is written to provide prompt feedback[12].

Driven Aspect refers to TDD leading analysis, design and programming decisions, achieved through refactoring [12]. TDD's Driven Aspect is based on two assumptions, firstly, software design is either incomplete or pliable and thus open to changes. This agrees with Agile Process Models' idea of adaptive development. Secondly, as the process of test writing is one of the first steps in deciding what a program should do, it is essentially a form of analysis [12].

Based on the order of approach in TDD, since tests are written before coding, and the assumption that test writing is a form of analysis, it leads to the conclusion that the process of writing tests drives the design of the system. Agile Alliance, a non-profit organization, provides a definition that captures this concept: "Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code." [12].

The idea is that the tests can be generated before actual code is written and the process of doing so can be used to drive the development and design

of the system as the tests control the behaviour of what the actual code should do indirectly through the fact that the functionality codes must pass the tests designed in order to proceed on in the TDD development cycle[12].

Development Aspect implies that TDD is not itself a software development methodology or process model, instead it is to be used in the context of other process models as a form of micro-process [12].

TDD assumes that automated tests generated throughout the development cycle are not discarded once a design decision is made, instead they become a crucial part of the development cycle by providing prompt feedback to any subsequent changes made to the system. This allow developers to make changes with confidence as regression testing can be executed immediately after and should any change results in a failure, the tests are still fresh in the developers mind. However, a drawback of this practice is that the developer must now maintain both the coding and the suite of automated tests generated thus far [12].

1.2 Why Introduce TDD to School

Educators have been constantly struggling with trying to improves a student's comprehension and analysis skills. It has been noted that most students rely on trial and error approach to fixing errors and debugging[8, 7]. In addition, it is also noted that in most education curriculam, there has been insufficient coverage on the subject of testing[27, 25].

TDD's emphasis on incremental development and frequent testing has lead many educators to push for the introduction of TDD into education curriculum[7, 25, 27]. Edwards from Virginia Tech's Department of Computer Science suggested that TDD is attractive for use in an educational settings for the following reasons[8]:

1. It promotes incremental development, and promotes the concept of always having a working version of the program.
2. It promotes early detection of errors introduced by coding changes through frequent testing.
3. Its focus on incremental development and frequent testing, directly combats the "big bang" integration problems that students encounter when they write larger programs and leave testing till the end.

4. It dramatically increases student's confidence in the portion of coding that they have finished.
5. It allows students to make changes and additions with greater confidence due to continuous regressions testing.
6. It increases the student's understanding of the assignment requirements, by forcing them to explore grey areas in order to completely test their solution.
7. The growing test suite developed by students using TDD, acts as a progress indicator, allowing students to be clearly aware of how much of the required behaviour for the program has already been completed.
8. Students begin to see the above stated benefits for themselves after using TDD on just a few assignments.

In addition, the results of many studies and experiments which suggest that TDD improves code quality, increased confidence in code correctness and programmer productivity in terms of program and requirements understanding[22, 10, 12, 32, 9, 4] only serve to provide more incentive for the introduction of TDD.

Studies on the introduction of TDD into education curriculum have also met with favorable results both in terms of students' programs' code quality as well as students' response to TDD as a software development approach[8, 19]. Most importantly, it is noted that most students exposed to TDD, view it favorably as their choice of software development approach[8, 13].

1.3 Problem

TDD however is not without its drawbacks, many problems have been observed in the numerous studies and experiments conducted on TDD in both industrial and academic settings[19, 14, 10, 4].

Several case studies and experiments in industry noted that using TDD resulted in about 15% increase in development time[4, 10]. This can be attributed to the observation that that TDD typically results in more test cases written compared to traditional test-last approach to software testing[9, 32, 10, 13].

In academic settings, surveys on students involved in case studies and experiments on TDD highlighted several issues that hinder the adoption of TDD amongst novice programmers. Firstly, some students felt that the lack of upfront design phase in TDD proves problematic to learning TDD[10].

Secondly, a significant number of students thought that getting into the TDD mindset is difficult[10]. This could be attributed to the foreign concept of Test-first, as it was noted in a case study by Melnik that some students believe Test-first approach is almost like working backwards thus logically confusing[19]. Melnik's case study observed that some students felt that writing the test code is more a part of design than testing, supporting the hypothesis that writing tests before functionality is difficult as TDD forces design issues forward[19, 12].

Thirdly, some students also felt that testing involves too much effort and they did not see the short-term benefits in using TDD[19]. Lastly, students also did not know how many tests would be sufficient and when do they stop testing[19].

In addition, to problems faced by students in learning TDD, Mugridge, in a two-year TDD curricular experiment at The University of Auckland, observed that using TDD as a design approach requires certain set of skills such as the following[14]:

- Testing and test case writing.
- Use of automated testing frameworks.
- Message tracing to troubleshoot test failure.
- Writing the simplest possible implementation code to meet test specifications.
- Method and class refactoring.
- Common refactor patterns.
- Packaging unit tests into appropriate acceptance tests.

Mugridge also suggested that, for introductory programming courses, that a reduced set of TDD skills be taught and a lightweight testing framework that uses simple assert statements be used[14].

Others have also noticed the lack of coverage in education curriculum with regards to testing[27, 25]. From personal experience, in The University of Western Australia, for most assignments it is typically left to the students to test their own work if they wish to. In addition, from my experiments in evaluating 1st year Software Engineering(SE) workshop submissions, it was observed that students found it difficult to create a set of comprehensive test cases despite existing support tools for TDD such as JUnit in Eclipse3.3. This is because, tools such as JUnit despite providing support for test case writing and execution, does not help in choosing test data or generating the test cases themselves.

1.4 My Hypothesis

For TDD to be readily accepted and used effectively by students, we believe that there is a need to resolve the issues raised thus far. The strategy suggested in this project dissertation is two-pronged. Firstly, a tutorial is needed to guide students on testing and test case writing. In addition, this tutorial will also provide information on how to use my unit test case generation tool, called Unit Test Case Generator, UnitGen for short. This tutorial, is not however a replacement for a course on testing, rather it is to cover the basic concepts on testing that is required to use UnitGen effectively.

Secondly, a unit test case generation tool, UnitGen, is to be developed to aid students in developing a suite of unit test cases, in the form of a JUnit testclass, for use with the JUnit testing framework. UnitTestGen works by accepting test parameters from users for methods they wish to test, and generating black-box test cases based on user inputs. The features to be incorporated into UnitTestGen seeks to eliminate deficiencies observed in the JUnit wizard support provided for Eclipse. The reasons for using JUnit as the testing framework as well as features of UnitGen will be discussed later under the section of Tool Rationale.

Our two-prong strategy aim is to eliminate or mitigate to some extent the following problems that have been raised:

1. Insufficient coverage on testing and test case writing in education curriculum.
2. Training in the use of automated testing frameworks; in this case, using UnitTestGen with JUnit 3.8.1 on Eclipse.
3. Reduce development time, by reducing the effort needed in developing test cases.
4. Help students see the short-term benefits in TDD, by reducing the effort required in testing.
5. Help students with the foreign-concept of Test-First, by providing a tutorial which guides students on selecting suitable testing parameters black-box test cases for use with the UnitGen tool.

While our strategy certainly does not eliminate all problems observed in the case studies and experiments reviewed, it does cover most of the common problems observed.

The main contributions of my research are

1. Identifying and evaluating the barriers for students learning TDD.
2. A tutorial to address test case generation issues, which also includes a user-manual for UnitGen.
3. UnitGen, a unit test case generation tool, that supports the management and automatic generation of test cases.

1.5 Related Research

1.5.1 Experiments and Case Studies on TDD

Advocates of TDD claim that it provides benefits such as improving product quality and programmer productivity while skeptics maintain that it is both counterproductive and difficult to learn[9]. With such differing views on TDD, there have been various studies and experiments conducted in an attempt to prove the effectiveness of TDD, both in the industry as well as in academia. In this section, I will review the various studies and experiments conducted on TDD, and compare how the experiments were setup as well as the results obtained.

1.5.2 Comparison of Experiment Setups

In order to compare the effectiveness of TDD, we look into how each experiment is set up to give an idea of which variables are held constant. This in turn allows us to have a fair perspective at the results obtained from the experiments when measured up against each other.

Most of the experiments and studies conducted are similar in the way they are set up. For example, five out of six of the experiments and case studies reviewed used Java as the programming language with JUnit as the testing framework [10, 32, 8, 22, 9]. Only one used C++ with their own testing framework based on CUnitTest class from unittest.h[4].

All of them focused on evaluating TDD, with half of them being controlled experiments[10, 22, 9] and the other half case studies[32, 8, 4]. In addition, half of them used professionals as test subjects[32, 10, 4], while the remainder used students[8, 22, 9]. Experiences regarding using TDD amongst the test subjects varies as well, with most of them having at least basic experience [8, 22, 9, 10]. Professionals that participated in Williams and Maxmillian's case study did not have prior experience with TDD[32] while there was no mention regarding test subjects' experiences with TDD in Bhat and Nagappan's case study[4].

Figure 1.1: Summary of Experiment Setups

Authors	Study Type	Subjs.	Prog. Lang.	Prior TDD Exp.	Sample Size (TDDG/CG)	Study Scope	Testing Framework	Testing Approach (TDDG/CG)
Williams, L. & Maximilien, E. M. [7]	Case Study	Prof.	Java	None	9 / 5	TDD	JUnit	Test-First vs. Ad-Hoc
George, B. & Williams, L. [4]	Cont. Exp.	Prof.	Java	Varied: Beginner to Expert	12 / 12	TDD	JUnit	Test-First vs. Test-Last
Bhat, T. & Nagappan, N. [1]	Case Study	Prof.	C++	N/A	5-8 / N/A	TDD	CUnitTest class	Test-First vs. N/A
Edwards, S. H. [2]	Case Study	Students	N/A	Basic	59 / 59	TDD	WebCat	Test-First vs. Test-Last
Mullar, M. M. & Hagner, O. [6]	Cont. Exp.	Students	Java	Basic	10 / 9	TDD	JUnit	Test-First vs. Test-Last
Erdogmus, H., Morisio, M. & Torchiano, M. [3]	Cont. Exp.	Students	Java	Basic	11 / 13	TDD	JUnit	Test-First vs. Test-Last

Abbreviations used: Prog. Lang. - Programming Language, Exp. - Experience, TDDG – TDD Group, CG – Control Group, Prof. - Professionals, N/A – No Information Available, Cont. Exp. - Controlled Experiment, Subjs. - Subjects

With regards to sample size, the majority of the experiments have relatively small sample size for both the TDD group and the control group[32, 10, 4, 22, 9]. Only one case study was based off a large sample size of 59 students for both the TDD group and the control group[8]. The study by Bhat and Nagappan which compared applications developed with TDD against comparable past projects, failed to mention the size of the development team of past projects[4].

Lastly, four out of six of the experiments, focus on comparing test-first approach to testing against test-last approach to testing [10, 8, 22, 9]. Out of the remaining two, Williams and Maxmillian’s case study compared test-last approach against ad-hoc testing[32] while Bhat and Nagappan’s case study did not state the testing approach used in developing the past projects that were used for comparison[4].

1.5.3 Comparison of Experiment Results

TDD’s efficacy is evaluated by determining if it improves either code quality, productivity or both. Code quality is typically determined by measuring the number of defects within the application developed through black-box test cases[8, 22, 32, 9, 10, 4].

Productivity on the other hand has many conflicting views as to what constitutes as improved productivity. Some measure productivity by the development time of the application[4, 10, 32, 22], while others include factors such as program and requirements understanding and number of test cases written per unit of programming effort[9]. In order to provide a fair comparison of results, we consider productivity improved only if development time is reduced as some of the papers that was used for the survey did not provide sufficient information on other aspects that can be considered improved productivity, e.g. effect on programmer’s understanding.

Number of test cases written is also not taken into account as TDD which pushes testing to the forefront typically results in a more significant number of test cases generated compared to traditional software development approaches where testing is usually left out in the end[9, 32, 10, 13].

Figure 1.2: Summary of Experiment Results

Authors	Quality	Productivity
Willaims, L. & Maximilien, E. M. [7]	Better	No Difference
George, B. & Williams, L. [4]	Better	Worse
Bhat, T. & Nagappan, N. [1]	Better	Worse
Edwards, S. H. [2]	Better	N\A
Mullar, M. M. & Hagner, O. [6]	No Difference	No Difference
Erdogmus, H., Monsio, M. & Torchiano, M. [3]	No Difference	N\A

Abbreviations used: N\A – No Information Available

In industry, William and Maxmillian’s joint case study by North Carolina State University, Department of Computer Science and IBM Corporation was conducted to observe the effect of TDD as a defect-reduction practice. Participants of the case study were software engineers of an IBM development group, with experience in developing device drivers for more than a decade. They also developed the legacy product used as the baseline in the case study. The other group of participants were nine full-time IBM engineers, without prior experience with TDD and novices to the target device used as the baseline. This second group was to develop a “new” system for the target device through the practice of TDD. Both groups’ code would be evaluated against an external FVT (Functional Verification Test) group, which would run the code against 3 sets of black-box tests that have been generated: partly automated, fully automated and fully manual. It was observed that the defect rate (i.e. faults per KLOC or fault density) of the code of the “new” system developed using TDD, performed significantly better in the FVT/regression testing compared to the legacy system developed by the experienced IBM development group which primarily used ad-hoc testing. The “new” system seems to show a 40% lower defect density, and this was achieved with minimal impact to developer productivity[32].

Other studies in industry shows TDD improving code quality at the cost of development time[10, 4]. In George and William’s experiment in North Carolina State University, 24 professional pair programmers was split into two groups. One group used TDD development while the other used a waterfall-like model(i.e. design-develop-test-debug). Both groups were to develop a small bowling game application in Java, in addition all programmers were randomly assigned to work in pairs, using pair programming practice. The results showed that TDD developers produced higher quality code which passed 18% more functional black box test cases at the cost of 16% more development time attributed to developing test cases. The comparison of development time however is skewed as only one pair out of six in the control group developed worthwhile test cases. It was noted in a survey of the test subjects that majority of them believe TDD improves programmer understanding[10].

Bhat and Nagappan’s case study evaluates the efficacy of TDD in two different environments, i.e. Windows and MSN divisions of Microsoft. A significant increase in quality of code by an order of at least 2 was observed for projects developed using TDD as compared to similar projects developed in non-TDD approach. This improvement of quality came at a cost of at least 15% development time in the form of upfront time taken for writing test cases[4].

In the academia, Edwards’s case study at Virginia Tech, Department of Computer Science, analyzed two semesters of programming assignments, one in Spring

2001 and the other Spring 2003, with the latter being a pilot test of including TDD and Web-CAT, an automated grading system which assigns scores base on three criteria; code correctness, test completeness with respect to the code and test completeness and validity, as part of the teaching curriculum. It was observed that grades from Spring 2003 are slightly higher than those from Spring 2001, when graded by the traditional method, and significantly higher when graded by Web-CAT. In addition, it is also noted that there are significantly fewer test case failures from the master suite when compared against the code from Spring 2003 than when compared against the code from Spring 2001. Lastly, students in Spring 2003, who used TDD and Web-CAT submitted programs containing approximately 45% fewer defects per KLOC, as compared students in Spring 2001. There was no mention of any effect on productivity caused by TDD[8].

However, not all studies support the hypothesis that TDD improves code quality. In Erdogmus and Morisio's experiment at Politecnico di Torino, 24 third year computer science students, were split into two groups of 11 and 13 forming the Test-first group and Test-last group respectively. It was concluded that test-first did not achieve better quality on average, although it did achieve more consistent quality. The reason being that writing more tests improved the minimum quality achieved, reducing the variation. However, this effect is not specific to Test-first, rather it is a product of writing more tests. However, it is noted that Erdogmus and Morisio believe that Test-first does improve productivity in terms of number of test cases written per unit of programmer effort. In addition, it supports incremental development and writing of tests before implementation which encourages better decomposition and understanding of requirements[9].

Results of Erdogmus and Morisio's experiment agree with Mullar and Hagner's case study at the University of Karlsruhe. 19 computer science graduates participated in the case study, of which 10 formed the test-first group while the remaining 9 formed the control group which used traditional development process. All the students had just previously participated in a 1-semester graduate laboratory course introducing the XP methodology and they had a median programming experience of 8 years total. They were tasked to develop a GraphBase class in Java, which will be checked against a total of 20 JUnit test cases with 522 assertions. The results show that test-first does not improve productivity nor produce better quality code. However, it is observed that test-first seems to support better program understanding[22].

1.5.4 Survey Results on TDD

In addition to empirical studies on the effectiveness of TDD, multiple surveys have been done with both professionals and students to gauge the acceptance of TDD. In 2002, a survey of 32 respondents conducted across 10 industry segments, showed 14 firms using an Agile method, of these numbers, most were used in small projects lasting a year or less. In 2003, 131 respondents claimed they used an Agile method, and of these, 59% claimed to use XP and implied using TDD. Both surveys showed positive feedback regarding the application of agile methods, with results such as increased productivity and quality with reduced or minimal changes to cost. The surveys fail to mention the reasons why the remaining percentage of professionals chose not to adopt TDD[12].

George's survey of 24 professional pair programmers by North Carolina State University, 78% thought that TDD improves overall productivity in terms of better requirements understanding and reduction in debugging efforts. 80% thought TDD was effective in terms of higher quality code, simpler design and as an effective approach in general. However, 56% felt that getting into TDD mindset was difficult and 23% felt that the lack of upfront design in TDD was an obstacle[10].

Melnik at The University of Calgary and Southern Alberta Institute of Technology conducted a three year study of introducing agile methods in software engineering courses. Perceptions of the student body on eXtreme Programming(XP) and its individual practices were collected[19].

The following difficulties were noted by students throughout the three year study[19]:

1. Not used to test-first concept, difficult to write test cases before writing the code for functionality.
2. Logically confusing, as writing test first before functionality is like working backwards.
3. Testing involves too much work, and the short-term benefits are not apparent.

Despite the difficulties encountered by the students, more than 75% of them recognise the fact that TDD speeds up the testing process and believes that it improves quality of code.

Edwards' survey conducted at Virginia Tech, Department of Computer Science, on students who were involved in the pilot program introducing TDD and Web-CAT, an automated grading system, showed favorable reaction to TDD as

well. 65% of the students believed that TDD increases confidence in the correctness of their code. In addition, 67% believed that TDD increases confidence when making changes to their code. Lastly, 73.5% expressed that they would like to use Web-CAT and TDD for programming even if it was not a course requirement[8].

1.5.5 Conclusion on TDD Research

In most cases, in both academia and industry, TDD is shown to improve code quality. Another common agreement is that TDD improves productivity in terms of better understanding of the program. The common drawback observed is increase in development time. However, this does not necessarily mean loss of productivity as the increase in development time is due to development of test assets(i.e. test cases) which can be used in regression testing. Favorable feedback has been received with regards to acceptance of TDD in industrial or academic environment. However, several problems were highlighted which hinder the adoption of TDD. These include, the foreign concept of test-first and the failure to see short-term benefits in using the approach due to the additional effort required in producing test cases.

Tool Support for TDD

Janzen noted that TDD often assumes the existence of an automated testing framework, which will help simplify the creation and execution of unit test cases, and that for the implementation of TDD in Java, JUnit is essential[12]. In addition, it was observed by Jones[14] and in our own research that Java is the dominant programming language used in TDD studies, and that JUnit is the leading testing framework used[14]. In this chapter, we will look at JUnit as well as various projects such as JNuke and JML and how they improve upon the existing JUnit framework.

2.1 JUnit - An Automated Testing Framework

JUnit is a regression testing framework written Erich Gamma and Kent Beck. It is used by developers to implement unit tests in Java. Currently, it is an Open Source Software, released under the Common Public License Version 1.0 and hosted on SourceForge, a web repository of Open Source Projects[15].

JUnit defines how to structure your test cases by providing setUp and tearDown methods for setting up and cleaning up test fixtures. It also provides tools to run the test cases automatically and provides a report of test success or failure. Tests are implemented in a testclass which is a subclass of TestCase; a JUnit library class. The current practice is to place our testclass in the same package as the class to be tested, so that it is able to access package private methods[30].

A code example of testclass[30]:

```
public class MoneyTest extends TestCase{ —(1)
    public void testSimpleAdd(){ —(2)
        Money m12CHF = new Money(12,"CHF");
        Money m14CHF = new Money(14,"CHF"); —(3)
        Money expected = new Money(26,"CHF");
```

```

        Money result = m12CHF.add(m14CHF);
        Assert.assertTrue(expected.equals(result);—(4)
    }
}

```

Breakdown of the code example:

1. testclass MoneyTest extends TestCase, thus making it a subclass of TestCase.
2. testSimpleAdd() is a test implemented in the testclass MoneyTest.
3. test setup codes such as creation of objects for interaction, can be reused by storing the objects as instance variables of your testclass and initializing them by overriding the setUp method. The symmetric operation of setUp is tearDown, which you can override to clean up test fixtures after each test. Note that each test method runs within its own test fixture and JUnit calls setUp and tearDown for each test such that there's no side effects among test runs[30].
4. JUnit tests use assert statements to validate results of an action. There are many other types of assert statements, for example assertEquals and assertFalse. In this code example, assertTrue will trigger a failure to be recorded by JUnit if the argument given isn't true.

A code example of reusing setup codes[30]:

```

public class MoneyTest extends TestCase{
    private Money m12CHF;
    private Money m14CHF;
    private void setUp(){
        m12CHF = new Money(12,"CHF");
        m14CHF = new Money(14,"CHF");
    }
}

```

Running tests in JUnit can be done through the static or the dynamic way. In the static way, you override the runTest method inherited from TestCase and call the desired test case. A convenient way to do this is with an anonymous inner class. Each test must be given a name so that you can identify it if it fails.

An example of this is shown below[30].

A code example of a static way:

```
TestCase test = new MoneyTest("simple add") {
    public void runTest(){
        testSimpleAdd();
    }
}
```

In the dynamic way, create a test case to be run by using reflection, a feature of Java, to implement runTest. It assumes the name of the test is the test case method to invoke and dynamically finds the method and invokes it. An example of this is shown below[30].

A code example of a dynamic way:

```
TestCase test = new MoneyTest("testSimpleAdd");
```

To run the tests, we need to define a test suite. In JUnit, this requires the definition of a static method called suite. The suite method is like a main method that is specialized to run tests. We simply need to give the name of the class with the tests to the test suite and it will automatically extract and run the test methods. An example of this is shown below[30].

A code example of test suite using the dynamic way:

```
public static Test suite() {
    return new TestSuite(MoneyTest.class);
}
```

JUnit provides the framework for automated regression testing. However, it does not help generate unit test cases. The JUnit Eclipse plug-in, helps generate the method stubs for test methods as well as the setup and teardown methods, but does not help generate unit test cases either. However, using JUnit in Eclipse, the user need not setup test suites to run the test methods, instead, the user may run the test method by executing the testclass as a JUnit testclass. More information on how JUnit 3.8.1 works can be obtained at its official website, <http://www.junit.org/index.htm>.

2.2 Scaling up JUnit - JNuke Project

JUnit, which targets testing of individual method and classes, supports regression testing through fully automated execution of tests. However, JUnit fails to address several shortcomings of classical unit testing[2].

1. Functions that change the state of complex components are difficult to test individually. Sometimes, it is impossible to tell if a certain internal state is correct.
2. Some tests require a complex context that is difficult to setup.
3. Other tests rely heavily on other modules that are still under development.

In addition to the above issues, JUnit also fail to provide support for log and error log files, which Artho[2] asserts allows tracking the internal state of a component much more easily and is the key to larger-scale integration testing.

JNuke is a framework for verification and model checking of Java programs, which combines runtime verification, explicit-state model checking and counter-example exploration. As efficiency is crucial in dynamic verification, JNuke is written in C for improved performance and memory usage by an order of magnitude compared to other approaches and tools. JNuke provides functionality that extends beyond what is commonly available by unit test frameworks such as JUnit, allowing it to scale up to handle large-scale tests[2].

Features of JNuke includes the following[2]:

String Representation Imposed strict requirements on string representation of classes, i.e. their object state, following a Lisp-like representation, to allow it to be machine readable. This allows the internal object state to be analyzed at a glance as well as enabling it to be logged easily. The state of an object is defined by the values in its class fields.

Log Files Support for writing test results to a log file is provided. The log file containing results can be used to compare against a template output file containing expected results, this is called verified logging. In addition it allows the execution of large-scale tests in a unit test framework, as it allows the testing of large data structures by writing their content to a log file and than manually inspecting it. If the data is correct, the log file can be used as a template log file in the future. This prevents the need to codify each data element stored and simplifies the creation of complex test cases.

Memory Management When compiling with JNuke, size of each allocated memory lock is stored internally and validated when it is reallocated or freed. This extra information is used in each unit test to ensure the absence of memory leaks.

Coverage measurement Tight integration of coverage measurement (checking of statement flows, e.g. no unreachable blocks of code) with the GNU C compiler, allowed JNuke to fully automate coverage measurement across various platforms.

Automatic Indentation GNU indent is run on all source files prior to each CVS commit command, in order to facilitate editing of code and improve quality of the coverage output.

2.3 Scaling up JUnit - JML Project

Writing unit tests is a tedious and often difficult task. If testing code is written at a low-level abstraction, it may be tedious and time-consuming to revise it to match changes in the functional code[6]. While JUnit provides a framework that fully automates the execution of tests, it does not provide features to automatically generate unit test cases. In addition, JUnit requires the separate testing code to be developed and maintained in synchrony with the functional code under development, thus any changes in the functional code, requires the testing code to be reviewed in order to reflect the changes. As a result, the difficulty and cost of writing the testclass, especially during frequent code changes, increases the pressure not to write test code or not test as frequently as might be optimal[6].

Cheon and Leavens[6] came up with a JML support tool, which seeks to resolve the above problems by using Java Modeling Language (JML), to provide formal specifications of the requirements and thus generate test codes from them. The JML support tool shall be referred to as JML-JUnit tool, as Cheon and Leavens[6] did not name their tool.

JML is a formal behavioural interface specification language for Java. It allows the formal specifications of method pre- and postconditions based on requirements. These are written as a form of annotation comments on the Java source file, as shown below[6].

Example of JML Specifications:

```
/*@public behaviour
```

```
@requires n!= null && !n.equals("");
@assignable name, weight
@ensures n.equals(name) && weight == 0;
signals(RuntimeException e) false;
*/
public Person(String n){name = n; weight = 0;}
```

The JML-JUnit tool monitors the specified behaviour of the method using the JML runtime assertion checker, this is in contrast to conventional methods of comparing expected outputs versus actual outputs. A JUnit testclass is automatically generated from JML specifications. The generated testclasses send messages to objects of the Java classes under test, and catch assertion violation exceptions from test cases which are checked against an initial precondition check. Such assertion violation exceptions are used to decide if the code failed to meet its specification, and thus its expected behaviour. If the class under test satisfies its interface specification for some particular input values, no such exceptions will be thrown and the test succeeds[6].

JML-JUnit tool still requires the user to provide test data, however the generated testclasses make it easy for the user to add test data. In addition, only public, protected or package visible methods of the class are tested. In other words, no test methods are generated for private methods, as it is assumed that these maybe indirectly tested through the test of other methods. Constructors are not tested either as they are implicitly tested when creating the objects for interaction[6].

Figure 2.1: A Translation Scheme for Runtime Assertion Checking of JML Specifications[6]

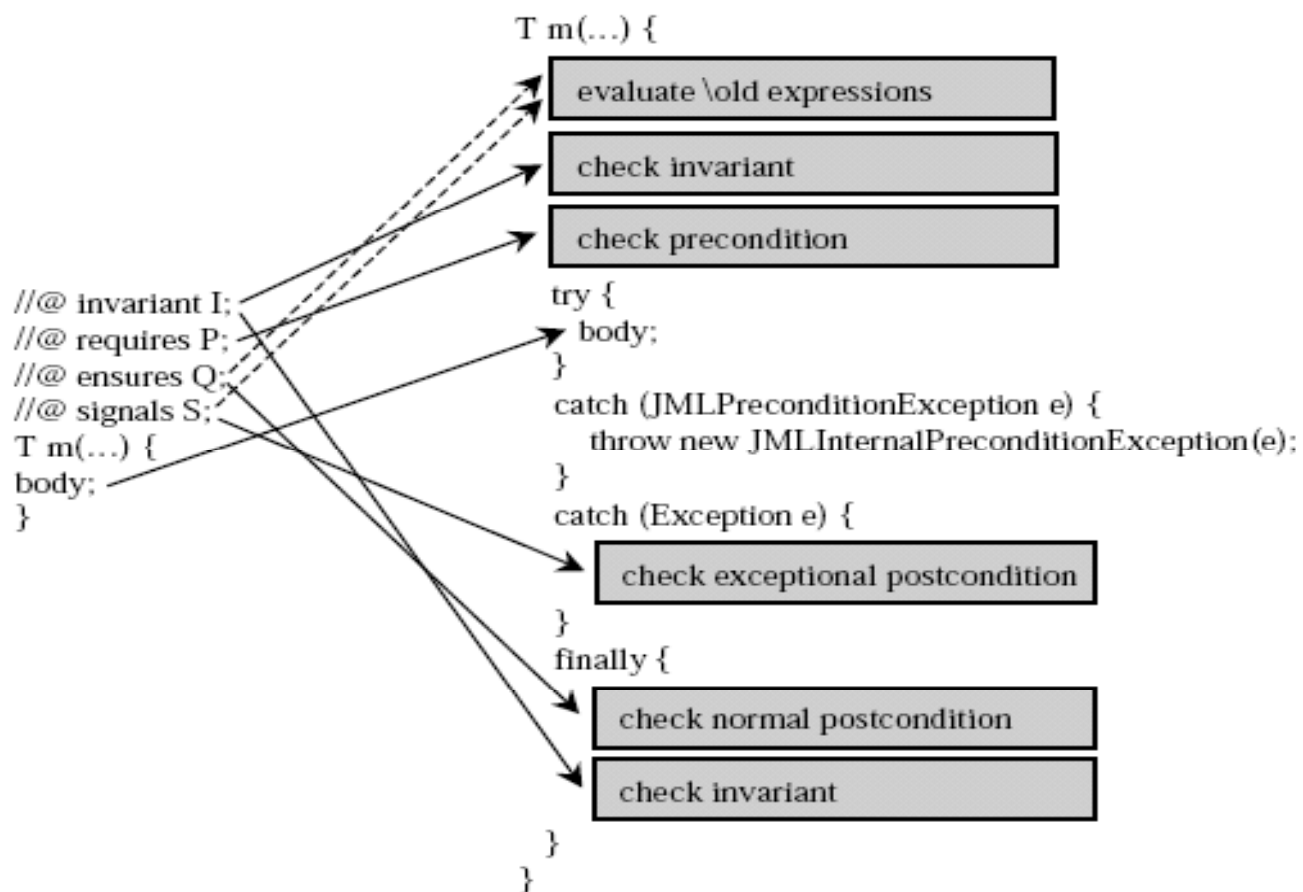


Figure 2.2: An Example of skeleton test code generated from JML specifications[6]

```
public void testM() {
    final A1[] a1 = vA1;
    ...
    final An[] an = vAn;
    for (int i0 = 0; i0 < testee.length; i0++)
        for (int i1 = 0; i1 < a1.length; i1++)
            ...
            for (int in = 0; in < an.length; in++) {
                try {
                    if (testee[i0] != null) {
                        testee[i0].M(a1[i1], ..., an[in]);
                    }
                }
                catch (JMLInternalPreconditionException e) {
                    String msg = /* a String showing the test case */;
                    fail(msg + NEW_LINE + e.getMessage());
                }
                catch (JMLPreconditionException e) {
                    continue; // success for this test case
                }
                catch (JMLAssertionException e) {
                    String msg = /* a String showing the test case */;
                    fail(msg + NEW_LINE + e.getMessage());
                }
                catch (java.lang.Exception e) {
                    continue; // success for this test case
                }
            }
        }
    }
```

2.4 Conclusion on Automatic Unit Test Tools

JUnit provides a framework for fully automating test executions, making it convenient to perform regression testing. However, it does not automate the generation of test cases and it is primarily built to support unit testing and does not scale well to larger scale tests, such as integration testing.

JNuke seeks to improve upon JUnit by providing features that facilitate the testing of internal object states as well as allowing it to perform larger scale tests, such as integration testing[2]. JML-JUnit on the other hand, provides support to allow the automated generation of test cases, thus reducing the effort required to build test assets. This is done by generating test cases from formal specifications defined by JML in the form of annotation comments in the Java source file of the class to be tested[6].

CHAPTER 3

Test-Driven Development Case Studies

3.1 Overview

In order to reinforce my understanding of Test-Driven Development and identify the type of support novice programmers need in order to apply TDD effectively, three experiments were conducted. This chapter describes the experiments and their results.

1. **Experiment 1 - Build a JUnit Testclass using TDD Approach**

Aims:

- To gain experience using TDD and JUnit.
- To identify areas which could be improved with regards to JUnit.
- To identify difficulties in using TDD approach for a first-time user, with myself as the test subject.

2. **Experiment 2 - Evaluate First Year Software Engineering (SE) Students' First Attempt with TDD**

Aims:

- To identify novice programmers' problems with using TDD, using a sample size of about 30 subjects.
- To identify and categorize the common errors made by novice programmers in their coding.
- To observe the effectiveness of Equivalence Partitioning and Boundary Value Analysis for Generating Unit Test Cases.

3. **Experiment 3 - Interview of Industrial and Academic Professionals**

Aims:

- To obtain professional opinions on TDD and its problems from various perspectives.

3.2 Experiment 1 - Build a JUnit Testclass using TDD Approach

3.2.1 Experiment 1 - Setup

A testclass was developed on a MobilePhone class, a workshop assignment for Year 2006 Semester 2's 1st year SE students. The choice of choosing this assignment was to allow the developed testclass to be used in experiment 2 which will be described in the later sections. The assignment was coded in Java and JUnit 3.8.1 was used to build the testclass. The testclass is run using the Eclipse IDE, which allows the execution of tests without setting up test suites.

JUnit test cases were first written in Eclipse without any guidelines. Subsequently, in order to generate a comprehensive set of unit test cases, two forms of unit testing methodologies were used. Equivalence Partitioning to segment the range of possible inputs for each method into subsets and applying Boundary Value Analysis on each subset such that the boundaries of each subset are chosen as test values.

Following TDD approach, the unit test cases are developed based on specifications provided, prior to any coding other than the class skeleton, such as method signatures etc, which will be used to generate test method stubs by Eclipse's JUnit Wizard. As I am mainly concerned with observing the difficulties of applying the "test-first" concept, this experiment did not include refactoring.

3.2.2 Experiment 1 - Results

Approximately 600 lines of test code were written compared to 150 lines of object implementation coding, this resulted in 39 unit test cases for 7 methods to be tested, excluding SET and GET methods. Out of the 600 lines of test code, roughly 20 lines of code was generated by the Eclipse's JUnit plugin, which provided the skeleton template of a JUnit testclass as well as test method stubs for each of the methods.

Advantages observed with TDD:

- Drives us to think of design issues, e.g. what input parameters are needed and what output is to be expected given certain inputs and specified behaviour from requirements specification.
- Allows instant feedback as to whether a method has been implemented as intended by the specifications, this also acts as a form of quality assurance, as the developer can be assured the method implemented is working before moving on.
- Pushes testing to the forefront, making it an integral and unavoidable part of the software development.

Disadvantages observed with TDD:

- Significantly more lines of code are written, due to the test cases written. This can be considered a decrease in productivity with regards to actual lines of code written for the application.
- Difficult to implement the test-first concept as the idea of writing tests before actual coding is foreign. It is difficult to imagine the types of test cases needed without having actual code to work with.
- Writing tests before any code is written limits us to black-box testing methodologies.

Advantages observed with JUnit 3.8.1:

- Makes it convenient to do regression testing.
- Provides support for automated unit testing.

Disadvantages observed with JUnit 3.8.1:

- No built-in support for testing object states.
- No built-in support for testing exceptions.
- No built-in support for accessing private class fields or methods.
- No built-in support for testing methods which do not return a value.

Observations with Unit Testing in General:

- Difficulty identifying when to stop writing test cases and start doing actual coding.
- Some test cases are inevitably left out, without proper guidelines to follow for generating test cases.
- Writing test cases requires creativity, there are no cookbook solutions.
- Equivalence Partitioning and Boundary Value Analysis, while efficient at reducing the number of input parameters to test, do not test for object states.

3.2.3 Experiment 1 - Conclusion

Using number of lines of code generated as an indication of effort and time spent in the development of the application, a 4:1 ratio was observed between lines of code of test cases and lines of code of object implementation. This shows that a significant amount of effort and time spent in development is due to generating test cases. This experiment also highlighted areas where JUnit could be improved upon, to make the testing process easier. A tool that makes up for the deficiencies of JUnit and helps generate test cases, would be useful in reducing the increased amount of effort and development time as a result of using TDD.

Lastly, several problems have been observed with regards to writing unit test cases for TDD, with Equivalence Partitioning and Boundary Value Analysis. These problems can be addressed in the form of a tutorial aimed to provide guidelines to writing test cases.

3.3 Experiment 2 - Evaluate First Year Software Engineering (SE) Students' Attempts with TDD

3.3.1 Experiment 2 - Setup

A sample size of 29 1st year SE students' source code of the MobilePhone class written in Java, developed using TDD approach was tested against the master testclass of 39 JUnit test cases developed in experiment 1. For the purpose of this experiment, all private fields and methods are changed to protected to facilitate the testing, as JUnit 3.8.1 does not provide any in-built mechanism to access private fields and methods. In addition, the test cases in the master testclass

have been scaled up to include checks on class fields where changes are expected, i.e. checking the object state.

Errors of the students' codes are noted and are classified into categories. According to Lutz [18], program faults, i.e. errors, are classified according to three levels as follows:

1. Internal Faults, e.g. syntax errors.
2. Interface Faults, e.g. interactions with other systems such as transfer of data or control.
3. Functional Faults, e.g. operating faults, conditional faults or behavioural faults.

In this experiment, we are concerned with identifying and classifying functional faults. Functional faults are further broken down as follows [18]:

Functional Faults

1. Operating Faults
 - (a) Omission of functions
 - (b) Unnecessary functions
2. Conditional Faults
 - (a) Incorrect Conditions, e.g. conditionals such as If-Else statements, For loops, While loops etc are setup incorrectly.
 - (b) Incorrect Limit Values, e.g. boundaries of loops or input parameters are incorrect.
3. Behavioural Faults
 - (a) Incorrect Behaviour, e.g. functions not performing to specifications.

3.3.2 Experiment 2 - Results

Despite all student's source codes passing their own test cases, errors were uncovered when ran against the master testclass. A total of 1131(29 sample codes x 39 JUnit tests) tests ran, uncovering a total of 389 errors.

Figure 3.1: Analysis of 29 Students' Passing Rate for 39 Tests

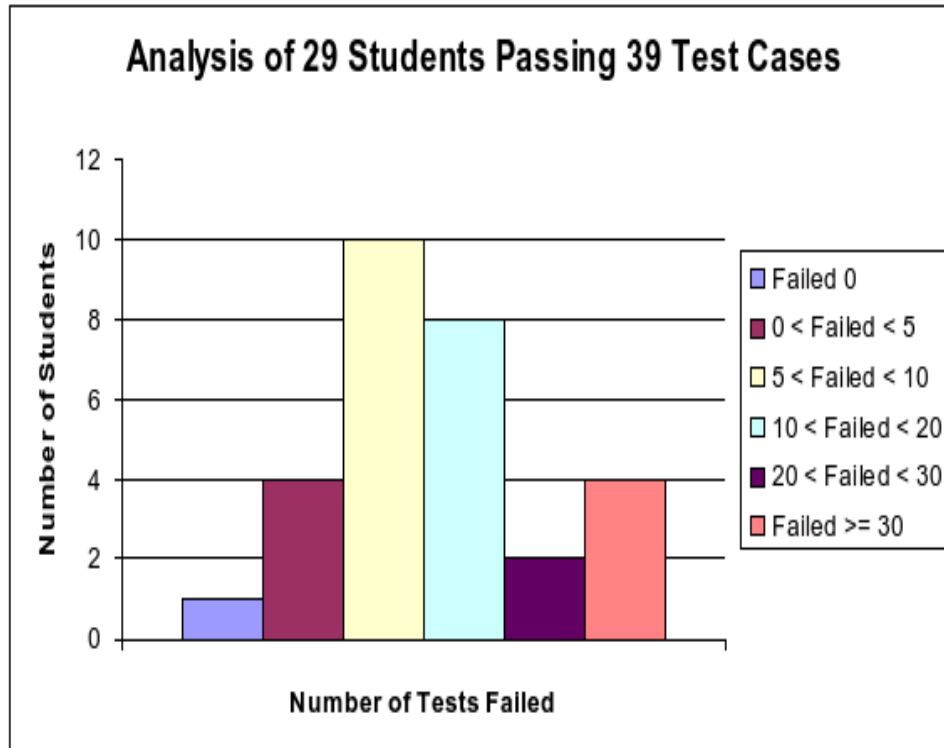


Table 3.1: Analysis of Errors Found

Analysis on Effectiveness of Test Cases	Results	Percentage
Average number of errors captured by the master testclass	13	30%
Analysis on Errors Found		
Total number of tests failed out of total tests ran	374	33%
Operating Faults		
Omission of functions	129	33%
Conditional Faults		
Incorrect Conditions	69	18%
Incorrect Limit Values	81	21%
Behavioural Faults		
Incorrect Behaviour, not conforming to specifications	110	28%

Examples of Errors Found

Omission of functions For example, a student neglected to initialize the field array *messages*, this resulted in a failure in test case 10, which attempts to store a message at the lowerbound of the *messages* array. Other incidents of omission of functions, includes neglecting to implement functions stated in the requirements specification.

Incorrect Conditions For example, a student incorrectly set the conditions of a For loop, in method *deleteMessage* as shown below. Note that the field *messageID* starts from 1 whereas the index for the field array *messages* starts from 0.

```
public void deleteMessage(int messageID){
    for(int i=messageID; i<this.messageCount+1; i++){
        this.messages[i-1] = this.messages[i];
    }
}
```

As shown, the For loop is initialized to end at *this.messageCount+1*, this causes the loop to reach index 21 if *messageCount* is 20, meaning that *messages* array was full. Trying to access *messages* at *i = 21* caused an

error. This error was uncovered by test case 34, which attempts to delete a message with *messageID* 19 from a *messages* array which was full. The input parameter 19 was chosen as it is 1 less the upperbound of *messageID* as suggested by the guidelines of Boundary Value Analysis.

Incorrect Limit Values For example, the same student which made the mistake above incorrectly, allowed *messageID* 0 to be accepted for method *deleteMessage* as shown above. This caused an error when the method tried to access *messages* at $i = -1$. This error was uncovered by test case 27, which tried to attempts to delete a message with *messageID* 0. The input parameter 0 was chosen as it is 1 less the lowerbound of *messageID* as suggested by the guidelines of Boundary Value Analysis.

Incorrect Behaviour For example, a student allowed the topup of a negative amount for the method *topUp*, as shown below.

```
public void topUp(int amount){
    this.credit += amount;
}
```

This error was uncovered by test case 8, which passes in a negative value for *amount*, and attempts to top up the credit by a negative amount. Other examples of incorrect behaviour include providing the wrong formatting for the *toString* method.

3.3.3 Experiment 2 - Conclusion

The total number of errors found, 389, exceeds total number of tests failed, 374, as some tests caught more than 1 error. As can be seen from the analysis above, the majority of errors found are conditional faults, followed by omission of functions and lastly incorrect behaviours.

Several conclusions can be drawn from this experiment. Firstly, Equivalence Partitioning, coupled with Boundary Value Analysis and taking into consideration of object states, generates a comprehensive set of test cases capable of capturing high percentage of errors. Secondly, most students require guidance in generating test cases. Out of 29 students, only 1 managed to pass all tests and only 5 failed less than 5 tests. Thirdly, the high number of omission of functions, could be due to students spending most of their time writing test cases rather than actual coding, which in turn could be due to their lack of experience working with JUnit or Java. Some tool to reduce the time needed to generate test cases would be useful in such a scenario.

3.4 Experiment 3 - Interview of Industrial and Academic Professionals

3.4.1 Experiment 3 - Setup

For this experiment, 4 individuals from varying backgrounds were approached separately for a fifteen minute interview on their opinion on TDD. The aim of these interviews is to identify the professional opinions on TDD, such as whether it is beneficial to use TDD and what problems would programmers encounter in adopting such an approach.

Interview Questions

1. What are your opinions on TDD, i.e. do you support or are you against TDD? Why?
2. What are the problems programmers encounter when using TDD?
3. What are the problems with generating test cases for TDD?
4. What benefits do you see with using TDD?
5. Are guidelines for generating comprehensive test cases useful when applying TDD?
6. Would a tool for automatic test case generation be useful for TDD?

Interview Participants

1. Dr. Rachel Cardell-Oliver (R), Deputy Head of School at The University of Western Australia.
2. Mr. Kevin Vinsen (K), System Architect at Thales Australia, a provider of system, products and services for defense, security and civil markets.
3. Mr. Pierre Morel (P), I.T. Director at Asiatravel.com Holdings Ltd.
4. Dr. Terry Woodings (T), Adjunct Senior Teaching Fellow at The University of Western Australia.

3.4.2 Experiment 3 - Results

Question 1.

Both K and T are advocates of TDD, K being a practitioner of TDD and T believes TDD is beneficial particularly when used in conjunction with other techniques. R and P stand neutral on TDD, as both do not have personal experience in using TDD but have heard about it.

Question 2.

K and R believes that most don't see the short-term benefits in using TDD due to increase testing overheads and are reluctant to practice it. In addition, from her experience in teaching, R realised students don't have a good idea on building test cases, which is needed to use TDD effectively. T pointed out that most do not know how many test cases are "enough" and that building comprehensive test cases is hard to achieve. P, believes that the effectiveness of TDD depends on how well specified the requirements of the project are. He also pointed out that in the industry, deadlines are very important, and that the increased overheads in testing caused by using TDD is a concern.

Question 3.

K stated that programmers under his management, did not have problems with TDD's test-first concept and that there are tools out there that aid programmers in developing test cases. T stated that some of the difficulties with testing is finding test assumptions as early as possible and making test cases as comprehensive as possible. R remarked that beginner programmers (i.e. 1st or 2nd year students) might have problems with TDD's test-first concept and that beginner programmers do not have a good grasp on building test cases. P feels that the test-first concept of TDD can be logically confusing.

Question 4.

K believes the real benefit of TDD, is that it helps cope with inevitable changes in software development. T and P believes that TDD is beneficial as it forces programmers to focus on design issues by bringing test case design to the forefront. R remains skeptical on any benefits on TDD. While she believes that the test assets developed using TDD, might be useful as a form of documentation, she has not seen any student programmers benefiting from using it.

Question 5.

K and P believes that experience programmers would not have problems designing test cases or with the test-first concept of TDD. R, P and T agrees that it is useful to beginner programmers who do not have the experience necessary in

identifying likely errors in programs.

Question 6.

All of them believe that tools that help generate test cases is useful as they help reduce development time, however K and T point out that there are already existing tools that perform such a task.

3.4.3 Experiment 3 - Conclusion

From the feedback obtained in the interviews, several conclusions can be drawn. Firstly, common problems identified in TDD is that most users do not see the short-term benefits of TDD and that the increase in overheads in testing is of concern. Secondly, tools that help generate test cases would prove useful, however it is noted that there are already such tools out there[6]. Lastly, a tutorial on how to generate test cases would be useful to beginner programmers.

UnitGen Implementation

4.1 Rationale

In this project, I developed two support tools for TDD, in order to aid students in developing unit test cases.

1. A tutorial to guide students on generating a comprehensive set of black-box test cases. It also includes a user manual on UnitGen, the second tool.
2. UnitGen, an automatic test case generation tool, which accepts user inputs and helps generate a JUnit 3.8.1 testclass with test methods built from inputs provided.

The tutorial is essentially a support tool for UnitGen as it guides users on how to use UnitGen and choosing appropriate test values for creating test cases. Despite this however, the tutorial can be used independently, as the guidelines provided in it are useful for learning about testing in general. The rationale behind each of these two tools are described below.

4.1.1 Tutorial

It is observed in a study by Mugridge[14], that for TDD to be effectively used by students, education curriculum needs to cover areas such as tests and test case writing, as well as the use of automated testing frameworks. In addition, Smith[27] noted that testing is a subject often overlooked in typical computer science curriculum. Lastly, in my own experiments, section 3.2 and 3.3, it is noted that students require aid in generating a set of comprehensive test cases and that traditional testing methodologies such as Boundary Value Analysis and Equivalence Partitioning do not cover the testing of object states, which is essential in testing object-oriented environments.

Melnik[19] and George[10], also noted that students who practice TDD, encountered problems such as finding it difficult to write test cases before coding, as it felt like working logically backwards. They also did not know how many test cases are sufficient before beginning to starting implement coding.

Thus to cover the oversight in the education curriculum and tutor students in the use of our tool UnitGen, allowing students to be able to use UnitGen and practice TDD effectively, I wrote a tutorial on testing, titled, "Object-Oriented Software Testing at the Unit Level for TDD".

Since test case writing is a creative effort and selecting appropriate test data is a difficult and non-algorithmic process[24, 25], my tutorial seeks to provide guidelines instead of fixed steps to test case writing by pointing out actions that a student should do, or considerations that a student should think about, in order to generate a set of comprehensive unit test cases.

4.1.2 UnitGen

It is observed that using TDD results in more test cases written thus an increase in development time[10, 4, 9]. As a result of the additional effort and time required in writing the test cases, many do not see the short-term benefits in practicing TDD[19]. However, I believe that with the aid of automated test case generation tools and testing frameworks, this problem can be mitigated.

Since it is commonly assumed that TDD requires the support of an automatic testing framework[12], and that it would take too much time and resources to developed an automated test case generation tool or testing framework from scratch, I decided to improve upon an existing framework, JUnit.

4.2 Tool Design

4.2.1 Tutorial

Information used to write my tutorial, Appendix B, is obtained mainly from Myers' "The Art of Software Testing"[24], and my own observations on testing gained from experiments described in section 3.2 and 3.3. It also includes a user manual to teach students how to use UnitGen.

My tutorial is aimed to cover the following areas in testing

1. Testing Mindset and Principles.

2. Preprocessing Steps to generating Comprehensive Unit Test Cases. These are the steps the user have to go through before beginning to generate the unit test cases.
3. Steps to generating Comprehensive Unit Test Cases. These are the steps the user has to go through in order to generate a set of comprehensive unit test cases. This is further divided in three sections, guidelines on Equivalence Partitioning, guidelines on Boundary Value Analysis and lastly, considerations that needs to be taken into account when testing object-oriented systems.
4. Postprocessing Steps for generating Comprehensive Unit Test Cases. These are guidelines on what the user should consider after generating the first set of unit test cases. It highlights the refactoring concept in TDD and what it means to testing, as well as areas in coding that are error prone that the user should pay further attention to.

4.2.2 UnitGen

UnitGen's purpose is to automatically generate a JUnit 3.8.1 testclass with test methods. Although it is difficult to generate test inputs due to the process being non-algorithmic[25], it is possible to automatically generate test cases (i.e. test methods), that invokes methods for testing, using user-supplied input parameters. The success or failure of test cases is determined by two things. Firstly, if method gives an output, a comparison between the user-supplied expected output against actual output of the method. Secondly, a comparison between user-supplied expected state of the object against the actual state of the object after method execution. As UnitGen creates the entire JUnit testclass automatically, users need not be familiar with how JUnit works in order to test their classes using the JUnit framework.

To facilitate the process of accepting test values, e.g. method inputs and expected outputs, from user, a graphical user interface is built. This GUI wizard directs the tester to provide the necessary information required from the user, from which it will generate a JUnit testclass based on the information provided. In order to reduce the amount of information that the user needs to provide in order to generate the JUnit testclass, UnitGen, employs Java reflection to obtain information on the class under test.

UnitGen, incorporates ideas from JNuke, by providing logging facilities, which provides a documentation output, i.e. a test data file (logfile) that contains information on test cases generated. However, the format of the test data is

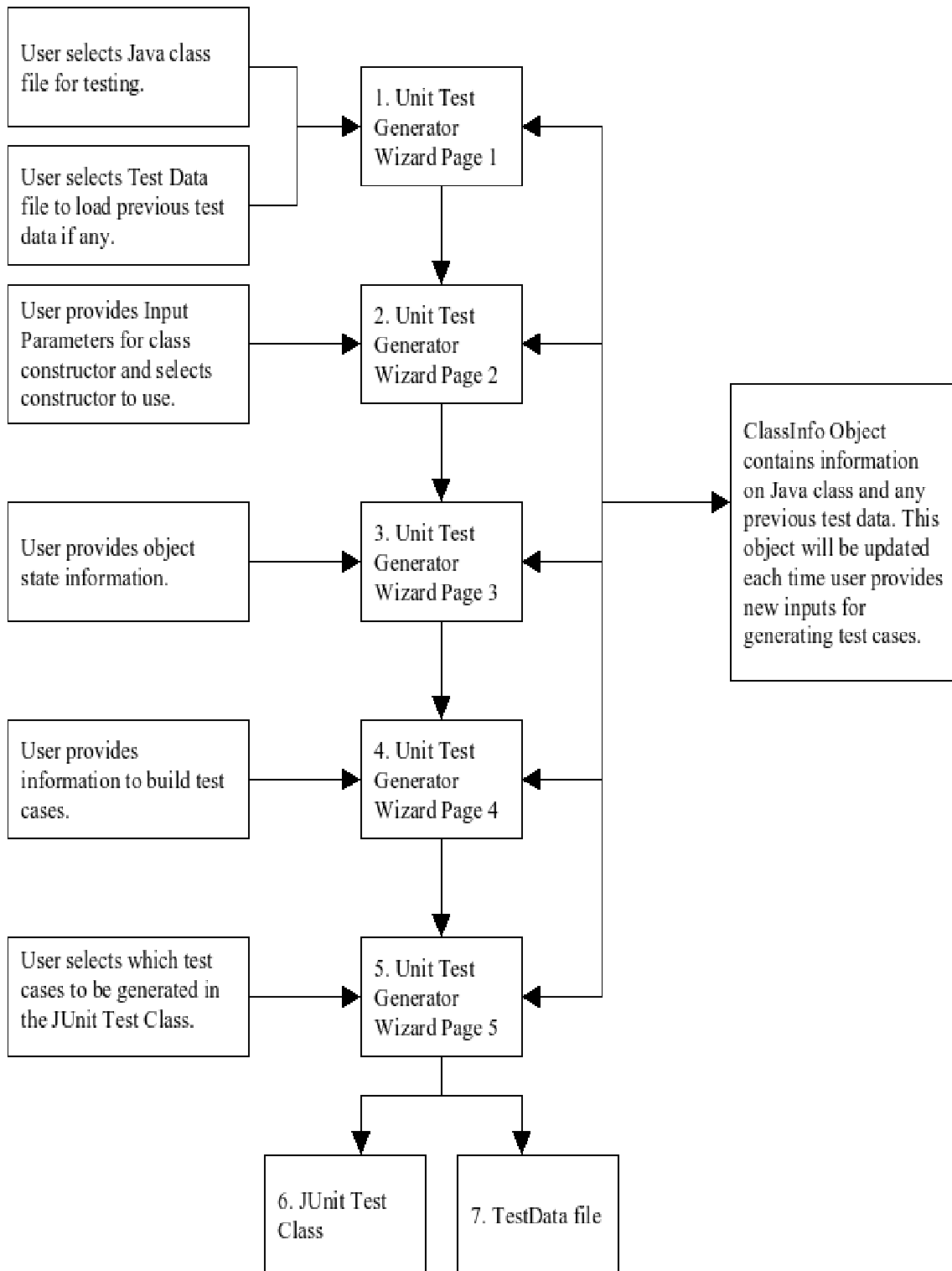
designed to be convenient for UnitGen to read and is not very readable to humans. Users can use the logfile as a test documentation much like JNuke[2], after some formatting. Users can also load it back into UnitGen to reuse previously created test cases. It is also possible for experienced users to input test cases directly into the logfile and generate the test methods using UnitGen, rather than going through UnitGen Wizard to create test cases. In an education setting, teachers can predefine object states for students to use, thereby further reducing the time needed to create test cases by students. This might encourage students to be more receptive to the idea of testing and using TDD as the effort required is reduced.

However, UnitGen, differs from JNuke in the way it handles examination of internal object states, defined by the values held in the class fields. Instead of providing strict requirements on string representations of Java classes, UnitGen uses reflection, a feature of Java, to examine internal object states to ensure that object state remains consistent, i.e. class fields only reflect expected changes, after method execution. This saves the tester effort in overriding the *toString* method of Java classes, in order to conform to the strict requirements of JNuke. Tester also need not resort to “dirty coding”, i.e. changing private class fields temporarily to public or protected for the sake of testing. In addition, through the use of reflection, UnitGen allows testers to define certain states of the object to focus on for testing, e.g. testing method execution when a certain array is full or empty.

While JML-JUnit assumes that the testing of private fields and methods should be done indirectly through other public or protected methods, I believe otherwise. Testers should be allowed the option to test private methods, as there are situations where the tester would wish to test complex private methods before using them. This also makes debugging easier, as any error found will be isolated to the private method being tested. Thus UnitGen, unlike JUnit, allows private fields and methods to be accessible for testing by generating the necessary Java reflection code when creating the JUnit testclass.

Unlike JML-JUnit, which examines the behaviour of a method, defined by JML specifications, in order to determine the success of a test case, UnitGen uses the classical method of comparing expected outputs versus actual outputs. The reason being that the concept of providing inputs and then comparing expected outputs versus actual outputs is easier to grasp for students compared to requiring them to learn how to construct formal specifications to define behaviours as required for JML. UnitGen, however is similar to JML-JUnit, in that it does not test constructors as it is assumed that constructors are implicitly tested when objects are created for interaction.

Figure 4.1: Overview of UnitGen Process Flow



Java Reflection

As mentioned earlier, UnitGen employs Java reflection, a feature of Java, for the dynamic retrieval of class information from a Java class file. The following information is retrieved by UnitGen from the supplied Java Class file using reflection:

1. Package name
2. Class name
3. Class Fields and their
 - (a) Access Modifier
 - (b) Type
 - (c) Name
4. Class Constructors and their
 - (a) Input parameter types
5. Class Methods and their
 - (a) Access Modifier
 - (b) Name
 - (c) Input parameter types
 - (d) Exceptions thrown
 - (e) Output Parameter type

Information retrieved from the Java Class file is stored into an object called `ClassInfo`, that holds all information on the class under test, including any inputs given by the user for creating test cases.

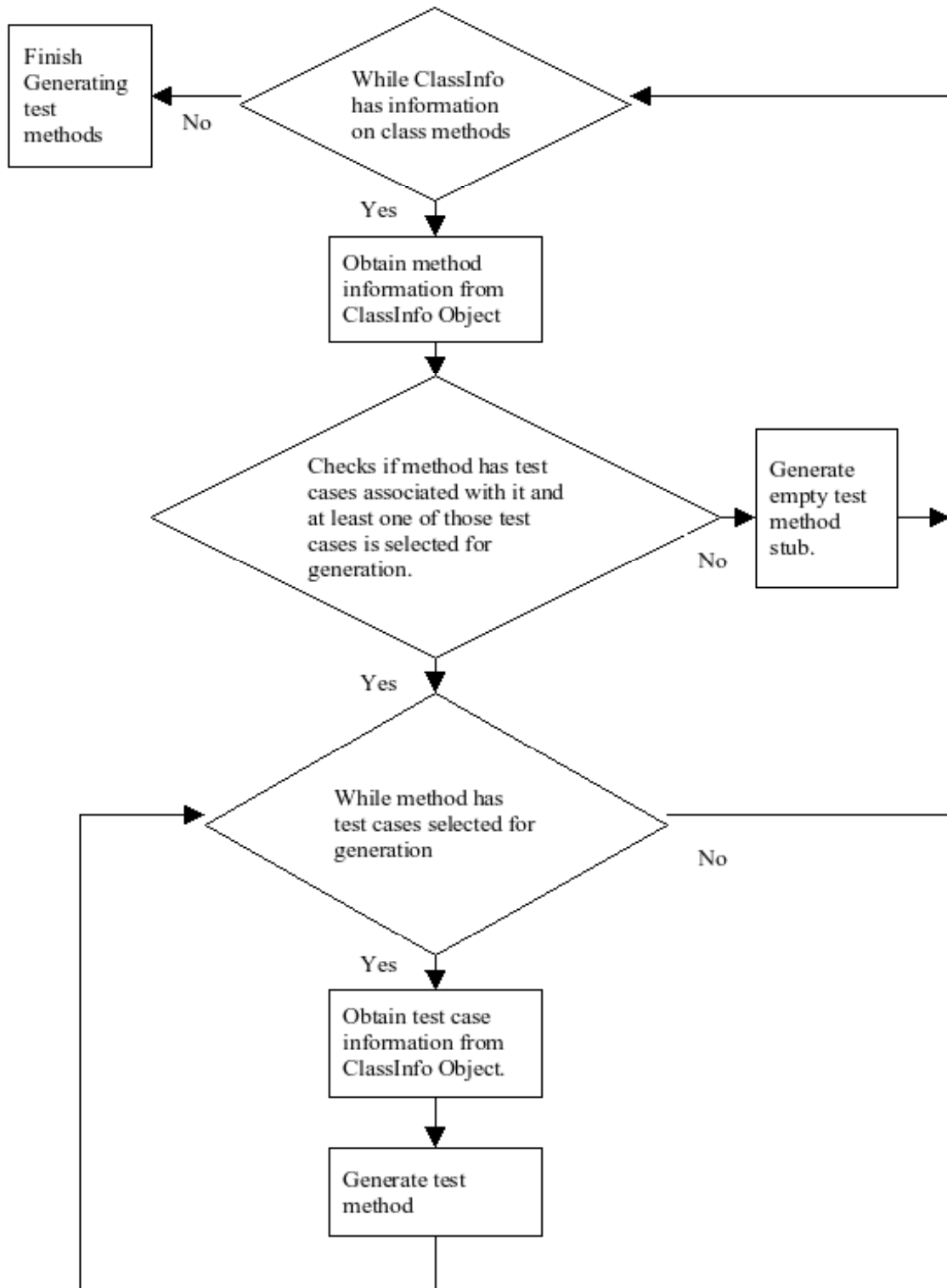
The drawback of using reflection to obtain class information, is that UnitGen is unable to retrieve the names of input parameters required by methods or constructors. As a result, while UnitGen is able to inform user that a method requires an integer, it is unable to inform user what the integer represents to the method. Users of UnitGen, must be clear about the specifications of the method they are testing, else it might be confusing, especially when a method requires several inputs.

JUnitClassGen

A key component of UnitGen is the JUnitClassGen class, which uses the information stored in the ClassInfo object to make decisions on what codes are required to be generated to build the output JUnit testclass. It consists of six main methods as described below.

1. **genClassHeader** - generates the necessary import statements, JUnit Class signature, Class constructor as well as the necessary class fields, i.e. private fields that is required to run the tests.
2. **genSetUp** - generates the JUnit setUp method, as well as any test fixture setup code required. E.g. instantiate the necessary objects required for test interaction, as well as the Class object necessary for applying reflection.
3. **genTearDown** - generates the JUnit tearDown method, as well as any test fixture tear down code required.
4. **genSetState** - generates the UnitGen private helper method, setState, that accepts the state of an object, in the form of an object array and the object which the state is to be applied to. This method employs reflection to initialize the object to a state specified by the UnitGen user to allow for testing specific situations.
5. **genGetState** - generates the UnitGen private helper method, getState, that accepts an object from which to obtain a string representation of the object's state. This method employs reflection to allow UnitGen user to obtain a string representation of the the state of the object passed in, without having to resort to provide a custom toString() method or changing class fields to allow access. With a string representation of the object state of the object under test, user can easily check the state of object to ensure that the method performs as expected.
6. **genTestMethods** generates the test methods in the JUnit testclass, based on the test cases created and selected for generation by UnitGen user.

Figure 4.2: Overview of genTestMethods Process Flow



To generate the proper test methods the following situations are considered:

1. Is the access modifier of the method to be tested, private? If so, a try and catch clause has to be created and the necessary reflection code generated to invoke the private method for testing.
2. Is the method expected to throw any exceptions? If so, the necessary try and catch clauses have to be created to handle those exceptions.
3. If the method is expected to throw exceptions, is the test case to be generated for the method, an invalid or valid test case? If invalid test case, an `assertFail` statement has to be created within the try clause, and other assert checks to be created in the catch clause. If valid test case, the opposite is true, and an `assertFail` statement has to be created in the catch clause, with the other assert checks within the try clause.
4. Is method expected to give an output? If so, actual output is to be checked against expected output provided by user. Whether method gives an output or not, a check is made to compare actual object state after method execution against expected object state provided by user.
5. Is method expected to require input parameters? If so, the method is invoked with the necessary inputs provided by the user.

Constraints of UnitGen

UnitGen, is not able in most cases to test methods which accept or returns other user-defined objects, unless the method only requires information from the object which can be supplied to the object by the object's constructor. UnitGen, is also not able to test if printouts to a console or screen are correct.

For details on how to use UnitGen, screenshots of the user interface, as well as examples of generated source codes and Test Data File please refer to the Tutorial attached in Appendix B.

CHAPTER 5

Tool Evaluation

In this chapter, I evaluate UnitGen and its tutorial against the aims of this project:

1. Reduce development time, by reducing the effort needed in developing test cases.
2. Help students see the short-term benefits in TDD, by reducing the effort required in testing.
3. Help students with the foreign-concept of Test-First, by providing a tutorial which guides students on selecting suitable testing parameters for use with the UnitGen tool.

I ran two practical workshops on TDD and UnitGen with third year SE students, to obtain qualitative results from surveys on their experience with UnitGen and its tutorial. I also performed further experiments to obtain quantitative results on the effectiveness of UnitGen, by comparing it against Eclipse's JUnit wizard with two fourth year students as test subjects.

5.1 Experiment 4 - Two workshop sessions on TDD and UnitGen

5.1.1 Experiment 4 - Setup

A ten to fifteen minute demonstration of UnitGen is conducted in two workshops consisting of a total of seventeen third year Software Requirements and Project Management (SRPM) students. After the introduction, they are given about half an hour with JUnit on Eclipse followed by another half an hour to have some hands on experience using UnitGen. They are also encouraged to

read the Tutorial written by me, to aid them in generating test cases. At the end of each workshop, students are given a survey form to complete and give a rating of 0 to 10 based on a rating system shown below. The SRPM tutorial materials for the workshops can be found on the SRPM unit webpage: <http://undergraduate.csse.uwa.edu.au/units/CITS3220/practicals/SRPM.Practical.7.2007/>.

Figure 5.1: Survey Questions Rating System

Rating	Experience Questions	Tool/Tutorial Questions (Degree of Usefulness)	Tool/Tutorial Questions (Degree of Ease of Use)
0	None	Useless	Impossible
1-3	Beginner	Useful but I won't use it as it is not worth the effort.	Difficult
4-6	Average	Useful and I might use it depending on circumstances.	Average
7-9	Good	Very Useful, I would use it if I can.	Easy
10	Expert	Very Useful, I would definitely use it.	Very Easy

Survey Questions

1. Your experience in Java Programming
2. Your experience in Unit Testing
3. Your experience in Junit
4. How would you rate the usefulness of UnitGen?
5. How would you rate the ease of use of UnitGen?
6. If you have used other unit test generation tool before, how would you compare the usefulness of UnitGen against it?
7. How would you rate the usefulness of the Tutorial?
8. How would you rate the ease of use of the Tutorial? E.g. is it easy to understand?

5.1.2 Experiment 4 - Results

For the survey results on UnitGen and my tutorial, if a bar is not shown in the chart, it means that student did not give a rating.

Figure 5.2: Survey Results on Programmer Experience

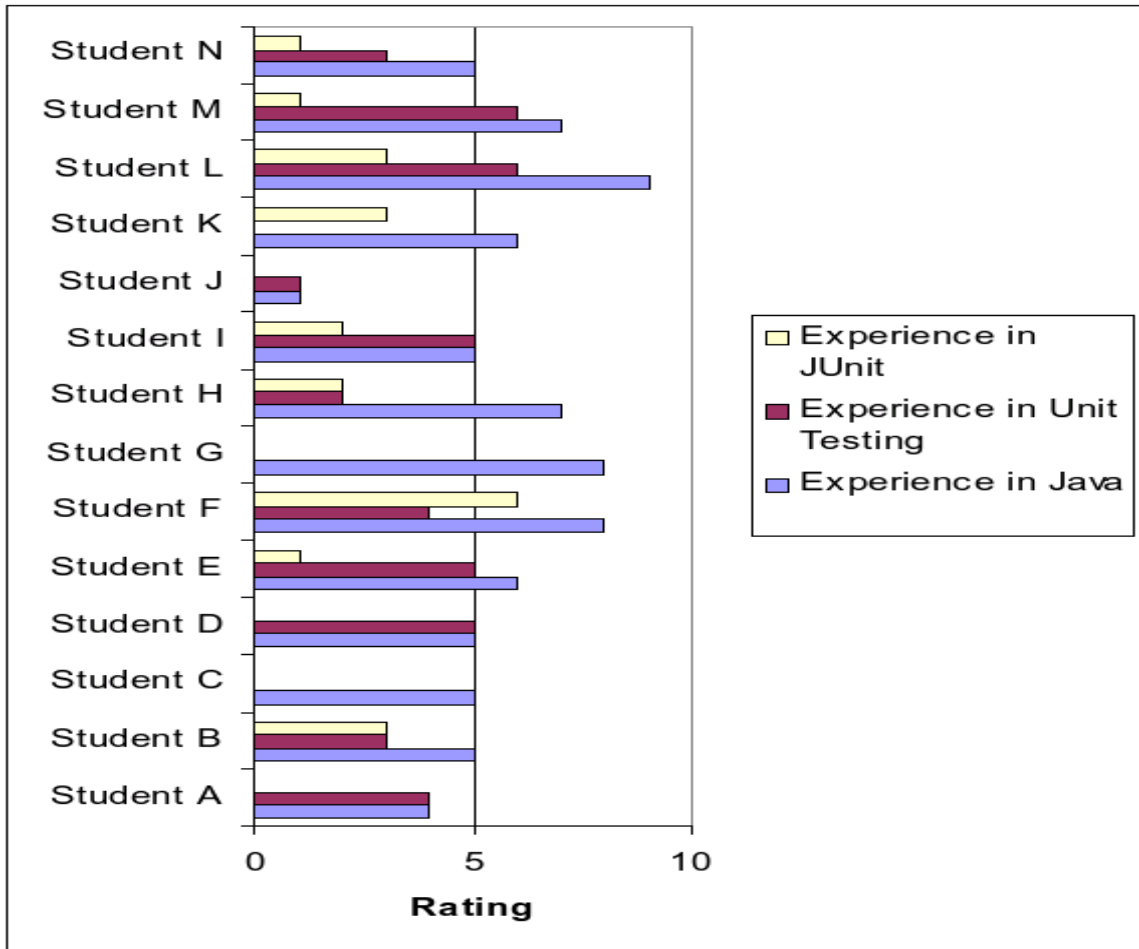


Figure 5.3: Survey Results on UnitGen and my Tutorial

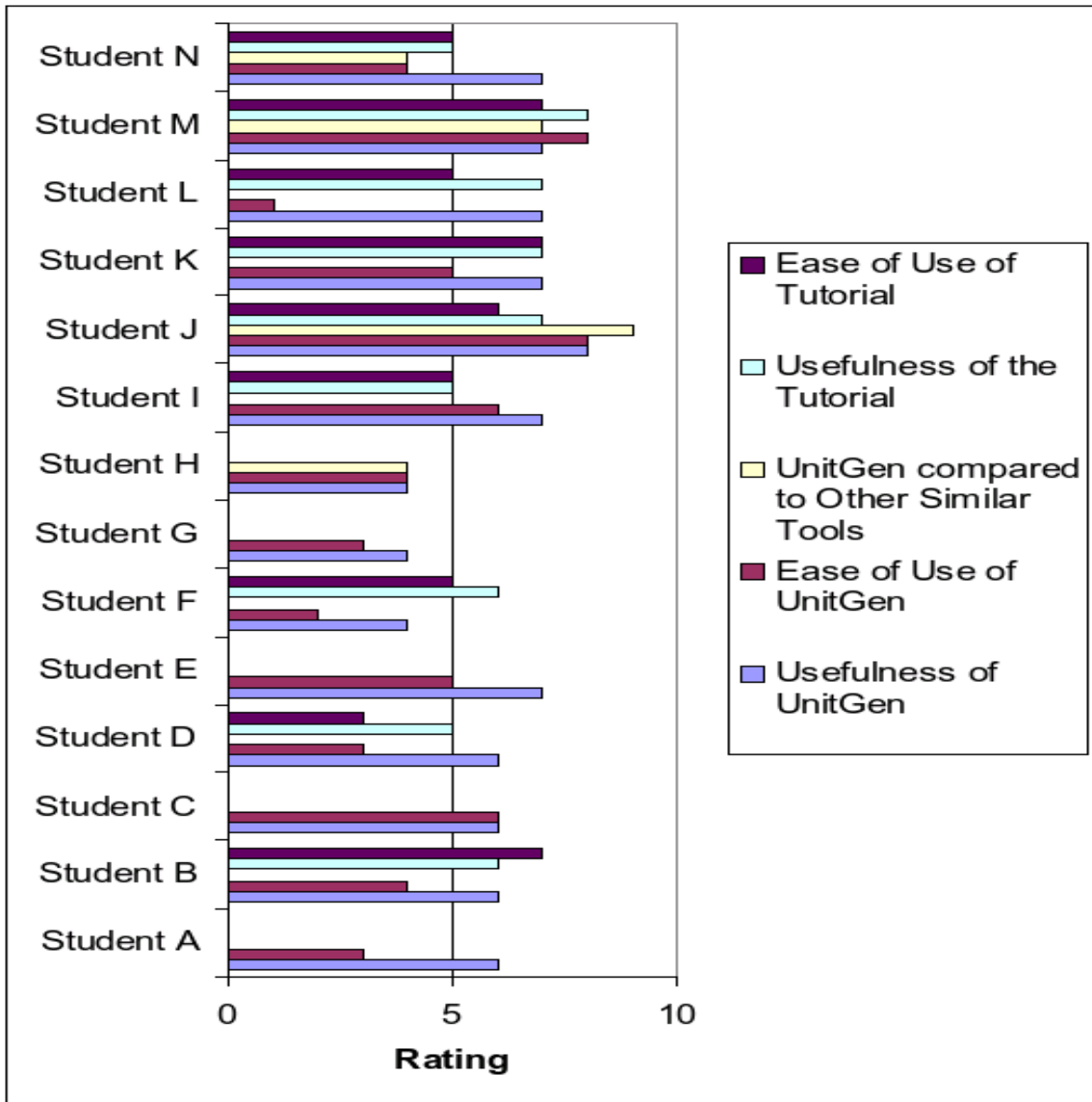


Table 5.1: Survey Feedback Analysis

Question No.	No. of Responses	Ave. Rating	Min. Rating	Max. Rating
1.	14	5.8	9	1
2.	14	3.1	6	0
3.	14	1.6	6	0
4.	14	6.1	8	4
5.	14	4.5	8	1
6.	4	6	9	4
7.	9	6.2	8	5
8.	9	5.5	7	3

5.1.3 Experiment 4 - Conclusion

Most of the participants in the survey are fairly experienced with Java. However few have much experience in unit testing or Junit.

The feedback from the fourteen survey forms returned showed a mixture of opinions on UnitGen and my tutorial. Opinions on the usefulness of UnitGen is very positive with most students finding it useful, and consider using it depending on circumstances and the rest finding it very useful and would like to use it. However, with regards to its ease of use, while most found its usability average with a minority finding it easy to use, there are some who actually found it difficult to use. Only four students have experience with other similar tools, with all of them finding UnitGen more useful. However, they did not comment on the other tools that they had experience with.

Of the nine students who read my tutorial, five found it useful and the remainder found it very useful. The positive feedback could be due to the fact that all of them only have beginner to average amount of experience with unit testing.

A couple of students pointed out that UnitGen needs to be more user-friendly. One of them suggested that a more comprehensive written help is needed to understand UnitGen. Another stated that not enough time was given to evaluate the tool. Several students expressed that it made life easier when doing testing. It was also pointed out that without names of the input parameters for methods and constructor, it can be confusing to the user as to what input values to provide.

5.2 Experiment 5 - Further Experiment on UnitGen versus JUnit

5.2.1 Experiment 5 - Setup

To further evaluate the performance of UnitGen against JUnit, two volunteers were found with similar amount of experience in Java programming. Both participants of the experiment are in their honours year and Java is their preferred programming language. Neither participant has had prior experience with JUnit and UnitGen.

Experiment Outline

1. 30 minutes of detailed explanation on both JUnit and UnitGen is given to both participants by me.
2. 30 minutes to write as many test cases as possible manually, using JUnit plug-in on Eclipse to generate a skeleton JUnit testclass, to test JUMobile, a variation of MobilePhone class used in my earlier experiments, described in section 3.2, that contains seeded faults.
3. 30 minutes to write as many test cases as possible using UnitGen to test UGMobile, another variation of MobilePhone class, that contains seeded faults different from those of JUMobile.
4. After the session, they are given the same survey form given to the SRPM students in the earlier experiment, described in section 5.1.

The following metrics were used in evaluating the performance of UnitGen against JUnit:

1. Number of source lines of code written - measures effort needed to create testclass.
2. Number of test cases generated - measures performance in terms of number of test cases created in a fixed period of time.
3. Number of errors found - measures the quality of testclass generated.

5.2.2 Experiment 5 - Results

Table 5.2: Results of UnitGen vs. JUnit on Eclipse

Tool Used	No. of Errors Found out of 5 Seeded Faults	No. of Test Cases Written	No. of SLOC Written
Participant A: UnitGen	4	10	0
Participant A: JUnit on Eclipse	2	4	22
Participant B: UnitGen	3	14	0
Participant B: JUnit on Eclipse	3	7	21

5.2.3 Experiment 5 - Conclusion

Participants managed to produce twice as many test cases using UnitGen compared to doing it manually with the aid of JUnit plug-in on Eclipse, in the 1/2 hour given for each tool. One of the participants found double the errors based on the test cases generated by UnitGen, while the other found same number of errors using both tools.

With UnitGen, participants did not have to write any test code themselves as the JUnit testclass was generated fully by the tool. This was seen as a major benefit by the participants as they are not familiar with writing test cases in JUnit, and they did not have to spend time coding the state of the object for testing.

One drawback observed is that UnitGen's disability to test constructors, which is a design oversight. It was incorrectly assumed that constructors are implicitly tested during object interaction, this is not true for the case of UnitGen as it allows user to set the state of objects, thus overriding the values initialised by the constructors. In such a situation, any errors made by the constructor in initializing values would have been missed by UnitGen's generated testclass. This is exactly what happened during the experiment. Despite this, both participants found UnitGen and the accompanying tutorial very useful and easy to use. One of them felt that UnitGen is more useful than other tools he has heard about, especially with regards to testing object states. He also remarked however that the tutorial was too lengthy to read.

Conclusion

In my research, I've managed to identify and evaluate the barriers people encounter in using TDD through literature survey and my own experiments, described in sections 3.2, 3.3 and 3.4. As a result of my findings, I felt that in order for TDD to be adopted and used effectively by students, tool support must be provided to help students with writing test cases, as well as reduce the time and effort required in building test cases so that students are more receptive to the idea of using TDD.

Just as JUnit contributed to the rise of popularity of TDD [12], I believe with additional tool support, on top of JUnit, TDD might become more popular amongst students. To that end, I have developed a unit test generation tool, UnitGen, with an accompanying tutorial with guidelines on writing test cases.

UnitGen, allows users to define the state of the object before method execution, thus users can setup specific situations for testing. It also allows user to test the object state after method execution, without having to resort to “dirty coding”, i.e. changing private access to public or protected, or writing a toString method to return a string representation of the object state, the approach used by JNuke [2]. Lastly, unlike JUnit or JML-JUnit [6], it allows users to test private methods.

UnitGen is designed for beginner programmers, using the traditional approach of testing expected output against actual output. This is in contrast to JML-JUnit, which uses formal specifications to generate source codes, which is difficult to grasp for beginner programmers. While UnitGen is not able to build test cases for all scenarios, it is able to reduce development time by helping to generate test cases for most situations.

Experiments were designed to evaluate the effectiveness of UnitGen against JUnit in terms of number of test cases produced, source lines of code written and number of errors found in a fixed amount of time. This demonstrates UnitGen's ability to reduce development time and effort spent in creating test cases. Initial evaluation of UnitGen and its accompanying tutorial shows positive results as

described in sections 5.1 and 5.2.

While the initial evaluation results suggest UnitGen and its accompanying tutorial are useful in aiding students create test cases, few tests have been done to evaluate the effectiveness of UnitGen and my tutorial in promoting the adoption of TDD amongst students. In addition, there isn't any statistical data to show that my tutorial helped students in building more effective test cases. As such, further experiments need to be conducted to demonstrate the effectiveness of UnitGen, as well as observed the effects of introducing TDD with UnitGen as support to students.

6.1 Future Work

To further evaluate the effectiveness of UnitGen and its tutorial, experiments must be conducted with more participants as well as participants of various skill levels in unit testing. Comparison of UnitGen against other similar tools such as JML-JUnit is also necessary. Stress testing of UnitGen is also needed, by using UnitGen on more complex Java classes or on classes with more seeded faults. It is also a good idea to deploy UnitGen as a teaching tool, for use in school laboratories and obtain feedback on whether it helped with encouraging students to use TDD.

An experiment should be conducted on two groups of students, with one group given the tutorial to study. Both groups should then be tasked to develop a JUnit testclass to test a Java class with seeded faults, without any time constraints. This allows us to observe which groups' testclass is able to detect more errors, and thus prove if the tutorial helped students in generating better test cases.

Based on feedback from the participants of my experiments, the following areas of UnitGen and my tutorial have been highlighted for improvement.

1. Assumption that constructor is implicitly tested when it is used to instantiate an object is wrong. Constructors may have functions that needs to be tested as well. For example, initialising the values of the class fields. This oversight in the tool design needs to be corrected.
2. A more user-friendly interface is needed, as many students have difficulties using it.
3. Provide names of input parameters to methods and constructors. While it is not possible to obtain them from the Java class file using Java reflection, it is possible to parse the Java source code itself for the required information.

4. The Test Data File can be printed in a more human-readable format, and be used directly as a test documentation, as well as make it more convenient to enter test data through the Test Data File.
5. Improve UnitGen to allow it to test methods that accept objects as input parameters or returns object as an output, as object interaction is an important part of object-oriented applications.
6. A more comprehensive guide to using UnitGen is required, as many felt that the current information is not sufficient for new users to grasp the workings of UnitGen.
7. Convert UnitGen into an Eclipse plug-in for convenient distribution into school laboratory machines.

APPENDIX A

Original Honours Proposal

Title: Generation of Comprehensive Test Cases for Test-Driven Development to Aid the Learning Process

Author: Boh Sui Jimm

Supervisor: Dr. Rachel Cardell-Oliver

Introduction to Test-Driven Development

Test-Driven Development (TDD), a test-first approach to software development, gained popularity with the rise of agile process models such as Extreme Programming (XP). Although, TDD gained visibility only in the recent years with the introduction of Extreme Programming in 1998, it has been practiced informally for decades with one of the earliest references to its use in the late 1950s in the NASA Project Mercury [12].

TDD, also commonly referred to by various names such as test-first programming, test-driven design or test-first design [12], adopts a novel approach to software engineering by generating test cases before writing the source code, and using the test cases to drive the development and design of the system.

In traditional programming practices, unit tests are constructed after code is written, which could be anytime after the code is written, be it a few minutes or even months later. In addition, unit tests might be written and conducted by either software testing teams or the same programmers who written the source code. In the case of TDD, unit test cases are generated by the programmers before writing the source code, and testing is conducted by the programmers immediately after the unit source codes are written.

TDD, however, is more than just a test-first approach, as mentioned by Kent Beck, creator of Extreme Programming and co-founder of Agile Manifesto and

JUnit unit testing framework[16]. TDD takes the test-first approach to the extreme by always writing tests before code, making tests as small as possible, and never letting code degrade [12]. Scott Ambler[1], in his article Introduction to Test-Driven Development (TDD), described TDD with a simple formula, TDD = TFD + Refactoring. Refactoring, a key aspect of TDD, refers to changing the structure of a code without changing its external behavior. That is, improving the code structure yet ensuring it still passes the tests [12].

Three Aspects of TDD

A common misconception about TDD is that it is a testing technique, in truth; its nature is far more complex. There are three aspects to TDD, as its name suggests; Test, Driven and Development.

Test Aspect which most people are familiar with, involves designing automated tests for each individual unit of a program. A unit in this case, refers to the smallest possible testable software component, such as a method or a procedure or in some cases a class in an object-oriented context. While tests may or may not need to be automated, automated testing reduces the effort required to conduct testing, particularly regression testing of large software systems. In addition, TDD assumes the existence of automated testing, as it requires frequent execution of tests and regression testing, for its iterative development cycles[12].

Without, automated testing, TDD would be too much of a hassle to practice. In fact, the existence of automated testing tools such as the xUnit family of frameworks, contributed to the rising popularity of TDD. Testing in TDD, as mentioned earlier in the text, refers to designing and writing automated unit tests before the code, and executing them immediately after the code is written to provide prompt feedback[12].

Driven Aspect refers to TDD leading analysis, design and programming decisions, achieved through refactoring [12]. TDDs Driven Aspect is based on two assumptions, firstly, that software design being pliable and flexible, open to changes, this is similar to Agile Process Models idea of adaptive development. Secondly, as the process of test writing is the one of the first steps in deciding what a program should do, it is essentially a form of analysis [12].

Based on the order of approach in TDD, since tests are written before coding, and the assumption that test writing is a form of analysis, it leads to the conclusion that the process of writing tests drives the design of the system. Agile

Alliance, a non-profit organization, provides a definition that captures this concept : Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code. [12].

The idea is that the tests can be generated before actual code is written and the process of doing so can be used to drive the development and design of the system as the tests control the behavior of what the actual code should do indirectly through the fact that the functionality codes must pass the tests designed in order to proceed on in the TDD development cycle[12].

Development Aspect implies that TDD is not itself a software development methodology or process model, instead it is to be used in the context of other process models as a form of micro-process [12].

TDD assumes that automated tests generated throughout the development cycle are not discarded once a design decision is made, instead they become a crucial part of the development cycle by providing prompt feedback to any subsequent changes made to the system. This allow developers to make changes with confidence as regression testing can be executed immediately after and should any change results in a failure, the tests are still fresh in the developers mind. However, a drawback of this practice is that the developer must now maintain both the coding and the suite of automated tests generated thus far [12].

Aim

To date, studies and research has been concentrated on evaluating the performance of TDD, such as defect-density reduction and its impact on programmers productivity etc. However, it is noted that little or no attention has been paid to the very first phase of TDD, i.e. generating test cases. For TDD to be widely adopted, it is in my opinion that proper guidelines or tools must be available to start-off or aid novice or professional programmers alike in generating comprehensive unit test cases which is essential to the performance of TDD. For example, with the advent of XUnit, family of frameworks, allowing for automated test cases, TDD has gained increasing popularity amongst software developers. Likewise, with proper guidelines or tools for generating test cases, it is reasonable to assume that it will boost TDDs popularity even further, particularly in an

educational environment, where there has been conclusive evidence to the effectiveness and popularity of using TDD to teaching programming. This is shown in a 2004 survey conducted at Virginia Tech where 65.3% to 67.3% of the 59 students involved in the survey agree or strongly agree that using TDD increases the confidence that they have in the correctness of their code as well as when making changes to their code [8]. This research aims to come up with a test-design methodology with two criteria. Firstly it must be capable of generating comprehensive test cases based on formal specifications alone, i.e. black-box testing, due to the fact that in TDD, test cases are generated before coding, thus code structure is not available at the time of designing test cases. Secondly, the methodology should prove to be simple enough that it can be applied in a cookbook manner by programmers who are novice to the art of software testing. The proposed test-design methodology could be a whole new testing strategy or an improved version of a current testing strategy. As such, to start off, research will be done in the area of software testing, particularly as to what constitutes as a good test-case and test-case design methodologies. Two fundamental test-case design methodologies, Equivalence Partitioning and Boundary-Value Analysis, have been highlighted for further research. In Equivalence Partitioning, the idea is to avoid exhaustive-input test of a program as it is impossible, instead inputs are selected from subsets with highest probability of finding most errors[24]. As for Boundary-Value Analysis, the concept behind is the reasoning that test cases that explore boundary conditions having higher probability of finding errors than those that do not. Boundary conditions, in this case refer to situations directly on, above or beneath the edges of input equivalence classes and output equivalence classes[24]. Both methodologies are suitable for generating black-box test cases, as they focus on the inputs and outputs of a program. However it is noted that boundary-value analysis requires a degree of creativity and certain amount of specialization towards the problem at hand and thus not be suitable as a cookbook solution[24]

In addition, while the above two approaches, are suitable for black-box unit testing, they need to be scale up for unit testing in an object-oriented system. The problem lies not only with Equivalence Partitioning and Boundary Value Analysis, but in software testing methodologies suggested in the past two decades. This is because, these software testing strategies are based on traditional function-oriented paradigm. As such, they have been found inadequate for testing systems developed by object-oriented paradigm. The problem is that the object-oriented paradigm introduces new concepts, such as encapsulation, object state-dependent behavior, object interactions, inheritance, polymorphism etc. These new concepts, introduces new angles to software testing which must now be taken into consideration in order for the testing strategies to be efficient.

Related Works

Research on TDD

With the growing interest in TDD, there have been various studies and experiments conducted in an attempt to prove the effectiveness of TDD, both in the industry as well as in academia. In the industry, a joint case study by North Carolina State University, Department of Computer Science and IBM Corporation was conducted to observe the effect of TDD as a defect-reduction practice. Participants of the case study were the software engineers of a IBM development group, with experience in developing device drivers for more than a decade, which developed the legacy product used as the baseline in the case study and a group of nine full-time IBM engineers, without prior experience with TDD and novices to the target device used as the baseline. This second group was to develop a new system for the target device through the practice of TDD. Both groups codes would be evaluated against an external FVT (Functional Verification Test) group, which would run the codes against 3 sets of black-box tests that have been generated; partly automated, fully automated and fully manual. It was observed that the defect rate (i.e. faults per KLOC or fault density) of the code of the new system developed using TDD, performed significantly better in the FVT/regression testing compared to the legacy system developed by the experienced IBM development group which primarily used adhoc testing. The new system seems to show a 40% achieved with minimal impact to developer productivity [32]. In the academia, Stephen H. Edwards of Virginia Tech, Department of Computer Science, analyzed two semesters of programming assignments, one in Spring 2001 and the other Spring 2003, with the latter being a pilot test of including TDD and Web-CAT, an automated grading system which assigns scores based on three criteria; code correctness, test completeness with respect to the code and test completeness and validity, as part of the teaching curriculum. He observed that grades from Spring 2003 are slightly higher than those from Spring 2001, when graded by the traditional method, and significantly higher when graded by Web-CAT. In addition, it is also observed that there are

significantly fewer test case failures from the master suite when compared against the code from Spring 2003 than when compared against the code from Spring 2001. Lastly, students in Spring 2003, who used TDD and Web-CAT submitted programs contained approximately 45% [8]. Despite the rising popularity of Agile methods, in particular XP, which implies a growing adoption of TDD, there is no concrete evidence to show that TDD is being widely accepted (Janzen Saiedian 2005). Recent surveys, paint a rosy picture of the situation however, despite the lack of solid confirmation of TDDs adoption. In 2002, a survey of

32 respondents conducted across 10 industry segments, showed 14 firms using an Agile method, of these numbers, most were used in small projects lasting a year or less. In 2003 however, 131 respondents claimed they used an Agile method, and of these, 59 to use XP and implied using TDD. Both surveys showed positive feedback regarding the application of agile methods, with results such as increased productivity and quality with reduced or minimal changes to cost [12]

Research on Unit Testing

To achieve our goal of generating comprehensive test cases for use with TDD, our unit test cases must be adapted to test object-oriented systems. In the article by Kung et al, An Object Oriented Testing and Maintenance Environment, it was suggested that in order for testing to be adequate for Object-Oriented Systems, Object-Oriented concepts introduced have to be taken into account for testing. Their approach was to come up with an object-oriented testing and maintenance environment, termed OOTME Model. This model incorporates three types of diagrams; Object Relation Diagram (ORD), Block Branch Diagram (BBD) and Object State Diagram (OSD), to illustrate the OO concepts used in the system. With a clearer picture of the OO concepts used, it facilitates the understanding and test preparation and maintenance of OO programs[17]. Their idea of improving test generation by taking into account of Object-Oriented concepts can be used to improve the effectiveness of the test cases generated for TDD. Testing strategies based on function-oriented paradigm can be scaled up to take into consideration OO concepts, for used in generating comprehensive test cases for TDD, which is after all, a programming practice used in object-oriented development environment. In addition to coming up with a testing strategy to make test case generation easier, we could further our research goal of making TDD easier to learn by novice programmers, by making the testing process easier. This can be achieved by automating the test case generation process and providing mechanisms lacking in current unit testing framework, i.e. JUnit. In an article by Artho et al, Advanced Unit Testing: How to Scale up a Unit Test Framework, it was suggested that while JUnit certainly facilitates unit testing, it lacks support for log and error log files. Such logging allows tracking of internal state of a component much more easily and is a key for larger-scale integration testing[2]. In addition, it also suggests that by providing a String representation of the object state, it allows the analysis of the internal object state at a glance, making debugging easier[2].

Draft Task Schedule

The following tasks have been identified to be completed over the course of this semester including Summer Holidays:

1. Learn to use Eclipse IDE and the JUnit plug-in, which will be necessary in conducting experiments on the effectiveness of various test-design methodologies. (Completed)
2. Research into various test-case design methodologies, focusing on methodologies that can be applied to black-box testing, e.g. Equivalence Partitioning, Boundary- Value Analysis. (Completed)
3. Identify the strengths and weaknesses of each methodology and prepare a report of each. (In Process)
4. Implement the methodologies using a baseline program, on Eclipse and JUnit, and observe their strengths and weaknesses in an actual working situation.
5. Come up with an effective test-case design methodology which could either be a new methodology on its own, or an improvement of existing methodologies. Methodology should be simple enough to be applied in a cookbook manner, to aid 1st and 2nd year programming students in generating test cases for practicing TDD, yet effective enough to generate comprehensive test cases. (In Process)
6. Devise a software engineering tool set based on the methodology and field-test it. Software engineering tool could be a checklist, a tutorial or optimally a software program to aid user in generating input/output values for black-box testing using afore-mentioned methodology, thus automating in part the generation of test cases themselves. The following tools are under consideration for the project:
 - (a) Tutorial on Testing Strategy.
 - (b) Template for Generating Test Cases
 - (c) A program that generates Test Cases based on User Inputs. This program could include features such as log files to aid in the debugging process.
 - (d) Integrate above mentioned program into Eclipse as a plug-in.
 - (e) Usage studies could be conducted and the results compared against other test strategies

The above mentioned schedule maybe subjected to further revision in the future.

APPENDIX B

Tutorial

Bibliography

- [1] Ambler, S. 2006, Introduction to Test-Driven Development, [Online], Available from: <http://www.agiledata.org/essays/tdd.html> [7 Aug 2006].
- [2] Artho, C. & Biere, A. 2006, Advanced Unit Testing: How to Scale up a Unit Test Framework, Proceedings of the 2006 international workshop on Automation of software test, pp. 92-98, Available from: ACM Digital Library, [16 October 2006].
- [3] Atherton, J., S. 2005, Learning and Teaching: Experiential Learning, [Online], Available from: <http://learningandteaching.info/learning/experience.htm> [16 May 2007].
- [4] Bhat, T. & Nagappan, N. 2006, 'Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies', Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, pp 356-363, Available from: ACM Digital Library, [31 March 2007].
- [5] Binder, R. V. 2000, Testing Object-Oriented Systems, Addison Wesley, Massachusetts.
- [6] Cheon, Y. & Leavens, G. T. 2001, 'A Simple and Practical Approach to Unit Testing: The JML and JUnit Way', Department of Computer Science, 226 Atanasoff Hall, Iowa State University, Available From: <http://archives.cs.iastate.edu/documents/disk0/00/00/02/58/00000258-00/tr01-12.pdf> [5 April 2007].
- [7] Edwards, S. H. 2003, 'Rethinking Computer Science Education from a Test-First Perspective', Proceedings of the 18th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pp 148-155, Available from: ACM Digital Library Library, [31 March 2007].
- [8] Edwards, S. H. 2004, Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action, Proceedings of the 35th SIGCSE technical symposium on Computer science education, vol. 36, no. 1, pp. 26-30, Available from: ACM Digital Library, [24 Jul 2006].

- [9] Erdogmus, H., Morisio, M. & Torchiano, M. 2005, On the Effectiveness of the Test-First Approach to Programming, *IEEE Transactions on Software Engineering*, [Online], vol. 31, no. 3, pp. 226-237, Available from: IEEE Digital Library, [10 Jul 2006].
- [10] George, B. & Williams, L. 2003, 'An Initial Investigation of Test Driven Development in Industry', *Proceedings of the 2003 ACM Symposium on Applied Computing*, pp. 1135-1139, Available from: ACM Digital Library, [31 March 2006].
- [11] Geras, A., Smith, M. & Miller, J. 2004, A Prototype Empirical Evaluation of Test Driven Development, *Proceedings of the 10th International Symposium on Software Metrics*, pp. 1-12, Available from: IEEE Digital Library, [3 Jul 2006].
- [12] Janzen, D. S. & Saiedian, H. 2005, Test-Driven Development: Concepts, Taxonomy, and Future Direction, *IEEE Computer Society*, pp. 43-40, Available from: IEEE Digital Library, [3 Jul 2006].
- [13] Janzen, D. S. & Saiedian, H. 2006, On the Influence of Test-Driven Development On Software Design, *Proceedings of the 19th Conference on Software Engineering Education & Training*, pp.1-8, Available from: IEEE Digital Library, [8 Jul 2006].
- [14] Jones, C. G. 2004, 'Test-Driven Development Goes To School', *Journal of Computer Sciences in Colleges*, vol. 20, no. 1, pp. 220-231, Available from: ACM Digital Library, [31 March 2007].
- [15] JUnit, Testing Resource for Extreme Programming 2004, 'JUnit.org', Available From: <http://www.junit.org/index.htm> [1 April 2007].
- [16] Kent Beck - Wikipedia, the Free Encyclopedia 2007, 'Kent Beck', Available From: http://en.wikipedia.org/wiki/Kent_Beck [3 Jul 2006].
- [17] Kung, C. K., Gao, J. & Hsia, P. 1994, An Object Oriented Testing and Maintenance Environment, *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative research*, pp. 1-13, Available from: ACM Digital Library, [16 Oct 2006].
- [18] Lutz, R. R. 1993, 'Analyzing Software Requirements Errors in Safety-Critical Embedded Systems', *Proceedings of IEEE International Symposium on 4-6 Jan*, pp. 126-133, Available From: IEEE Digital Library, [8 Jul 2006].

- [19] Melnik, G., Maurer, F. 2004, 'Introducing Agile Methods: Three Years of Experience', Euromicro Conference, 2004 Proceedings 30th, pp 334-341, Available from: IEEE Digital Library, [3 Jul 2006]
- [20] Mugridge, R. 2003, 'Test Driven Development and the Scientific Method', Proceedings of the Agile Development Conference, pp. 1-6, Available from: IEEE Digital Library, [10 Jul 2006].
- [21] Mugridge, R. 2003, 'Challenges in Teaching Test-Driven Development', Springer Berlin / Heidelberg, Available from: http://springerlink.metapress.com/content/qrckaetvtxbn/?sortorder=asc&p_o=60, [3 Jul 2006].
- [22] Mullar, M. M. & Hagner, O. 2002, 'Experiment about Test-First Programming', Software IEEE Proceedings, vol. 149. no. 5, pp. 131-136, Available from: IEEE Digital Library, [31 March 2007].
- [23] Muller, M. M. & Tichy W. F. 2001, 'Case Study: Extreme Programming in a University Environment', Proceedings of the 23rd International Conference on Software Engineering, pp 537-544, Available from: IEEE Digital Library, [31 March 2007].
- [24] Myers, G. J. 1979, The Art of Software Testing, John Wiley & Sons, New York.
- [25] Olan, M. 2003, 'Unit Testing: Test Early, Test Often', Journal of Computer Sciences in Colleges, vol. 10, no. 2, pp. 319-328, Available From: ACM Digital Library, [31 March 2007].
- [26] Paneur, M., Ciglarie, M., Trampus, M. & Vidmar, T. 2003, Towards Empirical Evaluation of Test-Driven Development in a University Environment, University of Ljubljana, Available from: IEEE Digital Library, [11 Jul 2006].
- [27] Smith, S. & Stoecklin, S. 2001, 'What we can learn from Extreme Programming', The Journal of Computing in Small Colleges, vol. 17, no. 2, pp. 144-151, Available From: ACM Digital Library, [31 March 2007].
- [28] Steindl, C. 2005, Test Driven Development, [Online], Available from: www.agilealliance.com/articles/steindlchristophtstd/file [1 Aug 2006].
- [29] Tan, R. P. & Edwards, H. S. 2004, 'Experiences evaluating the effectiveness of JML-JUnit testing', ACM SIGSOFT Software Engineering Notes Archive, vol. 29, no. 5, pp. 1-4, Available From: ACM Digital Library, [31 March 2007].

- [30] Test Infected, 'JUnit Test Infected: Programers Love Writing Tests', Available from: http://en.wikipedia.org/wiki/Kent_Beck [3 Jul 2006].
- [31] Thales Australia 2006, About Us, Thales Australia. Available from: <http://www.thalesgroup.com.au/site.asp?page=2> [16 May 2007].
- [32] Williams, L., Maximilien, E. M. & Vouk, M. 2003, Test Driven Development as a Defect- Reduction Practice, Proceedings of the 14th International Symposium on Software Reliability Engineering, pp. 1-12, Available from: IEEE Digital Library, [3 Jul 2006].