

Investigating the Effectiveness of Test-Driven Development

by
Adam Packer

Supervisor: Kemi Ademoye

Second Marker: Michael Dacey

Project submitted as part of the requirements for the award of
BSc(Hons) Web Development

April 2017

DECLARATION FORM AND FORM OF CONSENT

DECLARATION

I *Adam Packer* declare that I am the sole author of this Project; that all references cited have been consulted; that I have conducted all work of which this is a record, and that the finished work lies within the prescribed word limits.

This work has not previously been accepted as part of any other degree submission.

Signed: *Adam Packer*

Date: 25/04/17

FORM OF CONSENT

I *Adam Packer* hereby consent that my Project, submitted in candidature for the B.Sc. Web Development degree, if successful, may be made available for inter-library loan or photocopying (subject to the law of copyright), and that the title and abstract may be made available to outside organisations.

Signed: *Adam Packer*

Date: 25/04/17

ABSTRACT

The idea that test cases can be written before any of the functional code is a radical concept for most programmers: test-driven development is an agile practice that does just that. This research aims to address the conflicted views shared by industry professionals by determining the effects that test-driven development has on the development of software. A number of objectives were identified in meeting this aim, beginning with an exploration of the theoretical principles behind test-driven development and its canon of literature. An experiment was conducted as part of the second objective, wherein a software artefact was created using both test-driven development and a traditional approach to development. Data was collected on quality of code, programmer productivity, refactoring and dependencies allowing a detailed comparison of the two approaches. The third and final objective was to evaluate the effectiveness of test-driven development through analyses of the experimental results, and it was discovered that the test-first practice had a largely positive affect on the development of software. It produced code that was adjudged be of higher quality and better design than that of the traditional approach, howbeit some side effects were observed in terms of lower productivity and more dependant code.

CONTENTS

DECLARATION FORM AND FORM OF CONSENT	I
ABSTRACT	II
CONTENTS.....	III
TABLE OF FIGURES	VI
TABLE OF TABLES	VII
1 INTRODUCTION	8
1.1. AIMS AND OBJECTIVES.....	9
2 RESEARCH AND REVIEW OF LITERATURE.....	10
2.1. THE TESTING COMPONENT	10
2.1.1 <i>Unit Testing</i>	11
2.1.2 <i>Mocking</i>	12
2.1.3 <i>Regression Testing</i>	13
2.1.4. <i>Automated Testing</i>	14
2.2. THE DRIVEN COMPONENT	15
2.2.1. <i>Refactoring</i>	15
2.2.2. <i>Attitudes & Acceptance</i>	16
2.3. THE DEVELOPMENT COMPONENT	17
2.3.1. <i>Traditional Methodology</i>	17
2.3.2. <i>Agile Methodology</i>	18
2.3.3. <i>Agile Practice</i>	19
2.3.4. <i>Productivity</i>	20
2.4. CONCLUSION	21
3 RESEARCH AND DEVELOPMENT METHODOLOGY	23
3.1. PHILOSOPHY	23
3.1.1 <i>Positivism and Interpretivism</i>	24
3.2. APPROACH	25
3.2.1. <i>Hypotheses</i>	25
3.3. STRATEGY	26
3.4. METHODOLOGICAL CHOICE	27
3.5. TIME HORIZON	27
3.6. DATA COLLECTION	28
3.6.1. <i>Code Coverage</i>	28
3.6.2. <i>Defect Density</i>	30

3.6.3.	<i>Lines of Code per Time Frame</i>	32
3.6.4.	<i>Refactoring Density</i>	33
3.6.5.	<i>Dependency Density</i>	34
3.6.6.	<i>Ethical Issues</i>	35
3.7.	EXPERIMENT DESIGN	35
3.7.1.	<i>Variables</i>	37
3.7.2.	<i>Materials (Hardware and Software)</i>	39
4.	DESIGN	41
5.	IMPLEMENTATION	45
5.1.	STRUCTURING THE TESTS	45
5.2.	REFACTORING STRATEGY	47
5.3.	USE OF MOCK OBJECTS	49
5.4.	SELECTION OF IMPLEMENTATION PROBLEMS	51
5.4.1.	<i>Event</i>	51
5.4.2.	<i>Ticket</i>	52
5.4.3.	<i>Order</i>	54
5.4.4.	<i>Reservation</i>	56
5.4.5.	<i>Payment Gateway</i>	57
6.	RESULTS	59
H1.	TDD IMPROVES THE QUALITY OF CODE	60
H1.1.	<i>Code Coverage</i>	60
H1.2.	<i>Defect Density</i>	63
H2.	TDD AFFECTS THE PRODUCTIVITY OF THE PROGRAMMER	64
H3.	TDD ENCOURAGES MORE REFACTORING	67
H4.	TDD ENCOURAGES LOOSELY-COUPLED CODE	68
7.	EVALUATION	71
8.	CONCLUSIONS AND RECOMMENDATIONS	73
3.2	SUGGESTIONS FOR FURTHER WORK	74
9.	REFLECTION	76
10.	REFERENCES	79
APPENDICES		90
A.	FURTHER IMPLEMENTATION DETAILS	90
A.1.	<i>List of Test Cases</i>	90
A.2.	<i>Managing the Database</i>	91

<i>A.3. Use of Façades</i>	91
B. EXPERIMENT DEVELOPMENT LOG	92
C. DEFECT LOG	93
D. REFACTORING LOG	94
E. DEPENDENCY LOG	95

TABLE OF FIGURES

FIGURE 1: COHN'S TESTING PYRAMID [20]	12
FIGURE 2: ROYCE'S WATERFALL MODEL [40]	18
FIGURE 3: BECK'S XP MODEL WITH TDD [3]	20
FIGURE 4: SAUNDERS' ADAPTED RESEARCH UNION [51]	23
FIGURE 5: COMPARISON OF ITERATIVE TEST-LAST (LEFT) AND TRADITIONAL TEST-LAST (RIGHT) [87]	38
FIGURE 6: UML CLASS DIAGRAM FOR THE TICKET ORDERING SYSTEM	43
FIGURE 7: SEQUENCE DIAGRAM FOR THE TICKET ORDERING SYSTEM	44
FIGURE 8: TEST CASE IN THE SOFTWARE ARTEFACT USING THE ARRANGE-ACT-ASSERT PATTERN	46
FIGURE 9: CREATE ORDER METHOD BEFORE REFACTORING	48
FIGURE 10: CREATE ORDER METHOD AFTER REFACTORING	49
FIGURE 11: TEST CASE USING MOCK OBJECTS AS PART OF THE SETUP	50
FIGURE 12: TEST CASE USING SPY OBJECTS AS PART OF THE SETUP	50
FIGURE 13: TABLE STRUCTURE (LEFT) AND UML CLASS DIAGRAM (RIGHT) FOR THE EVENT MODEL	51
FIGURE 14: EVENT METHOD FOR CONVERTING TICKET PRICE IN PENNIES TO POUNDS	52
FIGURE 15: UML CLASS DIAGRAM FOR THE TICKET MODEL	53
FIGURE 16: HASHIDS CODE USED TO GENERATE UNIQUE TICKET CODES	54
FIGURE 17: UML CLASS DIAGRAM FOR THE ORDER MODEL	55
FIGURE 18: RANDOMISATION CODE USED TO GENERATE ORDER NUMBERS	55
FIGURE 19: PROCESS OF RESERVING SOME TICKETS, CHARGING THE USER AND CREATING AN ORDER	56
FIGURE 20: TEST FOR CHECKING THAT USER CANNOT PURCHASE TICKETS THAT ANOTHER USER IS TRYING TO PURCHASE	57
FIGURE 21: UML CLASS DIAGRAM FOR THE PAYMENT GATEWAY USING THE ADAPTER PATTERN	58
FIGURE 22: COVERAGE REPORT FOR APPLICATION CODE DEVELOPED USING TEST-DRIVEN DEVELOPMENT	60
FIGURE 23: COVERAGE REPORT FOR APPLICATION CODE DEVELOPED USING TRADITIONAL TEST-LAST DEVELOPMENT	61
FIGURE 24: COMPARISON OF TEST-DRIVEN DEVELOPMENT AND TRADITIONAL TEST-LAST COVERAGE DISTRIBUTION	62
FIGURE 25: COMPARISON OF TEST-DRIVEN DEVELOPMENT AND TRADITIONAL TEST-LAST DEFECT DENSITY	63
FIGURE 26: COMPARISON OF TEST-DRIVEN DEVELOPMENT AND TRADITIONAL TEST-LAST AVERAGE LINES PER HOUR	66

TABLE OF TABLES

TABLE 1: LIST OF FUNCTIONAL REQUIREMENTS FOR THE SOFTWARE ARTEFACT	42
TABLE 2: DEFECT DENSITY T-TEST ASSUMING EQUAL VARIANCES.....	64
TABLE 3: TOTAL LINES PER HOUR T-TEST ASSUMING EQUAL VARIANCES.....	65
TABLE 4: APPLICATION LINES OF CODE PER HOUR T-TEST ASSUMING EQUAL VARIANCES	65
TABLE 5: REFACTORING DENSITY T-TEST ASSUMING EQUAL VARIANCES	68
TABLE 6: DEPENDENCY DENSITY T-TEST ASSUMING EQUAL VARIANCES	69

1 INTRODUCTION

Test-driven development (TDD) is a controversial topic: some programmers swear by it, where others prefer traditional development approaches [1]. TDD describes a small and rapid development cycle wherein automated tests are written *before* any functional code. It has recently experienced growing attention as a core agile practice, though programmers have been using TDD for a number of years [2]. Beck is considered to have pioneered TDD with its rediscovery [3] in 2003, but the earliest known reference to this practice can be traced back to McCracken [4] in 1957:

The first attack on the checkout problem may be made before coding is begun. In order to fully ascertain the accuracy of the answers, it is necessary to have a hand-calculated check case with which to compare the answers, which will later be calculated by the machine.

Originally referred to as test-first programming, TDD has gained considerable popularity since the introduction of extreme programming: an agile model that uses short iterations to develop object-oriented software with limited upfront design [3]. As a result, a number of books explaining the underlying concepts and application of TDD have emerged, along with specific tools that support TDD across a number of different languages. Researchers have also started to examine the effects of TDD on the development of software and programmer attitudes in both academic and industrial environments, and educators [5] have begun to include TDD as part of their respective curriculums. Many of these research efforts have been implemented in the context of XP, while others are independent of them.

Recent years have seen TDD be the subject of much scrutiny from industry professionals. In 2014, Hansson [6] boldly declared that “test-driven development is dead” in a conference keynote of the same name. His comments are particularly pertinent because he created the popular framework *Ruby on Rails*, so in the weeks following the conference TDD pioneers Beck and Fowler [7] joined Hansson for some deliberation. Talks concluded with Hansson revering the intentions of TDD, but he voiced concerns about poorly designed code that exists solely to enable unit

tests. On the other hand, proponents Beck and Fowler defended the practicality of TDD and cited extensible code with looser coupling. Consequently, this research serves as a response to the conflicted views shared by industry professionals, offering clarity as to whether TDD is an effective approach to software development.

1.1. Aims and Objectives

Succinctly speaking, the overarching aim of this research is to determine the effects that a test-first approach has on the development of software. In meeting the aim, however, there are a number of sequential objectives that must be addressed and completed. They are as follows:

- O1. Gain an understanding of the test-first development approaches
- O2. Compare the effects of TDD against a traditional development approach
- O3. Evaluate the effectiveness of TDD through analysis of a result set

The objectives guide the research process and inform the structure of the report. Chapter 2 follows the introduction and explores the theoretical principles behind test-first approaches to software development. It also provides a critical review of the existing body of literature as it pertains to TDD. Chapter 3 builds upon the findings of the literature review, outlining the procedures and experimental design chosen as part of the research methodology. Chapter 4 shifts focus to the software artefact by detailing a high-level view of its design. Chapter 5 proceeds to document the techniques and strategies that were used in implementing the software artefact. Some of the interesting problems and subsequent solutions are also discussed for an insight into the decision-making processes that were undertaken. Chapter 6 analyses the results recorded during the course of the experiment, with a particular emphasis on the significance of these results. Chapter 7 evaluates the extent to which the research aim and objectives have been met. Chapter 8 summarises the previous chapters in the form of a conclusion and makes recommendations for further avenues of research. Chapter 9 presents an opportunity for reflection on the process of writing the report.

2 RESEARCH AND REVIEW OF LITERATURE

It is commonly misconceived that TDD is a testing strategy. Accordingly, the upcoming subsections of this literature review make an attempt to clarify the ambiguous nomenclature by investigating each component part in isolation (the testing component, the driven component and the development component). The major topics of discussion include the different types of testing and the levels at which they occur, before shifting attention to the ways in which testing can inform software design and the methodologies that guide development activity. The existing body of literature is referenced throughout, which serves to provide a better understanding of the ways in which TDD affects software development and the reasons for the reluctance exhibited by programmers in adopting the practice.

2.1. The Testing Component

The definition of successful software differs from person to person. For users, successful software provides real benefit; for investors, successful software generates return on investment. However, it cannot be disputed that quality is the biggest determinant of success. Cooper [8] identifies that, by this criterion, the top 20% of products have a success rate of 98%, whereas the bottom 20% of products have a success rate of just 18.4%. The definition of software quality is largely subjective. Pressman [9] defines quality as the extent to which a software meets the requirements and specification; Westfall [10] defines quality as the operability of a software in its intended environment. In either case, improving quality is in the best interest of stakeholders.

Testing is one method of improving software quality. It provides an objective, independent view of the software by discovering the faults (or bugs) that produce incorrect or unexpected results. Knowing that a bug exists, the programmer(s) can then make the necessary corrections. It is important to note, however, that testing and debugging are separate activities. Myers [11] made the distinction in 1956. He stated that testing serves to prove that software works, whereas debugging is the isolated process of removing specific bugs. Since then, the goals of testing have transitioned from being demonstration and destruction-oriented to evaluation and prevention-oriented [12].

There has also been a transition in the roles of software testers. Prior to the 1980's, testing was considered to be the sole responsibility of the programmer. As the importance of testing grew – due to the increasing cost of software failure [12] – roles dedicated to the writing and execution of tests began to emerge. Nowadays it is not uncommon for the joint involvement in testing activities.

2.1.1 Unit Testing

Testing in TDD involves writing automated tests for the software's individual units. A unit is the smallest testable component of a software, and is recognised [13] as the lowest level of software testing. However, there is some debate as to the definition of a unit. Binder [14] argues that a unit in object-oriented programming should constitute a class, whereas most programmers believe that a method is the smallest testable block of code. TDD programmers are encouraged [15] to write unit tests at the method level because it reduces the debugging effort. Moreover, smaller test cases are easier to understand.

Unit tests are traditionally written after the unit has been coded. They can be written by the programmer or a designated tester, and take anywhere from a few minutes to a few months. Nevertheless, one of the defining features of TDD is that unit tests are written *before* the code under test. The tests inform development decisions, and as such, must be written by the programmer(s).

Theoretically, by associating every piece of code with a unit test, TDD should produce a higher quality of software (when compared to traditional testing approaches). In practice however, the evidence is inconclusive. When IBM transitioned to TDD, Maximilien and Williams [16] documented a 47% increase in code quality. However, this contradicts the research of Pancur et al. [17] whose results showed a decrease in the quality of code (though the difference was marginal and could be attributed to chance). Another study [18] observed that TDD had an indiscernible effect on code quality.

The mixed results pertaining to the effect of TDD on code quality has encouraged other authors to adopt a sceptical stance on the practice. Consider Janzen and Saiedian's [19] paper entitled *Does Test-Driven Development Really Improve*

Software Design Quality, which concludes that TDD programmers do write smaller units that are tested to a greater degree. However, they could not confirm claims that TDD improves software quality, proposing instead that quality may be affected by factors outside of the test-first aspect. Interestingly, Erdogmus et al. [18] investigated the relationship between number of tests and code quality, independent of whether tests were written first or last. Although a fewer number of tests could produce high-quality software, they concluded that the minimum quality achievable increases with the number of tests written. Therefore, the incentive to write more tests from strict TDD adherence may be responsible for the increase in code quality, as opposed to the test-first concept.

2.1.2 Mocking

Integration and system tests occur at a higher level of software testing. An integration test combines units to verify their functionality as a group; a system test validates that integrated systems meet the requirements. Cohn [20] identifies that the time and resources required for testing increases with the level at which it occurs (see figure 1). There is also larger scope for error, which makes debugging particularly difficult.

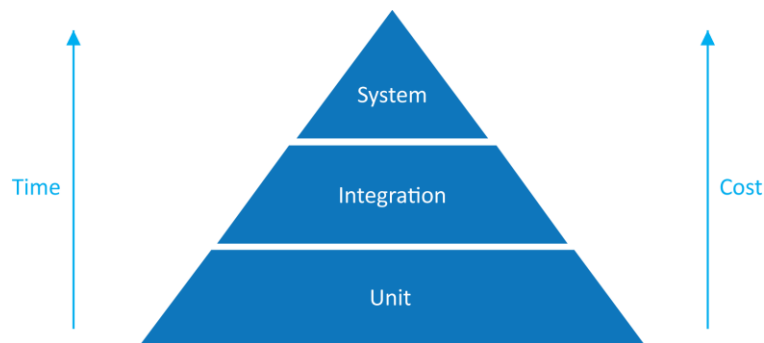


Figure 1: Cohn's Testing Pyramid [20]

TDD literature [21] discourages the use of high-level testing. Instead, fast-running unit tests should be the foundation of the development process. In situations where a unit depends on another system or resource, Beck [22] advocates the use of mocks: a fake object that simulates the behaviour of a real object. This allows units to be tested in isolation, without the need for high-level testing. Consider the classic database example. Databases are difficult to keep clean and take a long time to start. Moreover, if a database is located on a remote server, then any associated

tests are bound to a physical location on the network. Beck [22] offers an alternative:

The solution is not to use a real database most of the time. Most tests are written in terms of an object that acts like a database, but is really just sitting in memory.

Beyond reliability and performance, mock objects provide greater readability. By reading the preceding test from one end to another, programmers can ascertain *why* a particular result was returned. Alternatively, if a query on a database containing realistic data returns eight results, the programmer does not know whether eight is the correct answer. Mock objects also encourage the careful consideration of unit dependency, and therefore loosely coupled software. However, a mock presents some risk if it does not behave like the real object. As a consequence, Samimi et al. [23] argue that it is difficult and error-prone for programmers to test rich properties of their code. To mitigate this risk, additional tests for the mock objects should be written, which can then be applied to the real object once it becomes available.

2.1.3 Regression Testing

Test execution can be the responsibility of dedicated testers. This is the case at IBM, where test builds are scheduled on a daily or weekly basis [16]. However, programmers will frequently execute the tests for immediate feedback on any changes they have made: a technique known as regression testing. Regression testing is used to verify that previously developed software continues to work after code has been changed elsewhere. The most common technique is to re-run *all* tests after every addition or alteration to the code, ensuring that no bugs have been introduced to the system.

Regression testing plays an important role in TDD. However, the programmer does not need to write explicit regression tests. Being that a unit test describes the expected behaviour of a block of code; it serves as a regression test that can be re-executed after changes to the code. Nonetheless, it is well documented [24] that executing the entire body of test cases after every small change is expensive (in terms of execution time). Cibulski and Yehudai [25] have addressed this concern by

proposing a test selection technique, which uses a smaller subset of test cases for targeted regression testing.

2.1.4. Automated Testing

Test execution can be a manual or automatic process. Automated testing uses external software to manage the execution of tests and compares the actual result with the expected result. It is particularly useful for repetitive or difficult testing activities that would be too laborious to do manually [26]. An automated test is written in terms of source code, and as such, must be written by a programmer. Regression testing is a good candidate for automation because there are a large number of tests that need to be executed quickly and repeatedly.

Automated unit testing is a fundamental feature of TDD. Proponents [27] argue that it produces more reliable, lower cost software than that which is manually tested. Accordingly, strict adherence to the TDD practice implies the use of an automated framework: an integrated system that specifies the rules for automation. Without such tools, Becks [22] declares that TDD is “virtually impossible”.

There is another reason for writing automated unit tests, beyond simplifying the testing process. An automated test provides the programmer with confidence that a piece of code functions correctly. In other words, it is a tool for managing fear during the programming of software. Fear has a number of negative connotations in the context of programming. Not only does it create hesitancy, but it discourages communication and makes programmers shy away from feedback [28].

TDD is aware of the gap between decision-making and feedback, and the techniques used to control this gap. Once a test has been made to work, the programmer knows that the code is working now, and will continue to do so for the duration of development. They are one step closer to complete software. With the confidence provided by tests, the process continues: get the next test working, then the next, and so forth. When confronted with a tougher programming problem, each test should cover less ground to encourage progress.

The majority of people who learn TDD discover that their programming practices changed for good. Gamma [29] coined the term ‘test infected’ to describe this transition. Programmers find themselves writing more tests earlier, and progressing in smaller steps than would be considered sensible. Nevertheless, it is not uncommon [30] for programmers to learn TDD and revert back to their previous practices, using it exclusively in situations where traditional programming is not making progress.

2.2. The Driven Component

Some definitions [31] of TDD wrongly imply that it is a testing strategy. However, Cunningham [32] explains that “test-first coding is not a testing technique.” The *driven* in test-driven development actually refers to the fact that design, analysis and programming decisions are guided by automated tests (it is assumed that the design of the software is pliable, incomplete or subject to change). Writing a test serves as the first step towards deciding what the software should do, which is essentially an analysis step. Consider the explanation that the Agile Alliance [33] offers:

Test-driven development 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 and the test code.

2.2.1. Refactoring

In order to promote testing to a design and analysis step, the practice of refactoring must be introduced. Refactoring describes a change to the code’s internal structure without affecting its external behaviour [34]. Even though a test may pass, the code could be inflexible or highly complex. By refactoring the code, the test(s) should still pass but the internal structures will have been improved.

In TDD there are three main strategies for driving design through refactoring. The ‘fake it’ strategy requires the programmer return constants first, before gradually replacing them with variables until the real correct solution is exposed. On the

other hand, the obvious implementation strategy describes a process where programmers type the correct solution straight away. Beck [22] uses a combination of the two. When everything is progressing as expected and he knows what to type, he uses obvious implementation to speed-up development (still running the tests after any code addition). At the point where there is an unexpected result, the focus shifts to faking implementations and refactoring until his confidence returns. The third and most conservative strategy is called triangulation. By analogy, a programmer can only triangulate when there are two or more test conditions. One condition enables a programmer to hard code the result into the solution; with two conditions, the programmer is forced to generalise the solution (removing any duplication between test and functional code).

2.2.2. Attitudes & Acceptance

To understand that TDD is focussed primarily on analysis and design, as opposed to testing, is conceptually the most challenging transition for new adopters of the practice. For the most part, this can be attributed to the fact that software testing has generally assumed the existence of functional code. With regards to TDD, the idea that a test can be written before the functional code, or that a test can influence decision making, is a radical concept [30] for the majority of programmers.

Much of the existing body of TDD research ignores programmer attitudes towards the practice. However, it is an important factor to consider because it has a bearing on the acceptance and adoption of TDD. One study [17] addressed this by questioning programmers on the effectiveness of TDD. A large disparity was recorded, with over 60% of participants admitting difficulty with adopting the test-first mind-set (compared with 20% for traditional testing approaches).

Interestingly, the environment in which TDD is conducted (i.e. industry or academia) appears to affect the attitude of practitioners. In one study [16] the industry professionals exhibited a “very positive” attitude towards TDD and continued its use. Within academia however, a study [17] showed that students were hesitant to adopt the “not very effective” technique. This favourability shown by industry could be attributed to the professionals’ practical experience with

development methodologies and their familiarity with unit testing, whereas the majority of student participants had no prior experience of testing code or working within a set methodology.

2.3. The Development Component

Designing, developing and delivering software is an arduous process that can require the co-ordinated effort of many programmers and teams. In a commercial environment, development is further complicated by deadlines, budgets, and other business constraints. A software development methodology offers better planning and management by describing the sequence of stages involved in the software life cycle. These stages can broadly be defined as analysis, design, implementation, testing, and maintenance – though not necessarily in that order [35] – and outline the activities, roles and deliverables expected of the implementer(s).

2.3.1. Traditional Methodology

Traditional methodology has many defining characteristics. Firstly, it has a rigid, sequential approach to software development. Each stage of the software life cycle must be completed in order and cannot begin unless the preceding stage has concluded. This linearity makes traditional methodology easy to implement for small-scale projects [36]. However, the lack of mobility means that the cost of bug-fixing increases exponentially [37] as you progress through the software life cycle. Secondly, traditional development methodology begins the software life cycle with a detailed design and analysis stage. Whilst this leads to clear deliverables and milestones, it can also generate risk and uncertainty if late changes are made to the requirements. Finally, traditional methodology encourages the use of many specialised teams. It is not uncommon for larger projects to have a different team for each stage of the software life cycle. However, to facilitate the flow of information between teams, they must document their development activity: a time-consuming process that can cause decisions to freeze, delaying the delivery of software.

The most common example of traditional methodology is Royce's waterfall model (see figure 2) which, despite the well-documented drawbacks, continues to be used by over a third of companies [38]. Nonetheless, opponents [39] argue that the

traditional approach to software development is fundamentally flawed because it does not reflect the iterative nature of development.

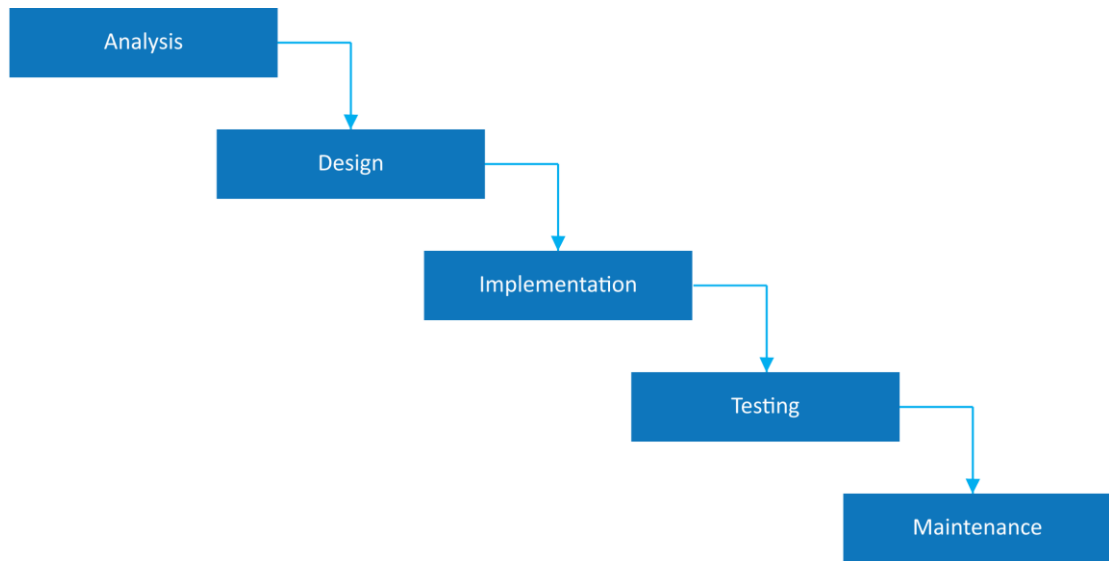


Figure 2: Royce's Waterfall Model [40]

TDD is one of the practices that emerged as a reaction to traditional development methodology, with its most important goal being incremental (or evolutionary) design [3]. Rather than making design decisions upfront and all at once, the software is designed incrementally test-by-test. This line of thinking aligns with that of the agile methodology.

2.3.2. Agile Methodology

Agile is a term that describes a lightweight approach to software development. Unlike traditional methodology, the characteristics of the agile methodology is informed [41] by a set of principles. The first principle of interest is integrated testing. Unlike traditional methodology, wherein testing is treated as its own stage in the software lifecycle, agile methodology tests as part of each software increment. Studies [6, 14] show that the increased frequency and availability of agile testing produces higher quality software than that of traditional approaches, though it was acknowledged that they are not directly comparable. The second agile principle to consider is continuous prototyping. Cooper [8] believes customer feedback loops to be the main contributor to product quality, thus producing software prototypes in iterative development cycles increases satisfaction through progressive feedback and risk mitigation [42]. Moreover, the limited use of upfront

planning means that projects are more responsive to late requirement changes. A final principle declares that software should be delivered early. As such, agile methodology uses self-organising, cross-functional teams that focus on working code, rather than static documentation. Opponents [43] have expressed cynicism in this approach, citing the disregard for core software engineering principles; however agile implementers do not dispute the benefits of documentation. Ambler [44] argues that development activity should be documented to the extent that programmers can communicate effectively, without creating any bureaucratic procedures.

It is apparent that agile methodology has addressed the problems inherent in traditional methodology, and is currently used by 95% of software development companies [45]. Whilst the transition was made difficult by deep-rooted anti-patterns and insufficient training, agile-specific adoption frameworks have since been introduced [46] to expedite the process.

2.3.3. Agile Practice

TDD is a micro process intended to aid with the construction of software. However, it is not a development methodology or model in its own right; it is an agile practice, a predefined way of developing software that is to be used in combination with other practices, in a specified order and frequency, and in the context of a model. Beck's XP model is a notable example within the agile development methodology that incorporates TDD (see figure 3).

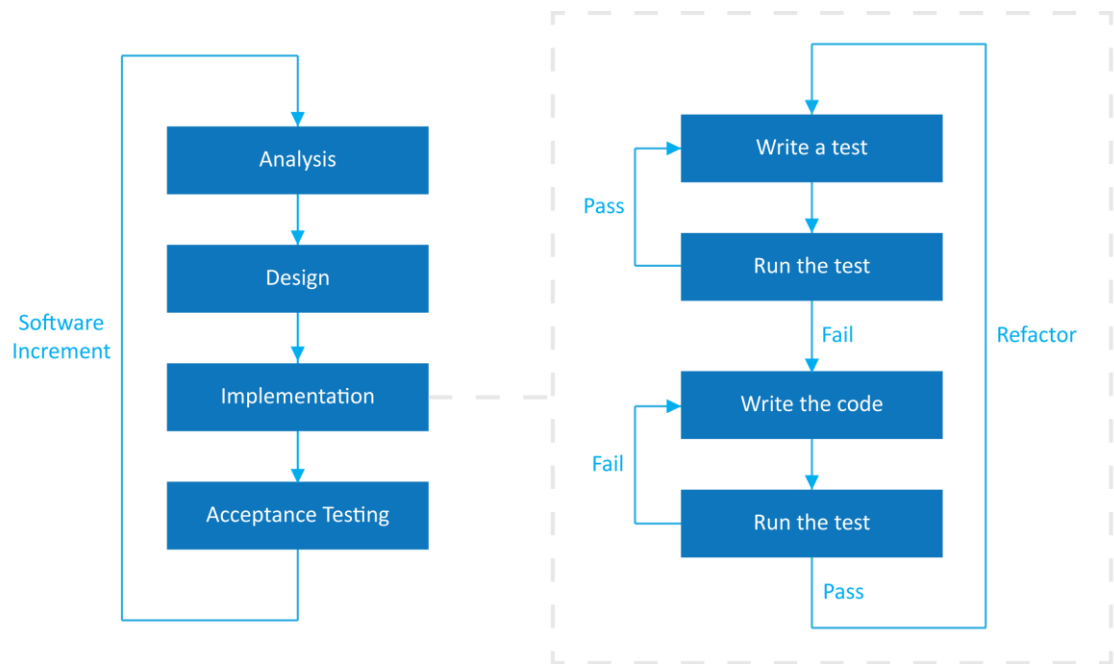


Figure 3: Beck's XP Model with TDD [3]

TDD introduces some side effects to the software development process. The practice involves the writing of automated tests, and makes the assumption that they will not be discarded after a design decision has been made. Instead, they become a resource that demonstrates how the code under test is supposed to work, thereby serving as a form of documentation. Unlike static documentation, tests will usually change in parallel with the code. However, there is now a responsibility to maintain the suite of tests. This has implications for programmer productivity.

2.3.4. Productivity

Many companies employ cost-cutting measures to remain competitive, and being that labour can [47] account for 70% of total costs, maximising productivity is of critical importance. Though adoption of the agile methodology has been somewhat stifled by retraining costs [48], it continues to gather support from programmers because it requires less human resources than that of traditional methodology. However, existing literature largely agrees that TDD has a negative effect on productivity, the extent to which is the subject of much debate; some research [16] describes a “minor” cost to productivity, whereas others describe considerable productivity loss. In one case [17], it was revealed that over two-thirds of

participants experienced an increased total development time, with the majority agreeing that TDD was ineffective.

On the other hand, instances of increased productivity (as a result of TDD) have been documented; in an experiment conducted by Erdogmus et al. [18] productivity was observed to improve “significantly”. On a side note, the programmers who practiced TDD also wrote 50-60% more tests, leading the authors to propose that a linear relationship exists between productivity and the number of tests written.

It can be argued that productivity would be expected to drop in the short-term because a high percentage of new code is being written, all of which must be accompanied by a new unit test. This is a slower process compared to that of traditional development approaches. However, Marchenko et al. [49] identify that little research has been performed using TDD over an extensive period of time. Over the course of three years, their study showed no negative effects with respect to productivity. The established test suite “simplified significantly” the software maintenance, allowing programmers to integrate new code at a greater speed with heightened confidence. In another long-term study, Sanchez et al. [50] indicate that perceived productivity losses are compensated by quality improvements. Hence, it can be inferred that the productivity benefits of TDD are mostly observed during the maintenance stage of the software lifecycle, where the code has already been associated with a body of tests.

2.4. Conclusion

In conclusion, the literature review clarified that TDD is an approach to software development and not strictly a testing strategy. The underlying theory claims that it improves the quality of software by associating all code with an automated unit test and demanding a period of refactoring from the programmer. These claims are substantiated by the literature, though the extent to which quality increases varies greatly among researchers. It was also revealed that TDD has a largely negative effect on the productivity of the programmer, but longitudinal studies have shown that it begins to increase as the test suite matures. While it can be inferred that the short-term productivity loss has stifled adoption of the practice, the literature

review suggests that retraining costs and difficulty transitioning to the test-first mind-set are more likely factors.

3 RESEARCH AND DEVELOPMENT METHODOLOGY

Following the analyses made from the literature review, a research methodology was devised to address the overarching aim of this research: determining the effects of TDD on the development of software. One of the most popular models available to researchers is Saunders' [51] research onion, which represents a perspective that presents information in a successive way, from broad constructions of research into the narrow procedures of methods. In planning this project, considerations were made for each of the six onion layers (see figure 4) each of which are discussed in detail in the following subsections.

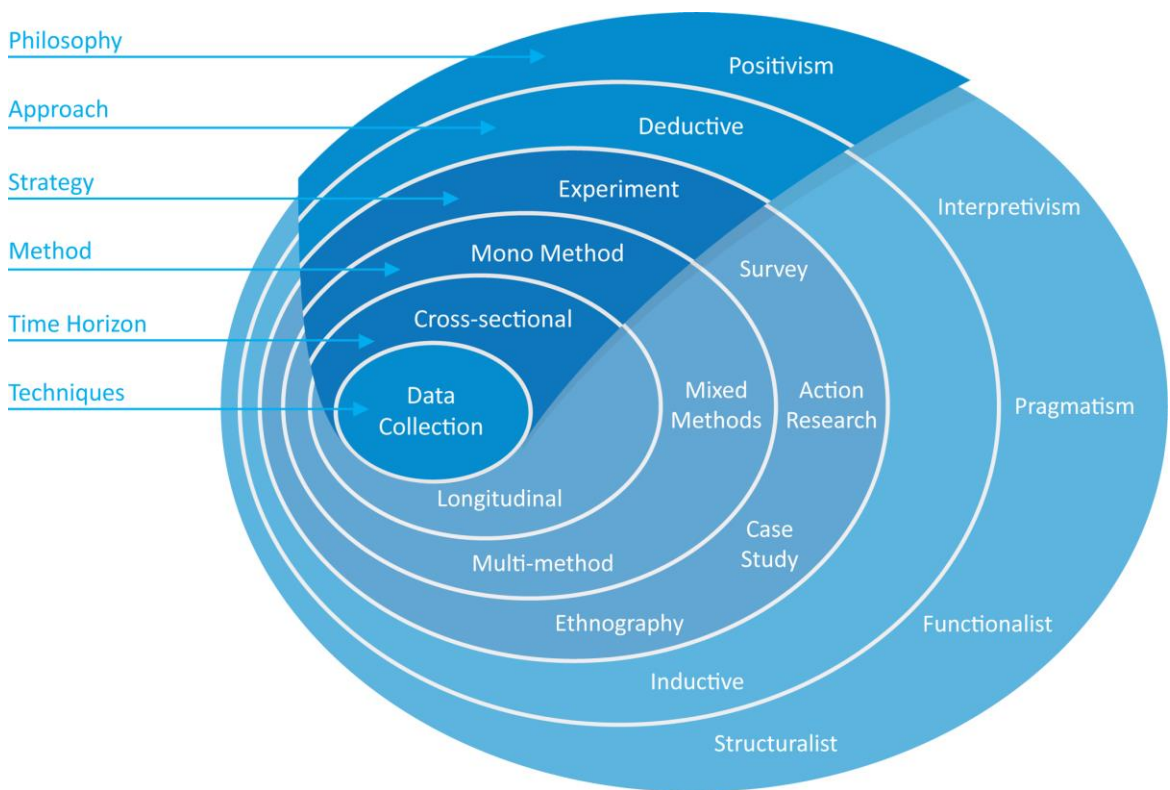


Figure 4: Saunders' adapted research onion [51]

3.1. Philosophy

Philosophy is an aspect of the research methodology that is largely obscured, though Slife and Williams [52] believe it to be an important and influential practice that should be considered by the researcher. Philosophies also help to justify the choices made in the subsequent layers of the onion model.

A research philosophy describes a belief about the ways in which phenomenological data should be gathered, analysed and used. Others [51] have defined it as “the

development of knowledge and the nature of that knowledge”. In either case, the two terms that encompass various research philosophies are epistemology and doxology. Epistemology defines what is *known* to be true; doxology defines what is *believed* to be true. As Davison [53] explains then, the purpose of science is to transform the believed into the known: doxa to episteme. This research operates in the domain of epistemology, and serves to provide evidence about the effect of TDD on the development of software in order to substantiate the doxological beliefs outlined in the literature review.

3.1.1 Positivism and Interpretivism

There are two major epistemological schools of thought in the western tradition of science: positivism and interpretivism [54]. Levin [55] defines positivism as the view that reality is stable and can be described from an objective viewpoint, without interfering with the phenomena being studied. It believes that reliable data can only be obtained through observation by imposing a distinct structure on the methodology. On the other hand, interpretivism is the view that reality can only be fully understood through intervention and subjective interpretation.

Vreede [56] recalls that interpretive research used to be the norm in software engineering research until the late 1970's. However, the significant majority of research now conforms to that of the positivist viewpoint [57]. As it relates to this research, there is a conscious effort to avoid what is considered philosophical monism: the insistence on using a single research philosophy. A positivist viewpoint is largely assumed to the extent that the methodology will have a distinct structure in order to obtain objective and reliable results regarding the effect on quality and productivity, but the researcher will also intervene in part to fully understand the subjective effects of test-first development (i.e. attitude and mind-set). It should be noted that the choice to adopt both positivist and interpretivist viewpoints is not as a result of an inability to determine their advantages and disadvantages. On the contrary, Kaplan and Duchon [58] believe that a combination of viewpoints improves the quality of research.

3.2. Approach

In the context of a methodology, the approach refers to the types of reasoning that can be applied when formulating and designing a research project. There are two different approaches to consider: inductive and deductive. An inductive approach to research is concerned with generating a new theory from collected data, and will typically use a question set to narrow the scope of study. On the other hand, a deductive approach to research is concerned with testing existing theory, and is typically prefaced with a hypothesis [59]. This research will apply a deductive approach because it embraces a number of scientific principles that align with the positivist philosophy, hence a number of suppositions will be created based on the theoretical expositions and findings from the literature review.

3.2.1. Hypotheses

A one-tailed (or directional) hypothesis determines whether there is a relationship between variables in one direction; a two-tailed (or non-directional) hypothesis determines whether there is a relationship between variables in either direction. Ruxton and Neuhäuser [60] argue that a hypothesis should only be one-tailed when the researcher is confident, based on subject knowledge and the available literature, that a directional difference exists. Otherwise the hypotheses could be too distinct and negatively impact the verity of the conclusions. The hypotheses being tested in this research are:

- H1. TDD improves the quality of code
- H2. TDD affects the productivity of the programmer
- H3. TDD encourages more refactoring
- H4. TDD encourages loosely-coupled code

It can be observed that hypotheses H1, H3 and H4 are one-tailed. This is because the literature reviewed in the previous chapter unanimously agrees that the principles behind TDD leads to higher quality software, more code refactoring and looser coupling between classes. Alternatively, hypothesis H2 is two-tailed because the effect on productivity has been documented as both increasing and decreasing (see Productivity in subsection 2.3.4).

It is also important to note that deductive research places emphasis on causality. Causality (or causation) is defined by Granger [61] as the efficacy that connects one process with another, where the first process is partially responsible for the second, and the second process is dependant on the first. In simpler terms, it describes the relationship between cause and effect. It can be observed that all of the aforementioned hypotheses are worded such that they imply causality between TDD and the parameter being measured.

3.3. Strategy

The strategic portion of a methodology refers to the ways in which research questions are answered. This research methodology is structured according to the experiment strategy, which is broadly used to evaluate solutions to problems. There are two distinct phases: an exploratory phase wherein measurements are taken to identify what questions should be asked of a system, and an evaluation phase wherein the questions are to be answered. One researcher [62] adds that a good experiment should start with a list of questions that the experiment is expected to answer. In this case, the literature review has produced questions and findings that will be tested through the hypotheses outlined previously. From a philosophical standpoint, the experiment strategy has also been chosen because it aligns with the thinking of positivism, in that results should be obtained by assessing the causes (i.e. the point at which tests are written) that influence outcomes.

That being said, a number of strategies may be used to answer a research question. Being that the development of software is the subject of this research, the process strategy is also used to some extent. The process strategy serves to understand the processes used in the completion of software engineering tasks. Amaral [62] believes them to be most useful in the study of activities that have human involvement, offering a number of examples including the evaluation of human-computer interactions and the understanding of cognitive processes. Another suggested example wherein the process strategy is considered useful is the design and construction of software, which overlaps with the build strategy. In the context of software engineering, the build strategy describes the building of an artefact (hardware or software) to demonstrate that it is possible. Within the scope of this

research, the build strategy is used as the vehicle of the experiment in order to compare software development processes.

3.4. Methodological Choice

Research can be categorised as being qualitative, quantitative or a mixture of the two. Slevitch [63] explains that qualitative research is based on the ontological view that reality depends on mental structure and activity. Alternatively, quantitative research is based on the ontological view that reality exists independently of human perception. In practical terms, qualitative research is used to identify underlying opinions, reasons and motivations, whereas quantitative research is used to generate numerical data that can be manipulated into statistics. To test the hypotheses, this research will collect and quantify data objectively using a highly structured method. Therefore, the methodological choice is considered quantitative.

Unquestionably, however, methodological choices are not as discrete as they may first appear. Newman and Benz [64] clarify that qualitative and quantitative research should not be viewed as distinct categories or dichotomies; instead they represent different ends of a continuum. Mono and multi-method research describe the singular use of qualitative or quantitative method(s) and reside on the ends of the continuum. On the other hand, mixed methods use both qualitative and quantitative methods, and they reside in the middle of the continuum. This research can be categorised as mono method because it uses a single quantitative method to answer the research question.

Often the distinction between quantitative and qualitative is made in terms of the type of data being collected (i.e. words or numbers). However, Creswell [65] argues that the distinction should be made in terms of the philosophical assumptions made by the researcher. This research assumes a largely positivist philosophy, which Polit and Beck [66] assert to be associated with quantitative methods.

3.5. Time Horizon

Time horizons denote the fixed period of time on which a research methodology is based [51]. As it pertains to the onion model, there are two horizons to consider:

cross-sectional and longitudinal. A cross-sectional methodology is one that makes comparisons at a single point in time, whereas a longitudinal methodology is one that makes comparisons over a prolonged period of time.

This methodology chooses to adopt a cross-sectional time horizon because it allows multiple variables to be compared at the same time (e.g. quality and productivity). The decision was also forced somewhat due to the time constraints imposed by the nature of the research. Without any deadlines to consider, a longitudinal horizon may have been preferable for testing claims that TDD increases productivity in the long term [49], [50]. These horizons also provide concrete information about causal relationships [67], which would be beneficial in testing claims by Erdogmus et al. [18] that quality and productivity is affected by the number of tests written, rather than the test-first feature of TDD. Nevertheless, the cross-sectional horizon is sufficient for testing the hypotheses outlined in subsection 3.2.1.

3.6. Data Collection

The final layer of the research onion model refers to the data collection techniques used by the research methodology to gather information [51]. In the case of this project, secondary data was collected as part of the literature review to identify the ways in which TDD affects the development of software. A number of credible sources were cited, including mostly academic journals and books, and cross-referenced with each other such that general conclusions could be drawn. Accordingly, the purpose of this subsection is to outline the techniques for collecting primary data to test the hypotheses, which were identified after an analysis of the literature review findings.

3.6.1. Code Coverage

Code coverage measures the extent to which functional code is tested by a particular test suite [68]. A piece of software with high code coverage has been subjected to thorough testing and has a lesser chance of containing bugs, unlike software with low code coverage.

There are a few different types of code coverage measurements. A test coverage measurement is calculated by exercising a code base with a test harness, capturing

information on code that has been ‘touched’ during the lifetime of the testing process. The most common form of measurement is line (or statement) coverage. Line coverage indicates that a particular line of code was exercised during a test run. For example, a method wherein eight of ten lines are exercised has a line coverage measurement of 80%. The process works at a higher level too. Consider a class wherein 60 of 100 lines are exercised in testing; the line coverage would be calculated as 60%.

Coverage can also be measured in terms of branches. Branch coverage collects information on the number of decision points, such as conditional statements with logical ‘AND’ or ‘OR’ operators, that are exercised during a test run. Much like line coverage, a method with two branches that are covered by tests can be considered to have 100% branch coverage.

Supplementing a unit testing approach with coverage measurement offers many advantages, adding precision and depth to an already beneficial process. Nonetheless, Glover [69] argues that code coverage data should be processed with some conjecture. A high percentage of coverage does *not* infer high quality code, rather that a lot of code was exercised. In other words, it does not necessarily ensure that the software is free of bugs (though they are far less likely).

However, code coverage information can be leveraged to achieve higher quality software because it exposes code that has yet to be adequately tested. When examining code coverage measurements, areas of suboptimal testing can be observed and understood, which is useful for a couple of different reasons. Firstly, knowing whether some code has been tested allows programmers to accurately predict the time needed to modify existing code: untested functional code is difficult to understand and harder to modify safely, whereas tested code increases the programmer confidence and facilitates a better assessment of the code quality. Secondly, monitoring coverage helps to determine code that is being written without an associated test. Consider a scenario wherein testing is routinely being neglected: code coverage will expose this trend, allowing proactive action to be taken (e.g. training programmers). For the most part, this does not apply to TDD because programmers cannot write functional code without a preceding unit test.

Nonetheless, being that code coverage reports are considered to be a valuable resource to both programmer and team, it will be measured as part of this research. Hypothetically, the requirements implemented using TDD should achieve 100% code coverage because of the aforementioned rule: no functional code without a failing test. However, Beck [22] acknowledges that some programming tasks cannot be driven by tests exclusively. He offers security software and concurrency as examples where TDD is “insufficient to demonstrate that the goals of the software have been met.” As such, there remains interest in comparing the coverage generated by TDD and a traditional test-last approach.

3.6.2. Defect Density

Having established that code coverage is not a direct measurement of quality, the defect density metric will also be used to collect data on quality for hypothesis H1. Defect density is calculated by dividing the number of confirmed defects by the size of a particular piece of software [70]. It is one of the only direct measurements of software quality, and allows the quantitative description of trends in defect discovery, repairs, process, product imperfections and responsiveness to users. Furthermore, Schneidewind [71] states that defect measurements are the basis for quantifying significant software quality attributes, factors and criteria.

Measuring defects with exactness and explicitness is of key importance, however, the definition of a defect remains the source of much ambiguity. A defect is commonly defined as an unintended characteristic that impairs the utility and value of an entity, or any type of imperfection, deficiency or shortcoming. While this definition is correct, though, it is of little help when considering the measurement of software. Alternatively, a defect can be defined as an inconsistency with the specification, as per *IEEE Standard 982.1* [72], but the implication then is that the reference specification cannot have defects of its own. This is not always the case so another approach must be used.

For the purposes of this research, a defect will be defined as any flaw or imperfection in the software product, where a software product is any artefact that is created as part of an application, including source code, plans and associated

documentation [73]. The definition covers a wide span of possibilities and does not eliminate software artefacts that are known to contain defects from experience.

Beyond the definition of a defect, software quality measurement is often inconsistent in the ways that data is collected, processed and recorded. Florac [74] has addressed this issue by proposing a software quality measurement framework wherein data is collected using a selection of defect finding activities at regular intervals. The test cases written as part of this research constitute a dynamic defect finding activity, which will attempt to identify defects during code execution. Code inspections will also be used as a static defect finding activity, and as per the framework, they will occur on a regular basis at the end of each programming session.

Moreover, the framework demands that a standardised report be used to collect information about defects. This is important because an organisation will typically utilise a number of different reports (e.g. problem reports, inspection reports and requirement review reports) to collect such information, hence the conflicting formats makes it difficult to communicate and assess defects.

Despite the variety of ways in which defects are reported and recorded, they all exhibit “remarkable” similarities [74]. The approach taken by the framework to categorise these similarities is to outline a set of common attributes and values that encompass the various types of reports, which also aligns with the thinking of Fenton [70] who applies the “who, what, when, where, why and how” in the context of his software measurement framework. The attributes are carefully chosen such that they are mutually exclusive (i.e. each defect can only have one value per attribute) but are also exhaustive enough to ensure that erroneous or inconsistent counts do not occur.

The following attributes were identified [74] with the intention of transcending the various types of reports generated from defect finding activities. They provide a basis for communicating the meaning of the defect prescriptively or descriptively, as well as the measurement(s) used.

- **Identification:** What software artefact is involved?
- **Finding Mode:** How was the defect found?
- **Criticality:** How critical or severe is the defect?
- **Problem Type:** What is the nature of the defect?
- **Uniqueness:** What is the similarity to previous defects?
- **Urgency:** What urgency or priority has been assigned?
- **Timing:** When was the defect reported?
- **Defects Found In:** What software artefacts caused or contain the defect?
- **Changes Made To:** What software artefacts were changed to correct the defect?

Once all of the defects have been logged, they will go through an initial counting process before the density is calculated by dividing by the size of the software (determining software size is discussed in the following subsection).

3.6.3. Lines of Code per Time Frame

Lines of code (or sometimes source lines of code) is a software metric used to quantify the size of a software artefact by counting the number of lines that appear in the source code. Whilst it can be useful for predicting the amount of effort required to develop software, it is often used after the fact to gauge programmer productivity or maintainability after software has been produced [75]. Lines of code will be used to measure programmer productivity in addressing hypothesis H2 of this research.

However, what constitutes a line is the source of much ambiguity. In Boehm's [37] well-known book *Software Engineering Economics*, the lines of code method counts those lines that are physical, including executable statements, data definitions and comments. On the other hand, Conte et al. [76] defines a line as those that are not a comment or not blank, irrespective of the number of executable or non-executable statements that are on the line.

Depending on the programming language and the line definition chosen, the size of a software artefact can vary greatly. For example, multiple statements can be entered on one physical line in BASIC, PASCAL and C, whereas statements in other languages may span several physical lines, particularly if the programmer is aiming for a maintainable codebase. According to Jones' [77] study, the difference between the two definitions of line counts can be as large as 500%, thus extreme

caution should be exercised when comparing two treatments in a study if the operational definitions are not identical. Otherwise, the results of any comparative analyses would be considered invalid.

For the purposes of this research, the line definition provided by Conte et al. [76] will be adhered to because the source code will be populated with redundant comments that have value only for the experiment being conducted, and as such should not be considered as part of the line count. Explicitly, all physical lines will be counted with the only exception being those lines that are commented. Moreover, as a matter of good programming practice, each statement will reside on its own line. Maintaining the same line definition throughout the course of the data collection will ensure that results are valid.

Nonetheless, it should be noted that productivity is not always an indicator of quality. This is particularly true in the context of software engineering because the product is intangible and difficult to measure directly. For instance, programmers that develop low quality software quickly will appear more productive than those who develop reliable and maintainable software. More does not always mean better, and may ultimately result in increased maintenance costs. This issue has been somewhat addressed by Gulecha [78] who discusses the idea of measuring the test cases written per time frame. He argues that test cases are more indicative of quality than the amount of lines written. However, measuring test cases as part of this research would be an unfair comparison and a poor indicator of productivity because TDD directly influences the number of test cases written by the programmer. Alternatively, Albrecht and Gaffney [79] suggest measuring the number of requirements completed per time frame, which is a metric that might be used as part of this research because it provides an additional point of comparison and does not require any additional data be collected.

3.6.4. Refactoring Density

Refactoring is a change to the internal structure of the source code without affecting its external behaviour [34]. Identifying and recording the number of refactors has been the subject of much research in the past decade, primarily because refactoring is the prerequisite to various different software analyses. Some

examples include error detection and capturing the intent of change. However, as it pertains to this research, the number of refactors is being counted in line with hypotheses H3.

As discussed in the literature review, Beck [22] has identified three different approaches to refactoring in the case of TDD. In order to make the two development approaches used in the experiment more comparable, the obvious implementation strategy will be used throughout. This encourages refactors to be made naturally as required, rather than forcing the programmer to refactor with each and every test case that is written. Should the 'fake it' strategy be used for the entirety, which would be deeply unrealistic in a real world scenario, TDD would inevitably produce a greater number of refactors, thus making the comparison with another approach obsolete.

Nonetheless, some ambiguity remains around what constitutes a refactoring of the code; not every change to the code can be considered a refactor. This research will adhere to the detection conditions outlined by Weissgerber et al. [80] in their aptly titled study *Identifying Refactorings from Source-Code Changes*. They propose that there are two different types of refactors: structural and local.

A structural refactor includes those that alter the class structure of the software, and specifically includes moving a class, moving a method and renaming a class. On the other hand, a local refactor is defined as those that are performed within a class. These include renaming a method, hiding or revealing a method and adding or removing a parameter. For the purposes of this research, both structural and local refactoring data will be collected and counted. Furthermore, the refactoring count will be normalised using the measured software size so that a comparison can be made between the two development approaches.

3.6.5. Dependency Density

In the context of software engineering, coupling is the extent to which components interdepend on one another [81]. It is generally considered a bad characteristic of a software artefact because changes to one component can have a ripple effect in other components, thereby making code difficult to maintain and reuse. Moreover,

it can inhibit the programmer's ability to test software components in isolation because dependant components must also be included in the execution.

There are a number of different types of coupling associated with procedural and object-oriented programming. The software artefact being developed as part of this experiment will be object-oriented, and as such will focus on structural dependencies. Structural dependencies are those that involve the software architecture, which Beck and Diehl [82] further subdivide into class and method dependencies. Class dependencies describe a relationship wherein one class depends on another to fulfil its functionality; method dependencies describe a relationship between concrete method or data attributes of an object.

With regards to hypothesis H4 of this research, data will be collected on both structural and method dependencies. Dependencies are therefore counted as instances whereby one class instantiates or calls upon another class to fulfil its function. In much the same way as the previously discussed refactoring metric, the dependency count will be normalised against the software size.

3.6.6. Ethical Issues

The nature of the experiment and the data being collected means that there are no ethical issues to consider. Any data used as part of the software artefact being developed, such as personal user information in the database, will be fabricated so as not to conflict with data protection and privacy interests. Moreover, the software artefact will be developed locally such that external access can be restricted. Preventing public access stops others from using it to store potentially sensitive information, which would ultimately be the responsibility of the programmer.

3.7. Experiment Design

The quantitative method used in this research is an experiment: a scientific procedure undertaken to test a hypothesis, demonstrate a known fact, or make a discovery. Experimental design intends to test the impact of a treatment on an outcome, while controlling all other factors that could influence that outcome [83].

There are two types of experiments available to researchers. A laboratory experiment is used to identify precise relationships between a small number of variables. Moreover, they are studied intensively using quantitative analysis techniques with the intention of making generalizable statements that are applicable to real world scenarios. However, Galliers [54] explains that laboratory experiments have a weakness. The extent to which relationships actually exist is limited due to the oversimplified design. In other words, the laboratory environment is incognisant of many variables that would exist in reality.

Field experiments extend laboratory experiments into scenarios that would occur naturally in the real world. As a result, this mitigates the extent to which results can be criticised for being contrived. Nevertheless, a field experiment cannot be used in this research because it would require experimenting on real programmers in their work environment, and organisations are hesitant to allow such experimentation due to the costs incurred. Bergersen [84] was able to overcome the limitation by compensating (\$40,000) programmers for their time, but this is not viable in the case of this research. Opponents of field experiments also argue that it is difficult to attain a sufficient level of control such that results are replicable by others. On the other hand, Čaušević et al. [85], [86] have experience with both laboratory and field based TDD experiments, and they observed “nearly the same behaviour” in both studies. Therefore, the use of a laboratory experiment may not be an inferior choice.

The decision to use an experimental method reflects the choices made in existing TDD literature. Background research conducted as part of this project revealed that 90% of studies use quantitative methods that could be considered experiment-based, with the remainder being mostly survey-based. Comparing the effects of two different treatments usually involves the administration of each treatment to a different group of participants and documenting the differences. Laboratory experiments [17], [18], [85], [87] in the existing TDD research were typically conducted with students in an academic environment, and involved a random sampling process. Alternatively, field experiments [16], [86], [88] were typically conducted with professionals in industry, and involved a non-random sampling

process. Due to the fact that significant inter-subject variability exists in each group (some participants change and others do not), inferential statistics are required to determine if an effect actually exists. According to Foster et al. [89], this is problematic when generalising results from the group average to the individual subject who should benefit from the research.

One way to eliminate inter-subject variability is to apply two treatments to a singular individual, allowing any effects to be observed directly. Statements about other individuals could then be made through the process of replication. This is called single-subject experiment design, and is the experiment design that will be used in this research. However, many single-subject designs do not factor in the learnability of the participant, and apply treatments sequentially. As a result, the participant becomes more familiar with the treatment during the experiment, limiting the reliability of the results. Barlow [90] proposes the idea of alternating treatment as a countermeasure. As the name suggests, treatments are alternated so conditions that might affect data (other than treatments) are counterbalanced as the experiment proceeds. Because confounding factors are neutralised, the effects corresponding with each treatment should be attributable to the treatment itself, thereby allowing a direct comparison between two (or more) treatments.

3.7.1. Variables

This research is measuring the extent to which TDD affects the development of software, and the experiment serves to determine this by comparing it against a more traditional testing approach. As with any scientific experiment, there are a number of different variables involved.

Independent variables (or treatments) are those that are manipulated by the researcher. In the case of this research, these are the two development approaches being compared. Inevitably the test treatment is TDD, but there are decisions to be made about the details of the control treatment. Similar experiments [17], [87] used a development approach called iterative test-last (ITL) that is the same microprocess as TDD, except the test cases are written after the code. An alternative approach taken by experimenters [16], [18] is traditional test-last (TTL). This is closely modelled on the waterfall, and involves writing test cases after the

entirety of code for a requirement has been implemented. A comparison of the two control treatments is shown in figure 5.

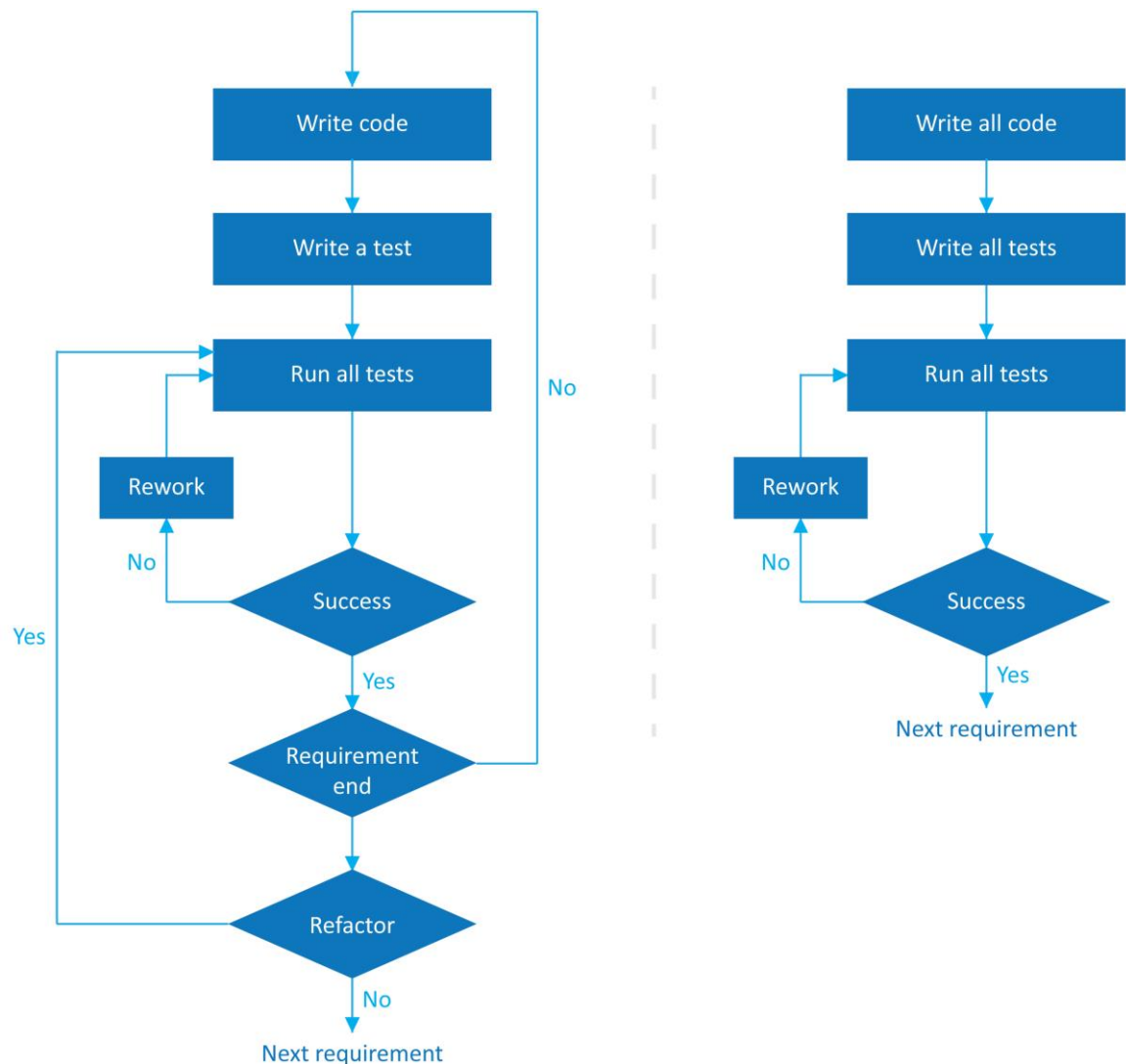


Figure 5: Comparison of Iterative test-last (left) and traditional test-last (right) [87]

TTL was chosen as the control treatment for this research because it is a better representation of the development approach conventionally used by programmers, particularly in terms of web-based development. It also allows for a comparison of the number of tests written as it reflects that fact that testing is often considered an afterthought, thus the expectedly lower number of tests can be compared with the theoretically higher number of tests produced by TDD. Should the research have aimed to test only the point at which tests are written (as opposed to all of the features of TDD) then the ITL treatment would have been preferable because it is a near identical microprocess to that of TDD.

Dependant variables are those that respond to a change in the independent variable, hence they form the main observations of the experiment. As it pertains to this research, the dependant variables are those measureable parameters outlined in the hypotheses: quality, productivity, refactoring and coupling.

Other variables that might affect the reliability of the results, such as extraneous and confounding variables, are largely controlled as a result of the experiment being conducted in a laboratorial environment. In other words, factors that would otherwise affect the results in the real world are not present.

3.7.2. Materials (Hardware and Software)

The materials used for the experiment were chosen with replication in mind. In the context of methodological design, the extent to which the experiment can be replicated is important because it increases the significance of the results. If a treatment is truly effective then the long-term averaging effect of replication will reflect its experimental worth [91]. In addition to the significance of results, experimental replication reduces the variability between results and allows conclusions to be drawn with a higher level of confidence. It is particularly important to promote future replication of this research because of the single-subject design (i.e. it does not have a large experimental population).

One of the factors conducive to a replicable software engineering experiment is high availability of the hardware and software used. For that reason, free and open source products have largely been chosen to develop the software artefact and record the results.

The experiment will involve the development of a web-based software artefact because the tools are more freely available than those associated with traditional desktop application development. In terms of environment configuration, the WAMP (WIndows Apache MySQL PHP) stack will be responsible for providing the local server environment because the hardware being used during the experiment is operating Windows. The database will be administered using phpMyAdmin.

The Laravel MVC framework will provide the underlying software architecture. Not only has Skvorc [92] demonstrated it to be the most popular free and open source framework, it has the added advantage of coming pre-packaged PHPUnit: an integrated unit testing framework. However, PHPUnit does not provide the analysis tools required for code coverage reports. As a countermeasure, the Xdebug PHP extension will also be included for code coverage analysis. While other tools are available (e.g. Coveralls and Spike PHPCoverage) they do not integrate with the PHPUnit testing framework as effectively as Xdebug, which also has a larger amount of documentation available.

The front-end of the software artefact will make use of the Bootstrap framework, which is also free and open source. Being that the presentation of the front-end is not being tested as part of this research, the framework serves to accelerate development time.

Microsoft Excel is the software to be used for recording all of the data collected, and while it is acknowledged that Excel is not free, it can be easily substituted for alternative spreadsheet software.

4. DESIGN

The design chapter details the software artefact that will be developed as part of this research, and given that it is a vehicle for collecting experimental data, some of the factors that would normally be considered when designing software do not apply here. For example, the majority of software projects are informed by a set of user requirements. In the case of this research, however, the software is designed to facilitate conditions for a fair comparison of the two development approaches.

It was decided that the software artefact would not be a complete application. Instead, the software artefact that will be created as part of this research is an application programming interface: a collection of clearly defined methods of communication between different software components [93]. Restricting the software artefact to a smaller subset of functionalities means that they can be explored in greater depth, and it also exposes complex implementation details that will allow the development approaches to be thoroughly tested across a range of problem solving activities.

The development of an application programming interface (API) does not, however, imply that the software artefact will be small in size. For experimental purposes, Krejcie and Morgan [94] iterate the importance of large sample sizes such that the margin for error can be minimised, and consequently the level of confidence in the results is higher. Moreover, larger sample sizes increase the probability that statistically significant results will be observed, thus it was important to ensure that the software artefact for this research had enough requirements to implement.

Making considerations for the balance between the number and depth of requirements, it was decided that the software artefact for this research would be a ticket ordering system. On an abstract level, Goldthwaite and Graylin [95] explain that systems which are responsible for ordering tickets to events must fulfil a range of different functionalities with varying complexities (from random generation and resource allocation to security and concurrency). These constitute ideal conditions for the experiment, and the entire list of functional requirements can be seen in table 1.

Requirement ID	Requirement Description
TDD01	Viewing published/unpublished event listings
TTL01	Purchasing event tickets
TDD02	Handling valid/invalid ticket orders
TTL02	Limiting ticket sales
TDD03	Cancelling failed orders
TTL03	Returning order details
TDD04	Pre-computing and logging the order amount
TTL04	Preventing double reservation of tickets
TDD05	Cancelling ticket reservations
TTL05	Tracking reservations/Validating order charges
TDD06	Adapting real payment gateways
TTL06	Viewing order confirmations
TDD07	Generating confirmation numbers
TTL07	Storing the last four card numbers
TDD08	Assigning ticket codes
TTL08	Viewing purchased tickets

Table 1: List of functional requirements for the software artefact

A total of sixteen requirements were identified, and being an even number allowed them to be split equally between the two development approaches (as indicated by the requirement ID). With regards to the way in which requirements were allocated, the intention was to group tasks with similar problem solving activities and divide them evenly between the approaches for the most equitable results. The majority of CRUD-based functionality, for example, was allocated to requirements TDD01, TDD03, TDD05 and TTL01, TTL05, TTL07. To the same effect, the majority of random generation functionality was allocated to requirements TDD04 and TTL03.

Having identified a list of functional requirements, a high-level design of the software artefact can be constructed (see figure 6). This is beneficial to the experiment because it serves as a performance baseline and provides a reference point at which to stop development. As Griffin [96] explains, not knowing when the development cycle should end often results in an increased project duration, and in the case of the experiment, it is important to avoid prolonging development beyond what would be considered valuable data collection because it makes results susceptible to the dilution effect [97].

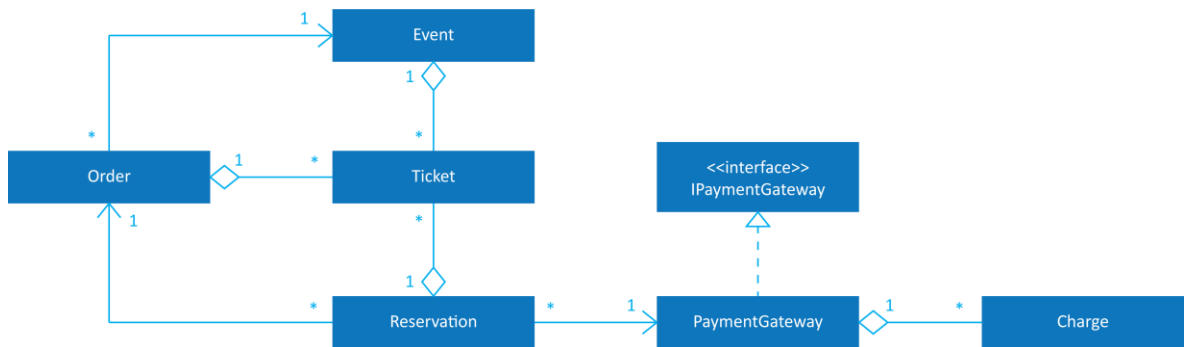


Figure 6: UML class diagram for the ticket ordering system

While the UML diagram is useful for illustrating the relationships between objects, it does not provide any insight into the ways in which these objects interact with one another. Accordingly, a sequence diagram (see figure 7) was devised for the proposed design, and it represents the method calls and data types that are exchanged between objects for the main use case of the API: users ordering tickets to an event.

It is worth noting that the general design of the software artefact may experience minor changes due to contingencies that are encountered during the implementation. Furthermore, much of the lower-level details of the design have been omitted from the diagrams because it is a conflict of interest with the experiment. Predetermining all details of the software design would interfere with the data collected on defects, dependencies and refactoring. Beck [22] also notes that TDD is intended to drive the design of software, and influencing this process with preconceived ideas of the implementation details would invalidate the results.

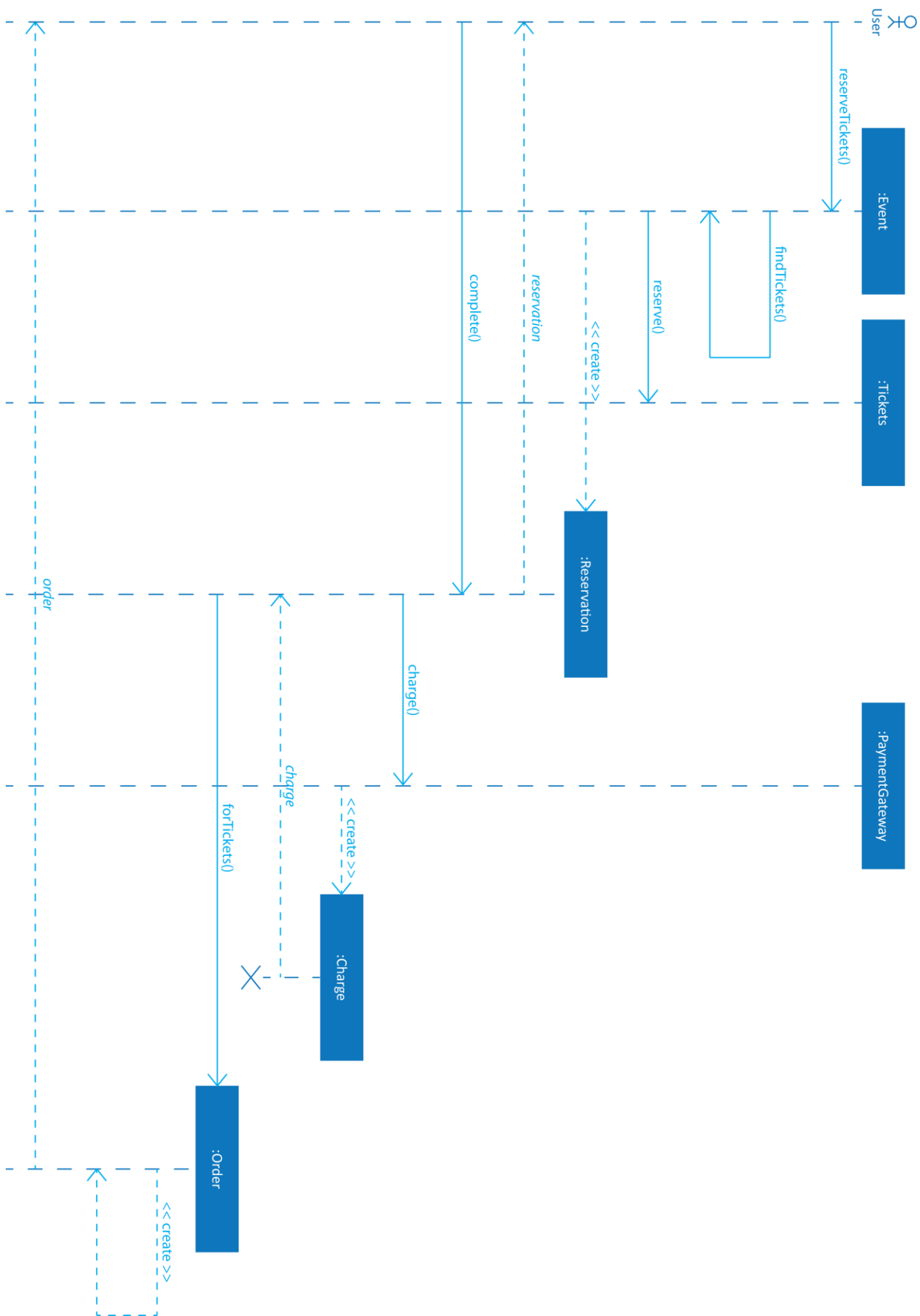


Figure 7: Sequence diagram for the ticket ordering system

5. IMPLEMENTATION

The implementation chapter of this report documents the thought processes and strategies that were undertaken during the course of development. It also makes use of examples from the code to help explain some of the more complex problems, as well as those solutions that are of particular interest to this research.

5.1. Structuring the Tests

The tests written as part of the implementation were structured according to the arrange-act-assert pattern: a simple structural pattern for arranging and formatting code in unit test methods. Grigg [98] explains that this pattern clearly separates the setup and verification steps from what is actually being tested, and it also makes it easier to identify those tests that are testing too many things at once. The arrange-act-assert pattern was chosen for this research, as opposed to other structural testing patterns, because it is considered to be a best practice when using TDD [99]. As a matter of consistency, it was also used for the structuring of the TTL tests; the only difference being the point in the development cycle and the frequency at which the tests were written.

The use of the arrange-act-assert pattern is a three-step process. In the first step, the aim is to arrange all of the necessary preconditions and inputs. This typically involves variable declaration and initialisation. The second step is to act upon the object or method under test by invoking the code, and in the third and final step, assertions are made as to whether the expected results have occurred. An example of the arrange-act-assert pattern being used in the software artefact for this research can be seen in figure 8.

```

function testRetrievingAnOrderByConfirmationNumber() {
    $order = factory(Order::class)->create([
        'confirmation_number' => 'ORDERCONFIRMATION1234'
    ]);

    $foundOrder = Order::findByConfirmationNumber('ORDERCONFIRMATION1234');

    $this->assertEquals($order->id, $foundOrder->id);
}

```

Figure 8: Test case in the software artefact using the arrange-act-assert pattern

Notice that the three steps are clearly delineated using blank lines as a separator, which is one of the principles associated with the arrange-act-assert pattern. The additional visual separation helps to further differentiate between code that is responsible for setup, verification and the object or methods under test [98]. Moreover, it makes the code more readable for programmers who may contribute to the test suite in future.

Another point to mention with regards to the structuring of tests is the use of direct model access: a technique that Chelimsky [100] describes as creating the required models directly inside of the arrange step (see figure 8) as opposed to simulating the user interacting with the browser to create them. The primary advantage of direct model access is that it exercises fine-grained behaviours of a system, when driving them through the full stack would be too cumbersome. It also makes test suite execution significantly faster, which becomes increasingly important as more test cases are written and the suite is executed more frequently. Proponents [100] have argued that it limits code duplication by negating repetitive steps, as well.

Besides performance and the reduction in code duplication, the decision to use direct model access was informed by the fact that the software artefact is an API, which provides back-end building blocks that can be used by the programmer in assembling an application. Without an integrated front-end, it does not make sense to test the way(s) in which users provide data to the model objects. In other words, this is an instance where exercising the behaviours of a system through the full stack would be considered cumbersome and unnecessary.

5.2. Refactoring Strategy

The approach taken with regards to refactoring was to use the obvious implementation strategy. Using this strategy, a solution that was perceived to be correct was implemented straight away (i.e. without going through a process of generalisation) but the test suite was still executed with every addition to the codebase. Obvious implementation is a strategy that is often used by programmers like Beck [22] to speed-up development cycles, but in the case of this research it was more important to use it because it is applicable to both development approaches. Although it is often advocated that the ‘fake it’ refactoring strategy be used when difficult problems are encountered during TDD, using it during the experiment would have produced inequitable results due to an artificial increase in the number of refactors. Similarly, the triangulation refactoring strategy could not be used because it encourages the programmer to write additional tests, and therefore would have influenced the data being collected on the number of tests written for each development approach.

As an example of the refactoring strategy in action, consider the mechanism for creating an order using the software artefact. Initially, the obvious implementation was to create a *forTickets* method that took a collection of tickets, an email and a charge and used these parameters to return an order object (see figure 9). This solution worked as desired and could have remained as is.

```

public static function forTickets($tickets, $email, $charge) {
    $order = self::create([
        'email' => $email,
        'amount' => $charge->amount(),
    ]);

    foreach($tickets as $ticket) {
        $order->tickets()->save($ticket);
    }

    return $order;
}

$order = Order::forTickets(
    $reservation->tickets(),
    $reservation->email(),
    $reservation->totalCost()
);

```

Figure 9: Create order method before refactoring

However, there were two code smells emanating from the *forTickets* method. Tufano et al. [101] describe code smells as a symptom of source code that indicates deeper problems with its design or implementation. In this particular case, one of the smells was due to too many parameters. Having many parameters results in code that is hard to read, and it also makes the method difficult to test. The other code smell is known as feature envy, and it refers to implementations that excessively use methods of another class. This breaks the object-oriented programming principle of encapsulation by creating interdependencies, and it too results in code that is difficult to test. While these code smells are not defects in themselves, they increase the likelihood of defects occurring and should therefore serve as heuristics for when to refactor.

```

public static function fromReservation($reservation) {
    $order = self::create([
        'email' => $email,
        'amount' => $charge->amount(),
    ]);

    $order->tickets()->saveMany($reservation->tickets());

    return $order;
}

$order = Order::fromReservation($reservation);

```

Figure 10: Create order method after refactoring

Considering the nature of the two code smells, it was decided that a preserve whole object refactoring was required. This type of refactoring collapses a list of parameters into one by passing the entire object, as opposed to its individual values. As it pertains to the order creation implementation, the reservation object was passed in as a whole (see figure 10) and values were extracted inside of the method. It should also be noted that the method name was changed from *forTickets* to *fromReservation* to better reflect its new prototype. Overall, the mechanism for creating an order was successfully refactored such that the code smells were removed by changing the internal code structure, but the external behaviour of the implementation remained the same.

5.3. Use of Mock Objects

The underlying theory behind TDD discourages the use of high-level testing [21]. Some test cases, however, require the instantiation of another class as part of the setup, which creates an environment wherein the integration between different objects is being tested. The solution to increase isolation is to mock any objects that are required as part of the arrange step, but are not themselves the subject of that particular test case. As an additional advantage, the use of mock objects increases test suite performance because they avoid any calls to the database that might normally be required when instantiating a real object. An example of mock object usage for the software artefact can be seen in figure 11.

```
function testReservedTicketsAreReleasedWhenAReservationIsCancelled() {
    $tickets = collect([
        Mockery::mock(Ticket::class)->shouldReceive('release'),
        Mockery::mock(Ticket::class)->shouldReceive('release'),
        Mockery::mock(Ticket::class)->shouldReceive('release')
    ]);
    $reservation = new Reservation($tickets, 'joe@example.com');

    $reservation->cancel();
}
```

Figure 11: Test case using mock objects as part of the setup

The method is testing whether reserved tickets are released when a reservation is cancelled, and notice that an array of ticket objects are being mocked as part of the test case arrange step. This is implemented using the pre-packaged Mockery framework. If an alternative approach were to be used with real ticket objects, then they would be created during test execution and would require an interaction with the database. Moreover, it would divert attention from the reservation class, whose *cancel* method is the subject of the test.

```
function testReservedTicketsAreReleasedWhenAReservationIsCancelled() {
    $tickets = collect([
        Mockery::spy(Ticket::class),
        Mockery::spy(Ticket::class),
        Mockery::spy(Ticket::class)
    ]);
    $reservation = new Reservation($tickets, 'joe@example.com');

    $reservation->cancel();

    foreach($tickets as $ticket) {
        $ticket->shouldReceive('release');
    }
}
```

Figure 12: Test case using spy objects as part of the setup

Eventually, the idea of mocking was further refactored by using spies: a variation of the mock object that allows expectations to be set at the end of the test, rather than when the mock object is being created (see figure 12). In the previous figure,

the expectation that ticket mocks received a call to their release method was set at the time the objects were created. The use of spies meant that these expectations could be moved into the assert step of the test case, which is more compliant with the arrange-act-assert structural pattern that has been used elsewhere.

5.4. Selection of Implementation Problems

This subsection documents some of the more interesting or complex problems that were overcome during the implementation of the software artefact. A problem was selected from each of the main parts of the software artefact so that a better understanding can be formed of the A as a whole and how the individual components interact with one another.

5.4.1. Event

The software artefact made use of Eloquent: an object-relational mapping (ORM) tool that creates a virtual object database by converting data between incompatible types, and is pre-installed as part of the Laravel framework. One such object that was relationally mapped to its corresponding database table was the event (see figure 13). By extending the base model, the event object becomes persistent through the use of Eloquent's storage and retrieval functionality.

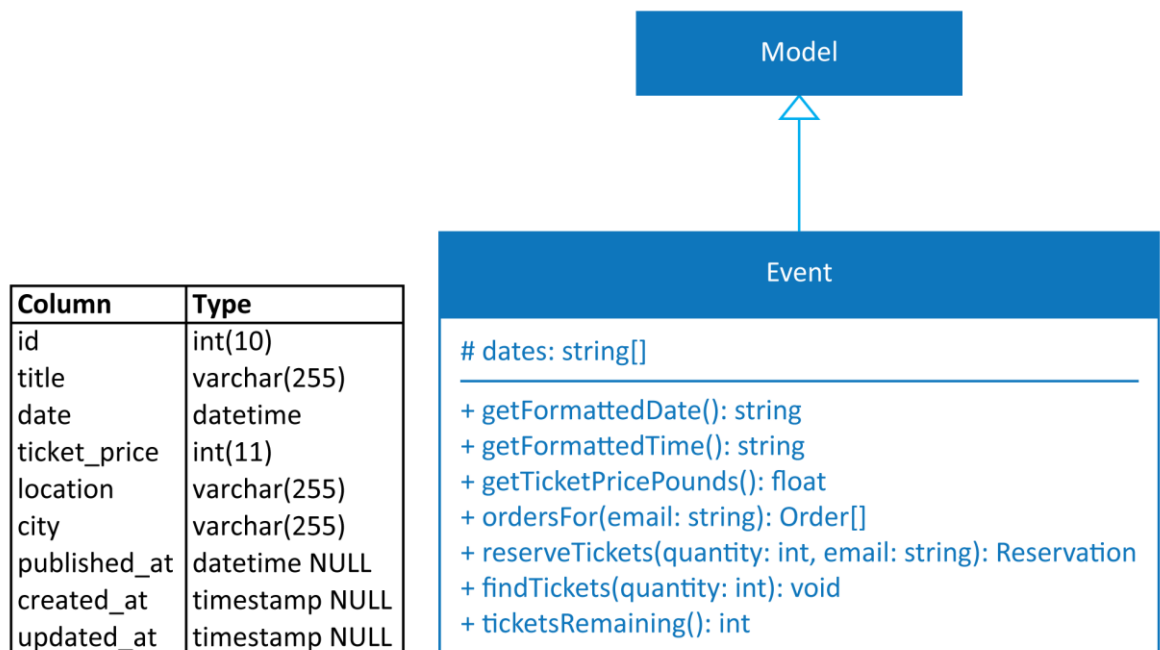


Figure 13: Table structure (left) and UML class diagram (right) for the event model

There was one interesting decision made as to the implementation of the event model. Notice that the price of tickets is stored as an integer data type, which might be considered naïve because it assumes that tickets prices are going to be whole numbers. However, it was an intentional decision to store the price in terms of pennies rather than pounds. If the ticket price were to be stored in pounds, then the data type would need to be a float, but Goldberg [102] asserts that monetary amounts should not be stored using this data type because of the errors associated with floating-point computation. He elaborates that there are only a finite amount of bits available to represent an infinite amount of numbers, thus floating point numbers are typically rounded to an approximate value so that they can be stored. These errors can accumulate and result in inaccurate calculations, which has ethical implications when dealing with currency. Hence, integers were used to store the ticket prices in terms of pennies.

```
public function getTicketPricePoundsAttribute() {  
    return number_format($this->ticket_price / 100, 2);  
}
```

Figure 14: Event method for converting ticket price in pennies to pounds

Nonetheless, it would be unconventional to display the ticket price to the user in terms of pennies. The preferred solution was to implement a *getTicketPricePounds* method inside of the event model, which takes the ticket price in pennies and reformats it into pounds using a simple calculation (see figure 14).

5.4.2. Ticket

The ticket is another model that is relationally mapped to the database and it has a many-to-one relationship with the event model (see figure 6 in the design chapter). However, the design decision to have a ticket model altogether could be contended. It would have been suffice to have an attribute on the events table that stored the number of available tickets, which could be manipulated whenever tickets are added or sold. That being said, the use of a dedicated model allows for more flexibility in terms of what data can be associated with the ticket, and also allows for more extensible business model going forward (e.g. tickets for an event

could be priced differently, rather than letting the event dictate a single price point).

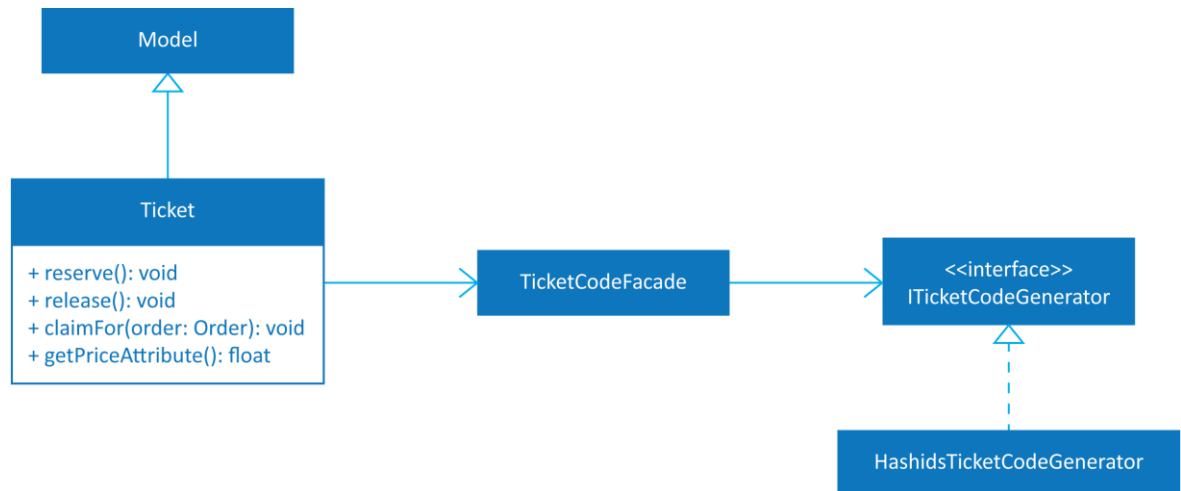


Figure 15: UML class diagram for the ticket model

One of the more complex problems with regards to the ticket model was unique identification. For reference purposes, it was required that tickets have a unique code such that they can be looked-up in customer service scenarios. From an implementation perspective, it was also desirable that the mechanism by which codes were generated could be interchanged with ease. These requirements resulted in an implementation that uses the façade pattern in combination with an interface that describes a general ticket code generator (see figure 15).

Then came the problem of generating the actual ticket codes. One of the main considerations was the birthday problem, which McKinney [103] describes as the probability that at least two people have the same birthday in a room full of randomly selected people. For instance, the probability of duplicate birthdays among a random population of 40 people is 88%. This was particularly relevant to ticket codes because a collision could lead to administrative complications, thus the probability that two ticket codes were the same needed to be minimised.

The first attempt at solving the problem was to randomly generate a six-character long code using any combination of uppercase letters. Calculating the number of unique ticket codes using this implementation (number of alphabetic characters to the power of the set length of the ticket code = 26^6) returns 308,915,776. It might be assumed that with over 300 million unique ticket codes, the chances of a

collision are extremely small. However, supposing that 25,000 tickets are sold (which is entirely viable for a well-established ticket distribution platform) the probability of a collision is 64%.

```
public function __construct($salt) {  
    $this->hashids = new \Hashids\Hashids($salt, 6, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ');  
}  
  
public function generateFor($ticket) {  
    return $this->hashids->encode($ticket->id);  
}
```

Figure 16: Hashids code used to generate unique ticket codes

The solution that was eventually settled upon circumvented the birthday problem altogether. Instead of using random generation, ticket codes were generated by supplying a unique value (auto-incrementing ticket ID) to an encoding mechanism provided by the Hashids PHP library (see figure 16). The encoding process is not random, and therefore an input will always produce the same output. This predictability is good for unique code generation because there can never be a collision, but it also creates vulnerability because the process can be reversed (i.e. the code can be decoded). Preventing the decoding vulnerability involved the use of salts: a secret string stored in an environment variable that is appended to each value before encoding. This means that ticket codes cannot be decoded unless attackers also have access to the salt.

5.4.3. Order

The third and final model that was relationally mapped to the database was the order, and in much the same way as the event model, it too has a one-to-many relationship with the ticket model. Orders are mainly responsible for encapsulating a collection of tickets after a successful reservation has taken place, but they must also provide an order number such that they can be used for purchase confirmation. The order number also serves an administrative purpose, allowing administrators to perform searches using it as a unique identifier.

It can be observed that the situation with order numbers is similar to that of the ticket codes. As a result, the façade pattern and interface combination was reused

so that the mechanism by which order numbers are generated can be interchanged as required (see figure 17).

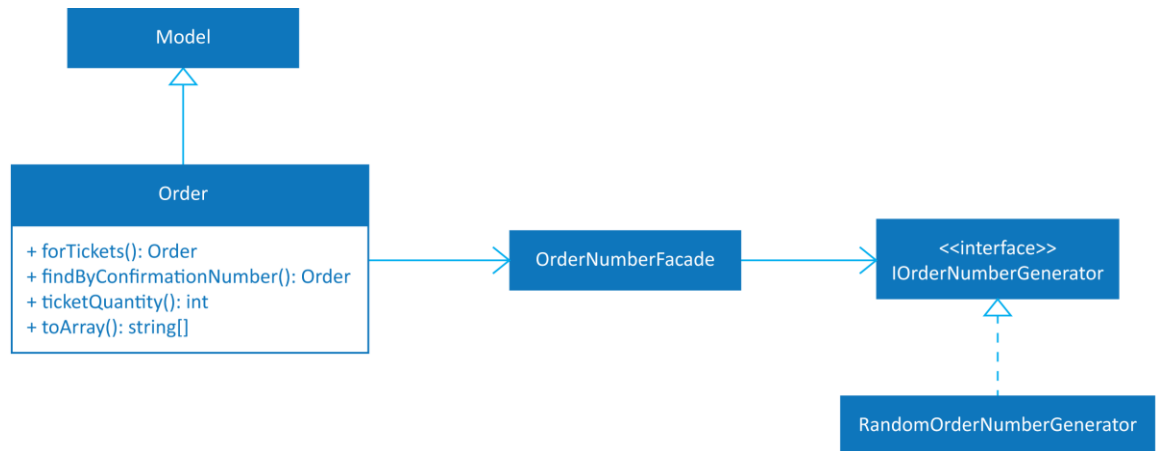


Figure 17: UML class diagram for the order model

Unlike the encoding implementation used for generating ticket codes, the implementation chosen for generating order numbers was based on randomisation (see figure 18). Notice that the *generate* method stores a pool of 32 alphanumeric characters, which omits some ambiguous characters (l, 1, O, 0) because they can be misinterpreted. Generating the order number is then a process of repeating the pool 24 times and appending it to itself, before it is thoroughly shuffled and the first 24 characters are selected.

```

public function generate() {
    $pool = 'ABCDEFGHJKLMNPQRSTUVWXYZ23456789';

    return substr(str_shuffle(str_repeat($pool, 24)), 0, 24);
}
  
```

Figure 18: Randomisation code used to generate order numbers

Using the appropriate calculation (768^{24}) it was determined that the total number of random order numbers that can be generated using this implementation is $1.7728389e+69$. Having such a large number of order numbers is essential because it reduces the probability of a collision, but more importantly because order numbers can be used within the software artefact to access information on purchase tickets through the URL. Accordingly, a vast amount of possible order

number combinations limits the ability of malicious users to guess the order numbers of others.

5.4.4. Reservation

The reservation object is primarily responsible for taking a collection of tickets and an email, before encapsulating this information in an order. However, the original implementation of the reservation object introduced a race condition to the software artefact because it did not do anything to make reserved tickets unavailable to other users before they were purchased. Considering the order of events shown in figure 19, notice the time lag between the point at which tickets are reserved and the point at which tickets are ordered. Being that the reservation object did not make tickets unavailable, it would be possible for two concurrent users to place an order for the same event tickets without knowing.

```
$tickets = $concert->findTickets(request('ticket_quantity'));  
$reservation = new Reservation($tickets);  
  
$this->paymentGateway->charge($reservation->totalCost(), request('payment_token'));  
  
$order = Order::forTickets($tickets, request('email'), $reservation->totalCost());
```

Figure 19: Process of reserving some tickets, charging the user and creating an order

The race condition was addressed by allowing the reservation object to alter the state of a ticket (i.e. setting its reserved attribute to the current date). This is not necessarily a complex implementation worth deliberating in its own right, but writing a test case to exercise the race condition posed a more interesting problem (see figure 20). It is very difficult to mimic two requests being fired simultaneously (that are perfectly staggered such that the ticket ordering processes runs in parallel) while making assertions about what is happening the entire time. Savage et al. [104] believe that achieving this setup in a test is “extremely hard” without the use of error-prone and complex methods that are not worth the effort.

```

function testCannotPurchaseTicketsAnotherUserTryingToPurchase() {
    $event = factory(Event::class)->states('published')->create(['ticket_price' => 500])->addTickets(12);

    $this->paymentGateway->beforeFirstCharge(function ($paymentGateway) use ($event) {
        $this->orderTickets($event, [
            'email' => 'user2@example.com',
            'ticket_quantity' => 1,
            'payment_token' => $this->paymentGateway->getValidTestToken()
        ]);

        $this->assertResponseStatus(422);
        $this->assertFalse($event->hasOrderFor('user2@example.com'));
        $this->assertEquals(0, $this->paymentGateway->totalCharges());
    });

    $this->orderTickets($event, [
        'email' => 'user1@example.com',
        'ticket_quantity' => 12,
        'payment_token' => $this->paymentGateway->getValidTestToken()
    ]);

    $this->assertEquals(6000, $this->paymentGateway->totalCharges());
    $this->assertTrue($event->hasOrderFor('user1@example.com'));
    $this->assertEquals(12, $event->ordersFor('user1@example.com')->first()->ticketQuantity());
}

```

Figure 20: Test for checking that user cannot purchase tickets that another user is trying to purchase

The chosen solution was to leverage the subrequest feature, wherein one request can be fired from within another. In this implementation, the ticket ordering processes are not interleaved. Instead, one instance of the ticket ordering process is consolidated to happen together inside of another ticket ordering process. It might be argued that these conditions are not ideal because they do not exactly represent the situation under test, but it was adjudged to be sufficient because the test should never reach the point where the nested ordering process can complete.

5.4.5. Payment Gateway

One of the more important components of the software artefact is the mechanism by which users are charged and orders are fulfilled. In the case of most web-based applications, payments are handled by a gateway service that is most likely provided by a third-party (e.g. PayPal or Stripe). However, the software artefact was not developed with a particular payment service provider in mind, thus a fake payment gateway was developed such that the API could be fully tested. Meszaros [105] defines a fake as an object that implements an interface but contains fixed data and no logic, it simply returns good or bad data dependent on the

implementation. The fake payment gateway used in the software artefact is not completely devoid of any logic, but similarly it does not emulate the functionality of a real payment gateway.

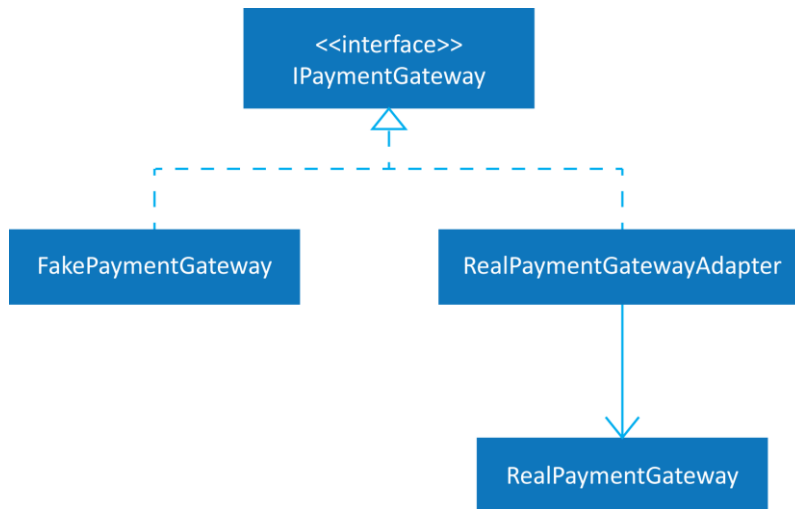


Figure 21: UML class diagram for the payment gateway using the adapter pattern

With extensibility in mind, it was decided that the fake payment gateway would be implemented in such a way that real payment gateways could be interchanged with relative ease. This was achieved by replacing all references to the concrete fake payment gateway with a reference to an abstracted payment gateway interface. On its own however, the abstraction references are not enough to substitute in a real payment gateway because the interface likely has different method names to that of the real payment gateway. The solution facilitated by the software artefact implementation is the adapter design pattern (see figure 21). An adapter class implements the payment gateway interface, but actually calls the methods of a real payment gateway, providing the necessary translation such that a real payment gateway can be interchanged with little effort.

Overall the implementation was completed successfully, despite the various problems that arose during the process. For further details about the implementation that were not necessarily pertinent enough to include in this chapter, such as a list of all the test cases in the suite, please see appendix A.

6. RESULTS

This chapter analyses the experimental results (see appendix B) collected throughout the implementation process using two different statistical techniques: descriptive and inferential. Descriptive analysis focuses on the exhaustive measurement of the characteristics of a defined population to compute a summary value (e.g. a mean average or standard deviation) by describing the relationship between variables. On the other hand, inferential statistical analysis uses a computed characteristic from a sample to make inferences about the population from which it was drawn [106]. It is somewhat forced that inferential statistical analyses be used primarily for this research because of the single-subject experimental design, but descriptive analyses is used where appropriate.

Faherty [107] identifies that inferential analysis may produce observed effects that occur due to sampling error alone. This can be accounted for by considering the statistical significance of results, wherein the observed p-value (probability of obtaining a result that is at least as extreme as that observed) of a statistic is less than the significance level defined for the study. In other words, p-values of an observed effect that fall under the significance level are adjudged to reflect the characteristics of the whole population [108].

Determining the statistical significance is a sequential process. First, the standard deviation of two datasets is calculated using the formula $\sqrt{\sum((x_i - \mu)^2 / (N - 1))}$. At this point, the variance of the results is calculated using the formula $\sqrt{((s_1/N_1) + (s_2/N_2))}$, which provides a t-score that can be used to find the significance by referencing a t-score table. The row with the t-score and corresponding degree of freedom contains the p-value. For the purposes of this research, the level of significance has been set at 0.05 (5%) as is considered conventional by Neyman and Pearson [109]. Therefore, any p-value under 0.05 represents a significant result.

The experimental dataset will be analysed according to the four hypotheses outlined in the research methodology chapter. For each hypothesis, the results are reported and their significance calculated such that it can be accepted or rejected. The null hypothesis (i.e. an acknowledgment that no relationship exists between

two measured phenomena) must be accepted in cases where the alternative hypothesis is rejected.

H1. TDD improves the quality of code

The methodology outlined two metrics for measuring software quality: code coverage and defect density.

H1.1. Code Coverage

In order to assess the code coverage for both development approaches, every test in the suite was first flagged using the @group TDD or @group TTL identifiers. These allowed PHPUnit to execute only those test groups that are specified at runtime, and thus specific code coverage reports could be generated for each approach as required. It should also be noted that any classes provided by the Laravel framework were omitted from the code coverage reports because they were not developed as part of the experiment and would only impair the results.

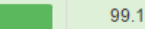
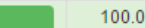
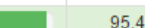
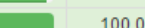
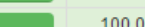
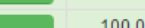
	Code Coverage					
		Lines			Functions and Methods	
Total		99.19%	122 / 123		97.96%	48 / 49
 Billing		100.00%	31 / 31		100.00%	9 / 9
 Facades		100.00%	3 / 3		100.00%	3 / 3
 Http		100.00%	20 / 20		100.00%	4 / 4
 Event.php		100.00%	20 / 20		100.00%	12 / 12
 HashidsTicketCodeGenerator.php		100.00%	3 / 3		100.00%	2 / 2
 IOrderNumberGenerator.php		n/a	0 / 0		n/a	0 / 0
 ITicketCodeGenerator.php		n/a	0 / 0		n/a	0 / 0
 Order.php		95.45%	21 / 22		83.33%	5 / 6
 RandomOrderNumberGenerator.php		100.00%	2 / 2		100.00%	1 / 1
 Reservation.php		100.00%	12 / 12		100.00%	6 / 6
 Ticket.php		100.00%	10 / 10		100.00%	6 / 6
 User.php		n/a	0 / 0		n/a	0 / 0

Figure 22: Coverage report for application code developed using test-driven development

From the results shown in figure 22, it can be observed that TDD achieved a very high level of code coverage. Line coverage was recorded at 99.2% and method coverage was recorded at 98%. When strictly adhering to test-driven practices, code coverage is expected to be nearing 100% because new application code should only be written after an associated test case has failed. That being said, Piwowarski [110] has identified cases where some code blocks are unreachable due

to conditions that are hard to recreate, or conditions that have an extremely low probability of occurrence. This might have explained why complete coverage was not attained in the case of this research, but further investigation reveals that the only method untested was *event* and this just returns a Laravel relationship definition. It would be unnecessary to write an explicit test for such a method because the relationship between models is tested in many other cases.

	Code Coverage					
	Lines			Functions and Methods		
Total		88.62%	109 / 123		81.63%	40 / 49
 Billing		96.77%	30 / 31		88.89%	8 / 9
 Facades		100.00%	3 / 3		100.00%	3 / 3
 Http		80.00%	16 / 20		50.00%	2 / 4
 Event.php		85.00%	17 / 20		75.00%	9 / 12
 HashidsTicketCodeGenerator.php		100.00%	3 / 3		100.00%	2 / 2
 IOrderNumberGenerator.php		n/a	0 / 0		n/a	0 / 0
 ITicketCodeGenerator.php		n/a	0 / 0		n/a	0 / 0
 Order.php		95.45%	21 / 22		83.33%	5 / 6
 RandomOrderNumberGenerator.php		100.00%	2 / 2		100.00%	1 / 1
 Reservation.php		75.00%	9 / 12		83.33%	5 / 6
 Ticket.php		80.00%	8 / 10		83.33%	5 / 6
 User.php		n/a	0 / 0		n/a	0 / 0

Figure 23: Coverage report for application code developed using traditional test-last development

On the other hand, the results for TTL code coverage shown in figure 23 reveals a notably lower level of coverage. Line coverage was recorded at 88.6% and method coverage was recorded at 81.6%. This is by no means a poor level of code coverage – Yang et al. [111] consider anything between 70-80% acceptable – but nonetheless it is notably lower than that of the TDD case. The differences can be somewhat attributed to the number of tests written, with TDD producing over 50% more test cases in total. However, a case can be made for the fundamental flaws associated with traditional testing strategies. It may be difficult to determine the correct number of unit tests to write because at the time of writing the tests a large amount of code will have already been implemented, and the programmer is unlikely to be able to recall with ease all of the newly implemented functionality. Moreover, new code may be perceived to work because of some arbitrary usage testing, thus the motivation to spend time writing additional unit tests is lower.

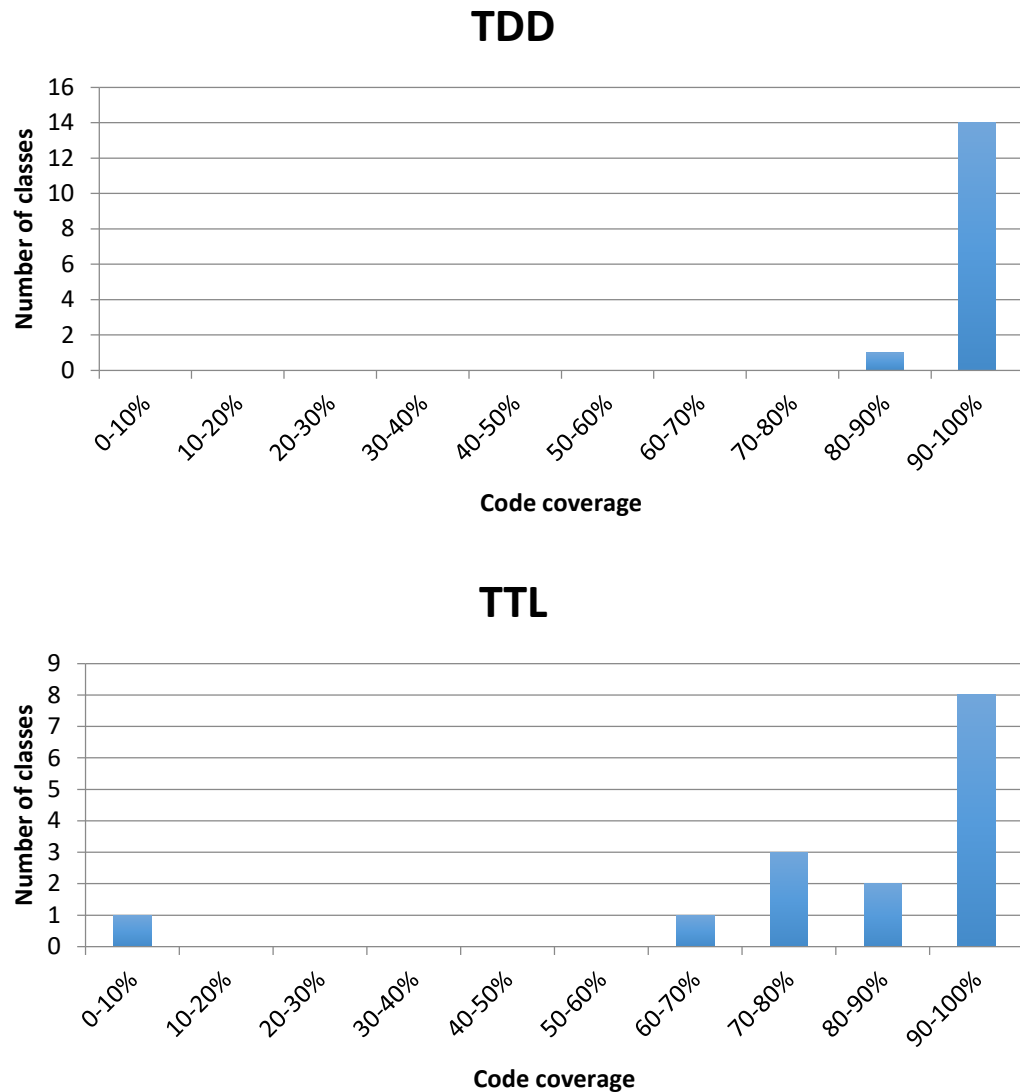


Figure 24: Comparison of test-driven development and traditional test-last coverage distribution

Comparing the results reveals that TDD produced a higher percentage of code covered by tests, and while it was acknowledged in the research methodology chapter that code coverage does not necessarily infer quality, the likelihood of defects occurring in test-covered code is significantly reduced [69]. Furthermore, the code coverage reports provide an additional opportunity for comparative analysis by generating distribution information. With reference to figure 24, it is evident that TDD has a tighter coverage concentration, which is indicative of a consistently applied testing strategy. The results for TTL code coverage, on the other hand, are more dispersed indicating that tests were written in a less consistent manner. These observed differences in consistency can largely be

attributed to the fact that TDD is a clearly defined and structured agile practice, whereas TTL represents a more discretionary approach to testing. This aligns with the sentiments of Maximilien et al. [16] that testing has traditionally been considered an “afterthought”.

H1.2. Defect Density

The second metric used to measure software quality was the defect density (see appendix C for the entire defect log). Even though the results show that there is a clear disparity in the total number of defects generated – five for TDD and fifteen for TTL – it cannot be used as a base for comparison because TTL produced considerably more application code, and therefore the scope for error was that much greater.

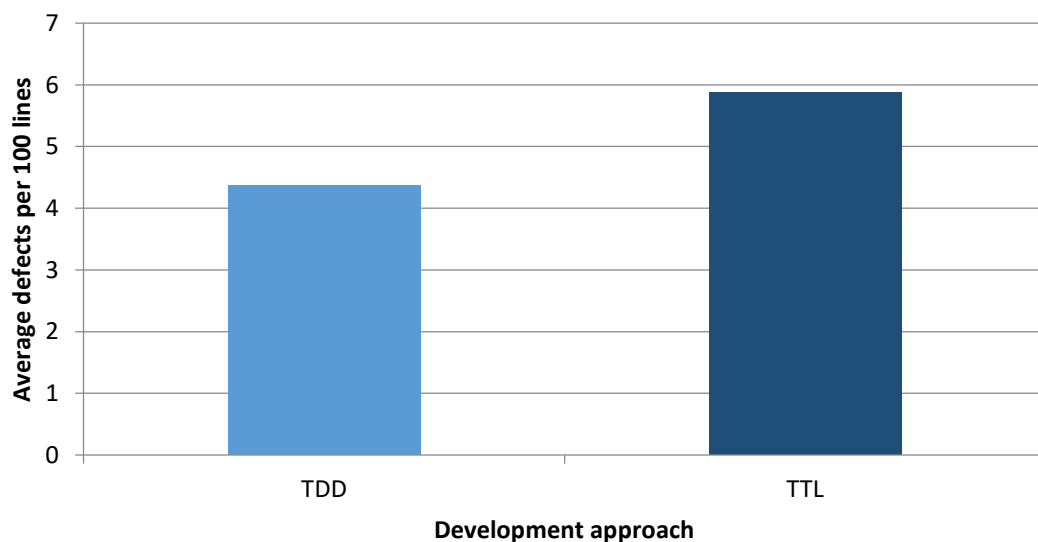


Figure 25: Comparison of test-driven development and traditional test-last defect density

Instead, the defect density for each development approach has been normalised by calculating the amount of defects per every 100 lines of application code. As was to be expected with the greater test coverage, figure 25 shows that the average defect density for requirements completed using TDD was lower (4.4 defects per 100 lines) when compared to those requirements completed using TTL (5.9 defects per 100 lines).

Defects per 100 lines	TDD	TTL
Mean	4.371345029	5.871574207
Variance	37.02409044	21.82308055
Observations	8	8
Pooled Variance	29.4235855	
df	14	
t Stat	-0.553146039	
P(T<=t) one-tail	0.294444022	
t Critical one-tail	1.761310136	

Table 2: Defect density T-test assuming equal variances

In summary, both the code coverage results and defect density calculations imply that TDD produces higher quality software than that of TTL. However, the extent to which these results are meaningful has been calculated in table 2 and the resultant p-value of 0.29444 indicates that the results are not statistically significant. Therefore, the alternative hypothesis H1 is rejected and the null hypothesis (i.e. that no relationship between TDD and software quality exists) must be accepted.

H2. TDD affects the productivity of the programmer

The total number of lines implemented was one of the attributes measured during the experiment, and this can be used to some degree to determine the productivity of the programmer. According to the results, TDD generated 443 lines of code in total and TTL generated 414 lines of code in total. These figures, however, cannot be used to draw concrete conclusions on productivity because the requirements for the application were not uniform in size. Therefore, the total lines of code must be normalised against the development time in order to form a base level of comparability. The results in table 3 show that TDD averaged 34.44 lines of code per hour, whereas TTL averaged 36.18 lines of code per hour. This indicator implies that there is a statistically insignificant ($p = 0.70441$) difference in productivity between the two approaches, which could be used as grounds to accept the null hypothesis.

Total lines of code per hour	TDD	TTL
Mean	34.442971	36.17537779
Variance	92.7085256	67.42175226
Observations	8	8
Pooled Variance	80.06513893	
df	14	
t Stat	-0.387220322	
P(T<=t) two-tail	0.70441578	
t Critical two-tail	2.144786688	

Table 3: Total lines per hour T-test assuming equal variances

However, Gulecha [78] believes test code should be excluded from any productivity calculations because it does not represent any functionality of the product. Making this adjustment to the calculations reveals a greater difference between the two approaches, with TDD averaging 13.74 lines of code per hour and TTL averaging 24.47 lines of code per hour (see table 4). This disparity in productivity was to be expected in consonance with findings from Pancur et al. [17] who report TDD has a negative effect on output due to the increased amount of testing that is demanded of the programmer. The experiment results confirms further that TDD encourages more testing activity with a total of 262 lines of test-related code for the test-first approach, compared with 130 lines for the TTL approach.

Application lines of code per hour	TDD	TTL
Mean	13.73540943	24.46862107
Variance	89.19596423	63.12361155
Observations	8	8
Pooled Variance	76.15978789	
df	14	
t Stat	-2.45978317	
P(T<=t) two-tail	0.02752071	
t Critical two-tail	2.144786688	

Table 4: Application lines of code per hour T-test assuming equal variances

One of the more interesting analyses of the experimental data is the change in productivity over time. Although the previous analyses have been based on averages, plotting the individual results onto a graph makes it possible to observe trends (see figure 26). Treating requirement TDD04 as an anomaly, the observable trend reveals that the lines of code per hour decreases logarithmically over time. Alternatively, treating requirement TTL02 as an anomaly, the observable trend reveals that the lines of code per hour decreases exponentially over time. While a

decrease in productivity is common to both approaches, the functions by which productivity decreases are opposite. It is not immediately clear as to why this is the case, but further analysis suggests that the fewer tests written during the course of TTL leads to defects that become increasingly *harder* to resolve because there is incomplete test coverage, and therefore more time is spent trying to debug. Conversely, the code base should be completely covered by unit tests when using TDD, and therefore problems become increasingly *easier* to solve and debug.

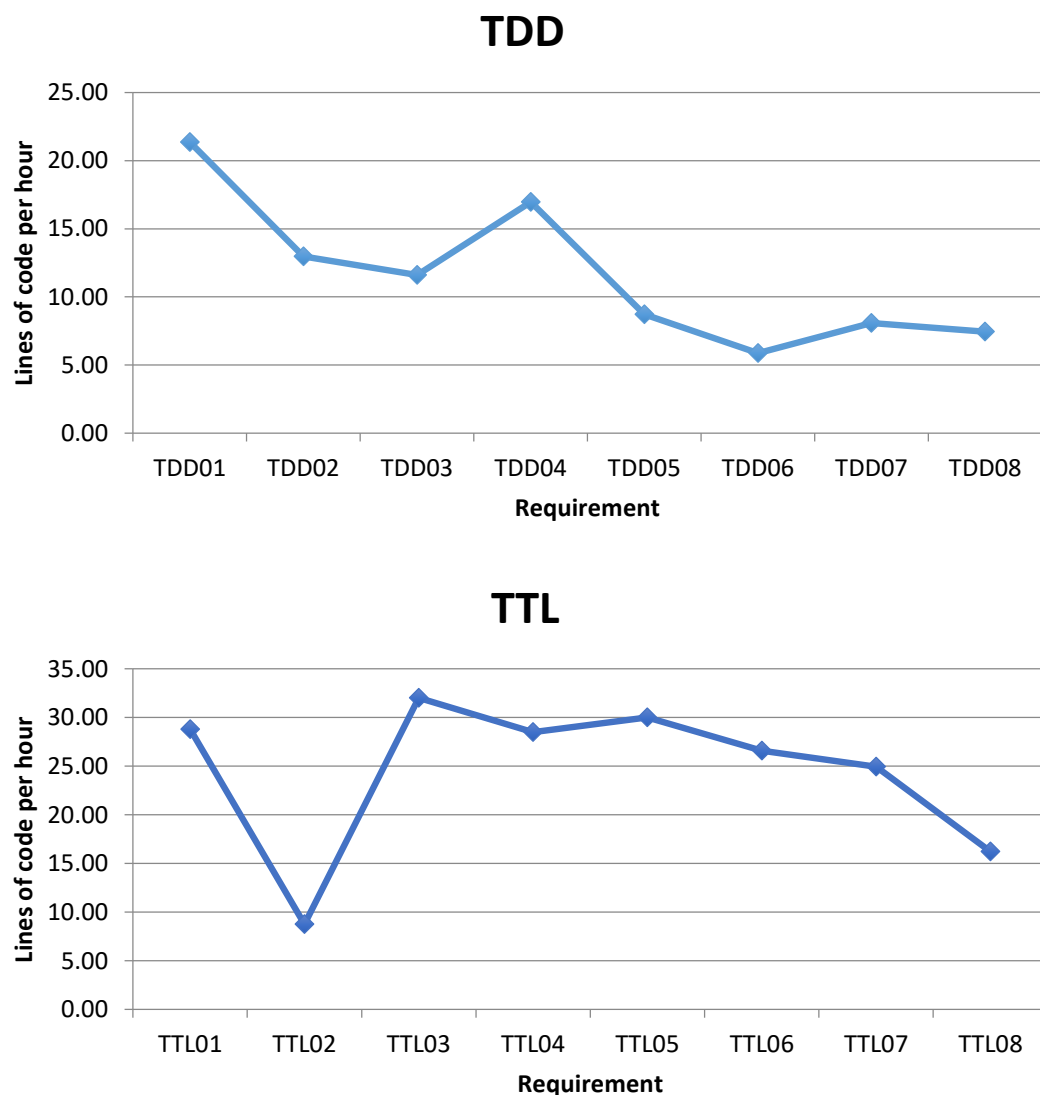


Figure 26: Comparison of test-driven development and traditional test-last average lines per hour

The experiment results represent research that was very much conducted within a cross-sectional time horizon, but inferences can also be made as to the longitudinal

trend of productivity. Referring back to the work by Erdogmust et al. [18] it was identified that productivity declined in the short-term due to TDD, but increased “significantly” in the long-term. The observed trend in figure 26 shows that TDD does in fact decrease in the short term, but there was not enough long-term data to conclude that productivity would begin to increase exponentially at some point. Equally however, the observed trend does not dismiss this possibility.

To summarise, analyses of the experiment results show that TDD leads to comparatively lower programmer productivity than that of TTL. It was also interesting to note that the observed rate at which productivity decreases differs between the two approaches, but the longitudinal trends could not be addressed with any accuracy due to the limited availability of long-term data. With regards to hypothesis testing, the difference in average lines of application code per hour between the two approaches was calculated to have a two-tail p-value of 0.02752 (see table 4). This is considered statistically significant, therefore the two-tailed alternative hypothesis H2 can be accepted and the null hypothesis (i.e. that TDD does not affect the productivity of the programmer) can be rejected.

H3. TDD encourages more refactoring

The detection conditions provided by Weissgerber et al. [80] were used to identify and log refactors during the course of the experiment. Consequently, results have been collected on the number of refactors prompted by each development approach (see appendix D for the entire refactoring log). An initial analysis reveals a clear disparity between the two approaches, with TDD prompting a total of 45 refactors and TTL prompting a total of 11 refactors. However, it was mentioned previously that the requirements were not uniform in size, meaning the total refactoring figures should be normalised against line count for a fair comparison. Making the necessary calculations, the disparity between approaches is reaffirmed with TDD averaging 34.35 refactors per 100 lines of code and TTL only averaging 3.7 refactors per 100 lines of code (see table 5).

Refactors per 100 lines	TDD	TTL
Mean	34.34930606	3.701819095
Variance	247.2974072	11.45003094
Observations	8	8
Pooled Variance	129.3737191	
df	14	
t Stat	5.388921201	
P(T<=t) one-tail	< 0.00001	
t Critical one-tail	1.761310136	

Table 5: Refactoring density T-test assuming equal variances

These results are very much expected because design decisions in TDD are guided by the automated tests, thus it is assumed that the software is pliable and subject to change. Writing a test serves as the *first* step towards deciding what the software should do, and Beck [22] implies that a period of refactoring should follow until the final design is agreed. On the other hand, the software design is conducted upfront in traditional development approaches. This results in fewer refactors because the majority of design decisions have already been made at the point of implementation.

In summary, the observed difference in refactoring between the two development approaches was very large. The TDD approach averaged over 30 more refactors per 100 lines of code compared to that of the TTL approach, which was attributed to the fact that the programmer does not have an upfront idea of the software design, and instead makes decisions as they go along. As a result, the opportunity to refactor presents itself more often when using TDD. Further analysis also suggests that these results are statistically significant having calculated an extremely low p-value of < 0.00001, which is well below the accepted significance level of 0.05 set for this research. This means that the alternative hypothesis H3 can be accepted and the null hypothesis (i.e. that TDD does not encourage more refactoring) can be rejected.

H4. TDD encourages loosely-coupled code

To determine the extent to which code was coupled, data was collected on the dependencies created as part of each development approach (see appendix E for the entire dependency log). A comparison of the dependency count would reveal that TDD promotes tighter coupling with a total of 11 dependencies created,

compared to a total of 8 dependencies created with TTL. To reiterate once more however, the requirements were not uniform in size and the results must be normalised against lines of code. In calculating the dependency density for each requirement it is observed that TDD creates an average of 7.98 dependencies per 100 lines of code, but TTL only creates an average of 3.25 dependencies per 100 lines of code (see table 6).

Dependencies per 100 lines	TDD	TTL
Mean	7.977379512	3.24524442
Variance	50.37013407	22.50575659
Observations	8	8
Pooled Variance	36.43794533	
df	14	
t Stat	1.567870504	
P(T<=t) one-tail	0.06961528	
t Critical one-tail	1.761310136	

Table 6: Dependency density T-test assuming equal variances

These results are somewhat unexpected for a number of reasons. Firstly, proponents [112] of TDD argue that it leads to reduced coupling because programmers are guided to create objects that are needed to pass tests, as opposed to creating objects that are thought to be needed as a result of improper understanding of the requirements. Another reason these results are unexpected is that TDD encourages the use of mock objects. As detailed in the literature review, mocking promotes loose coupling by forcing the programmer to consider the dependency of units.

One supposition as to why TDD creates more dependencies relates to the principle that programmers should write the least amount of code to pass the tests. Often times when using the ‘fake it’ refactoring strategy, the least amount of code to pass the test will involve the instantiation of an object, rather than passing it in through dependency injection [113]. These dependencies should then be refactored into a better design, but instances where the programmer forgets or cannot think of an appropriate refactoring may result in dependencies being left in the software. Nonetheless, this supposition is not applicable in the case of this research because the ‘fake it’ strategy was never used (as per the methodology). In addition, the Laravel framework is built around facades: static interfaces to classes that are

available in the application service container. The option to use these facades meant that there was never an incentive to instantiate objects.

A summary of the experiment results reveals that TDD creates more dependant code compared to that of TTL. Beck's [32] use of the word "tends" in his sentiment that "test-first code tends to be more cohesive and less coupled than code in which testing isn't a part of the intimate coding cycle" implies that these results may not be entirely unexpected, however, though the reasons for this observed effect could not be explained through an analysis of the data. The statistical significance of the results also has no relevance here because the observed effect was in the opposite direction to the alternative hypothesis. Therefore, the alternative hypothesis must be rejected and the null hypothesis (i.e. that TDD does not encourage loosely-coupled code) is accepted.

7. EVALUATION

With the completion of the experiment and an analysis of the results comes an opportunity to evaluate the extent to which the research was a success. In making the following evaluation, any output from the report and its deliverables are to be compared with the objectives identified in the introductory chapter (see subsection 1.1). Each objective serves as a success criterion, and thus constitutes its own part of the evaluation.

Objective O1 called for a better understanding of test-first development approaches. This was addressed by the literature review conducted in chapter 2, wherein a comprehensive investigation of the underlying theory and principles behind TDD ensued, combined with comparisons to traditional development approaches where appropriate. Relevant studies from other researchers were also included to support arguments for both proponents and opponents of the practice, allowing for a critical but ultimately balanced understanding to develop. With that being said, it is argued that objective O1 was met comprehensively, though it would have been preferable to explore the context in which TDD operates in greater detail.

Objective O2 described a comparison between a test-first and test-last approach to software development. Addressing this object was the responsibility of many chapters. The literature review identified the terms on which the development approaches should be compared, before the research methodology reformatted these into a collection of hypotheses and detailed the experiment that would facilitate the comparison. Design and implementation chapters provided further detail as to the software artefact itself, and documented the functional requirements from which data would be collected. Once more it is argued that objective O2 was sufficiently met, but improvements could have been made by including a qualitative method, which would have allowed for more data to be collected on programmer attitudes towards TDD.

Objective O3 sought to evaluate the effectiveness of TDD by analysing a result set. With thanks to a successful data collection process during the implementation of

the software artefact, a comprehensive results analysis could be conducted in chapter 6 of this report, using figures and tables as supporting material as appropriate. Beyond the descriptive and inferential analysis techniques used, results were also put through a process of hypothesis testing calculations to determine their significance. This allowed the effectiveness of TDD to be assessed with a high level of confidence, thus it is argued that objective O3 was met with a resounding success.

Recall that the overarching aim of the research was to determine the effects that TDD has on the programmer and the development of software. Considering the justifications made in meeting all of the success criteria (i.e. the objectives) it must also be adjudged that the aim of the research was met successfully, and the results of the experimental study showed that TDD does improve the quality of software and encourages better design through more refactoring. However, it also reduces programmer productivity and increases the coupling between objects.

8. CONCLUSIONS AND RECOMMENDATIONS

The research began with a literature review and concluded with some notable findings. It was first identified that TDD is not a testing strategy, but rather an agile practice that describes a micro development approach. A further examination of the relevant literature revealed that test-first development approaches increase the quality of code by increasing test coverage. Such approaches also improve the design of software by demanding a period of refactoring from the programmer. However, one of the recognised side effects was that productivity is inhibited as a result of the increased number of tests cases that must be written.

These findings were carried forward into the research methodology and served as the basis for generating a set of hypotheses. Aligning with a predominantly positivistic philosophy, the method chosen for testing these hypotheses was a quantitative experiment, wherein a software artefact would be created using both TDD and a traditional development approach (that would come to be termed TTL). It was decided that the experimental time horizon would be cross-sectional due to the time constraints, and a single-subject design was used because of the impracticalities associated with using other programmers.

The software artefact itself was the subject of much deliberation. Settling on an API that would be responsible for facilitating ticket orders for events, a list of functional requirements were chosen such that the two development approaches under test could be exercised across a wide variety of problem solving activity with the necessary complexity that can be expected in real-world development scenarios. A process of careful consideration then ensued, with requirements being allocated to a development approach strategically such that they created the fairest conditions for the experiment.

As a result of being integrated in the experimental method, implementing the software artefact was very much a balance between solving problems (expected and unexpected) and recording the data being collected on defects, dependencies, refactors, tests and total development time. Nonetheless, all problems were eventually resolved and a fully functioning software artefact prevailed.

The real value of this research stems from the results generated by the experiment, and following a period of analysis, the hypotheses outlined in the research methodology were tested and subsequently accepted or rejected. It was initially supposed that TDD would increase the quality of code, and this was the observed effect, but it was somewhat surprising that the results were considered insignificant. The reasons for this could not be explained through the result set. On the other hand, the hypothesis that that productivity would decrease was accepted and was attributed to the fact that the programmer spends comparatively more time writing test cases. Furthermore, in what proved to be the most significant result of this research, the hypothesis that TDD increases the amount of refactoring was also accepted with great confidence. The last hypothesis was the only one that had to be outright rejected because the observed effect was in the wrong direction. It was hypothesised, based on findings from the literature review, that TDD would produce comparatively looser-coupled code, but the results showed that TLL generated fewer dependencies. Once more, the reasoning could not be explained through an analysis of the data.

3.2 Suggestions for Further Work

The claims made in this report are based on a single programmer interacting with a small number of systems. Therefore, they have the status of hypotheses requiring more rigorous analysis by experimental techniques. Given this caveat, some avenues for further expansion of the research have been identified. Firstly, acknowledging that this research was conducted along a cross-sectional time horizon, the longitudinal effects of TDD remain to be explored in any great detail [49]. Interestingly the observed trend in productivity data for this research appeared to show that it was changing direction as time progressed, but there was not enough data to make any inferential analyses with an acceptable level confidence. A better understanding of TDD's long-term effects would be particularly beneficial to those companies that deal with large software development projects. It would also help to convince those decision-makers that transitioning to TDD is a worthwhile endeavour, despite the initial training costs and difficulty associated with transitioning away from traditional development approaches.

Another avenue for further research is the relationship between the number of tests and the effect on quality and productivity. Erdogmus et al. [18] explored this relationship to some extent, concluding that the number of tests does affect quality and productivity, irrespective of whether the tests were written before or after the application code. This research cannot confirm these claims with any confidence because the sample size was so restricted, though the results showed that requirements implemented with a larger number of tests generally had fewer defects. Further research that can substantiate these claims has implications for the future adoption of TDD because it implies that the test-first aspect of the practice does not necessarily lead to better software, instead suggesting that better software occurs as a result of increased testing.

9. REFLECTION

As the author, the reflection chapter gives me a chance to look back on the entire experience of completing a research project and discuss any lessons that have been learned along the way. Many of the lessons were learned from decisions made at the beginning of the project because I initially found it difficult to choose a research topic. It would have been considerably easier to pick a subject that I knew well, but being that I was going to commit so much of my time to the project, I wanted to use it as an opportunity to learn something new and relevant. This led me to choose testing after a period of deliberation. My reasoning was simple; testing is a critical part of the software development process that I had not been subjected to in any great detail during my undergraduate education. I then narrowed it down to TDD because it is a somewhat controversial development approach that remains a topic of debate among programmers, thus my research could offer some value to the existing canon of work.

With the selection of a topic came the realisation that I would be wholly responsible for deciding what should be researched as part of the project and when this research needed to be completed. Project planning proved to be a difficult endeavour for myself because I initially felt that the time could be better spent elsewhere. However, neglecting to engage in a thorough planning process meant that much of the research activity was misguided, mistake-ridden and ended up taking more time than necessary: a reminder that poor preparation produces poor performance. I eventually made the necessary adjustments by spending more time thinking about what arguments I wanted to construct before writing, and in the end I actually managed to anticipate the size of the project very well. A complete project was delivered using near enough all of the available time, along with a fully-tested software deliverable that is robust enough to use in real software developments.

The process of researching and writing the report presented its own set of challenges. Towards the beginning of the project I would rewrite each chapter numerous times because I was unsatisfied with the way in which it read. Chasing a concisely written and cohesive body of work would be considered a desirable trait

within academia, but under these circumstances the need for perfectionism was negatively affecting my progress. Continuing the process of rewriting constantly would only have resulted in the latter chapters being poorly written, or possibly even uncompleted altogether. For these reasons, my writing strategy changed about half way through and I began to balance quality and quantity such that each chapter was completed to its fullest extent and addressed in an appropriate amount of depth. It would have been nice for me to research and write each chapter with the quality exhibited in the literature review and research methodology, but doing so was not possible inside the project time constraints.

Conducting my own primary research was a learning experience in itself. Most of my undergraduate education has consisted of secondary research into the work of others, but conducting my own research activity was a refreshing change, and knowing that I was able to produce a complete result set was very rewarding. One of the lessons I learned was the importance of experimental design. There was only enough time to conduct the experiment once, and given that data was being collected on the development approaches themselves, it was critical that I knew what to measure in advance of the implementation process. Taking cues from the previous cases where there was very little planning, I spent a vast amount of time deliberating the ways in which data would be collected to ensure I had results that were able to prove or disprove the hypotheses.

Unto this point I have reflected upon the process of choosing, planning, researching and writing the project. I would argue, however, that the most valuable lessons I have learned, and those that will serve me greatest going forward, are those learned from the subject matter. Recall that I wanted to do a testing-related project because it was new to me; I started with no preconceptions about the ways in which software should be developed or tested. The undertaking of this project has resultantly given me a greater appreciation for these things and I am now convinced that testing is an essential part of the development process. But the question remains, what development approach will I use going forward and to what capacity will testing be a part of this? Bringing this research full circle to the literature review, where Gamma [29] discusses the idea that most people who learn

TDD discover that their programming practices changed for good, I can confirm that I am well and truly “test infected”. I will continue to learn and apply TDD wherever possible because the theoretical principles behind the practice are well founded, and thanks in part to the results of this research, it has been shown to have a largely positive effect on the development of software. I cannot see myself reverting to traditional development approaches and testing practices unless explicitly advised otherwise by employers or clients.

10. REFERENCES

- [1] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 337–346.
- [2] R. Victor, "Iterative and incremental development: A brief history," *IEEE Comput. Soc.*, pp. 47–56, 2003.
- [3] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
- [4] D. McCracken, *Digital Computer Programming*. Wiley, 1957.
- [5] B. Hanks, C. Wellington, T. Reichlmayr, and C. Coupal, "Integrating Agility in the Computer Science Curriculum," in *Proceedings of the 39th SIGCSE technical symposium on Computer science education - SIGCSE '08*, 2008, p. 19.
- [6] D. Hansson, "TDD is dead. Long live testing.," 2014. [Online]. Available: <http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html>. [Accessed: 03-Jan-2017].
- [7] M. Fowler, K. Beck, and D. Hansson, "Is Test-Driven Development Dead?," 2014. [Online]. Available: <http://martinfowler.com/articles/is-tdd-dead/>. [Accessed: 03-Jan-2017].
- [8] R. Cooper, *New Products: The Key Factors in Success*. Butterworth-Heinemann, 1990.
- [9] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. Boston, 2005.
- [10] L. Westfall, *The Certified Software Quality Engineer Handbook*. ASQ Quality Press, 2008.
- [11] G. J. Myers, *The Art of Software Testing*. John Wiley and Sons, 1979.

- [12] D. Gelperin and B. Hetzel, "The growth of software testing," *Commun. ACM*, vol. 31, no. 6, pp. 687–695, Jun. 1988.
- [13] R. E. Fairley, "Static Analysis and Dynamic Testing of Computer Software," *Computer (Long. Beach. Calif.)*, vol. 11, no. 4, pp. 14–23, Apr. 1978.
- [14] R. V. Binder, *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [15] Pathfinder Solutions, "Effective TDD for Complex Embedded Systems," 2012.
- [16] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *25th International Conference on Software Engineering, 2003. Proceedings.*, 2003, pp. 564–569.
- [17] M. Pancur, M. Ciglaric, M. Trampus, and T. Vidmar, "Towards empirical evaluation of test-driven development in a university environment," in *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, 2003, vol. 2, pp. 83–86.
- [18] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 226–237, Mar. 2005.
- [19] D. Janzen and H. Saiedian, "Does Test-Driven Development Really Improve Software Design Quality?," *IEEE Softw.*, vol. 25, no. 2, pp. 77–84, Mar. 2008.
- [20] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley, 2010.
- [21] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*. Pearson Education, 2009.
- [22] K. Beck, *Test-Driven Development: By Example*, 1st ed. Addison-Wesley, 2002.
- [23] H. Samimi, R. Hicks, A. Fogel, and T. Millstein, "Declarative mocking," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, 2013, p. 246.

- [24] G. Duggal and B. Suri, "Understanding regression testing techniques," in *Proceedings of 2nd National Conference on Challenges and Opportunities in Information Technology*, 2008.
- [25] H. Cibulski and A. Yehudai, "Regression Test Selection Techniques for Test-Driven Development," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 115–124.
- [26] R. V. Hooda, "An Automation of Software Testing: A Foundation for the Future," *Int. J. Latest Res. Sci. Technol. ISSN*, pp. 2278–5299, 2012.
- [27] B. Vodde and L. Koskela, "Learning Test-Driven Development by Counting Lines," *IEEE Softw.*, vol. 24, no. 3, pp. 74–79, May 2007.
- [28] T. K. Cheng, M. Jayasuriya, and J. Lim, "Removing the fear factor in Programming," *Python Pap. Monogr.*, vol. 2, pp. 1–9, 2010.
- [29] K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Rep.*, vol. 3, no. 7, pp. 37–50, 1998.
- [30] D. S. Janzen and H. Saiedian, "Test-driven development: Concepts, taxonomy, and future direction," *Computer (Long. Beach. Calif.)*, vol. 38, no. 9, pp. 43–50, 2005.
- [31] V. Massol and T. Husted, *JUnit in action*. Manning Publications Co., 2003.
- [32] K. Beck, "Aim, fire: test-first coding," *IEEE Softw.*, vol. 18, no. 5, pp. 87–89, 2001.
- [33] Agile Alliance, "Test-Driven Development Glossary." [Online]. Available: <https://www.agilealliance.org/glossary/tdd/>. [Accessed: 11-Dec-2016].
- [34] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.
- [35] G. Elliott, *Global Business Information Technology: An Integrated Systems Approach*. Pearson Education, 2004.
- [36] S. Balaji and M. Murugaiyan, "Waterfall vs. V-Model vs. Agile: A comparative

- study on SDLC,” *Int. J. Inf.*, vol. 2, no. 1, pp. 26–29, 2012.
- [37] B. W. Boehm, *Software Engineering Economics*. Prentice-Hall, 1981.
 - [38] P. A. Laplante and C. J. Neill, “The Demise of the Waterfall Model Is Imminent,” *Queue*, vol. 1, no. 10, pp. 10–15, Feb. 2004.
 - [39] S. Atkinson, “Why the Traditional Contract for Software Development is Flawed,” 2010.
 - [40] W. W. Royce, “Managing the development of large software systems,” in *proceedings of IEEE WESCON*, 1970, vol. 26, no. 8, pp. 328–338.
 - [41] K. Beck *et al.*, “Principles behind the Agile Manifesto,” 2001. [Online]. Available: <http://agilemanifesto.org/principles.html>. [Accessed: 07-Nov-2016].
 - [42] A. Moran, “Agile Risk Management,” in *Agile Risk Management*, Cham: Springer International Publishing, 2014, pp. 33–60.
 - [43] S. Rakitin, “Manifesto Elicits Cynicism,” *IEEE Comput. Soc.*, vol. 34, no. 12, pp. 4–7, 2001.
 - [44] S. Ambler, “Agile/Lean Documentation: Strategies for Agile Software Development,” 2001.
 - [45] Version One, “10th Annual State of Agile Report,” 2015.
 - [46] A. Sidky, J. Arthur, and S. Bohner, “A disciplined approach to adopting agile practices: the agile adoption framework,” *Innov. Syst. Softw. Eng.*, vol. 3, no. 3, pp. 203–216, Sep. 2007.
 - [47] J. B. Arthur, “Effects of Human Resource Systems on Manufacturing Performance and Turnover,” *Acad. Manag. J.*, vol. 37, no. 3, pp. 670–687, Jun. 1994.
 - [48] M. M. Müller and F. Padberg, “About the return on investment of test-driven development,” in *Edser-5 5 th international workshop on economic-driven software engineering research*, 2003, p. 26.

- [49] A. Marchenko, P. Abrahamsson, and T. Ihme, "Long-Term Effects of Test-Driven Development A Case Study," in *Agile Processes in Software Engineering and Extreme Programming: 10th International Conference, XP 2009, Pula, Sardinia, Italy, May 25-29, 2009. Proceedings*, P. Abrahamsson, M. Marchesi, and F. Maurer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 13–22.
- [50] J. Sanchez, L. Williams, and E. M. Maximilien, "A Longitudinal Study of the Use of a Test-Driven Development Practice in Industry," in *Agile2007 Conference*, 2007.
- [51] M. Saunders, P. Lewis, and A. Thornhill, "Research Methods for Business Students," *Harlow Prentice Hall*, 2009.
- [52] B. Slife and R. Williams, *What's Behind the Research?: Discovering Hidden Assumptions in the Behavioral Sciences*. SAGE Publications, 1995.
- [53] R. Davison, "An Action Research Perspective of Group Support Systems: How to Improve Meetings in Hong Kong," City University of Hong Kong, 1998.
- [54] R. D. Galliers and F. F. L., "Choosing Appropriate Information Systems Research Approaches: A Revised Taxonomy," in *In Proceedings of the IFIP TC8 WG8.2*, 1990, pp. 317–335.
- [55] W. Levin, *Sociological Ideas: Concepts and Applications*. Wadsworth Pub. Co., 1987.
- [56] G. J. De Vreede, *Facilitating Organizational Change: The participative application of dynamic modelling*. TU Delft, Delft University of Technology, 1995.
- [57] W. J. Orlikowski and J. J. Baroudi, "Studying Information Technology in Organizations: Research Approaches and Assumptions," *Inf. Syst. Res.*, vol. 2, no. 1, pp. 1–28, 1991.
- [58] B. Kaplan and D. Duchon, "Combining Qualitative and Quantitative Methods in Information Systems Research: A Case Study," *MIS Q.*, vol. 12, no. 4, pp.

571–586, 1988.

- [59] K. F. Hyde, “Recognising deductive processes in qualitative research,” *Qual. Mark. Res. An Int. J.*, vol. 3, no. 2, pp. 82–90, Jun. 2000.
- [60] G. D. Ruxton and M. Neuhäuser, “When should we use one-tailed hypothesis testing?,” *Methods Ecol. Evol.*, vol. 1, no. 2, pp. 114–117, 2010.
- [61] C. W. J. Granger, “Some Recent Development in a Concept of Causality,” *J. Econom.*, vol. 39, no. 1–2, pp. 199–211, Sep. 1988.
- [62] J. Amaral, “About Computing Science Research Methodology,” 2006.
- [63] L. Slevitch, “Qualitative and Quantitative Methodologies Compared: Ontological and Epistemological Perspectives,” *J. Qual. Assur. Hosp. Tour.*, vol. 12, no. 1, pp. 73–81, Feb. 2011.
- [64] I. Newman and C. Benz, *Qualitative-quantitative Research Methodology: Exploring the Interactive Continuum*. Southern Illinois University Press, 1998.
- [65] J. W. Creswell, *Educational Research: Planning, Conducting, and Evaluating Quantitative and Qualitative Research*. Pearson Education, Limited, 2015.
- [66] D. F. Polit and C. T. Beck, “Generalization in Quantitative and Qualitative Research: Myths and Strategies,” *Int. J. Nurs. Stud.*, vol. 47, no. 11, pp. 1451–1458, Nov. 2010.
- [67] A. Rindfleisch, A. J. Malter, S. Ganesan, and C. Moorman, “Cross-Sectional Versus Longitudinal Survey Research: Concepts, Findings, and Guidelines,” *J. Mark. Res.*, vol. 45, no. 3, pp. 261–279, Jun. 2008.
- [68] M.-H. Chen, M. R. Lyu, and W. E. Wong, “Effect of code coverage on software reliability measurement,” *IEEE Trans. Reliab.*, vol. 50, no. 2, pp. 165–170, Jun. 2001.
- [69] A. Glover, “In pursuit of code quality: Don’t be fooled by the coverage report,” 2006.
- [70] F. E. Norman, *Software Metrics: A Rigorous Approach*, Chapman & Hall,

1991.

- [71] N. F. Schneidewind and H.-M. Hoffmann, "An Experiment in Software Error Data Collection and Analysis," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 3, pp. 276–286, May 1979.
- [72] IEEE Computer Society, "IEEE Standard Dictionary of Measures to Produce Reliable Software," 1989.
- [73] M. Paulk, "Capability Maturity Model for Software," in *Encyclopedia of Software Engineering*, Hoboken, NJ, USA: John Wiley & Sons, Inc., 2002.
- [74] W. A. Florac, "Software quality measurement: A framework for counting problems and defects," 1992.
- [75] E. Rashid, "R4 Model for Case-Based Reasoning and Its Application for Software Fault Prediction," *Int. J. Softw. Sci. Comput. Intell.*, vol. 8, no. 3, pp. 19–38, Jul. 2016.
- [76] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1986.
- [77] C. Jones, "Critical Problems in Software Measurement," *Burlington, Mass. Softw. Product. Res.*, pp. 72–77, 1992.
- [78] L. Gulecha, "Software Testing Metrics," 2015.
- [79] A. J. Albrecht and J. E. Gaffney, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Trans. Softw. Eng.*, vol. SE-9, no. 6, pp. 639–648, Nov. 1983.
- [80] P. Weissgerber and S. Diehl, "Identifying Refactorings from Source-Code Changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 231–240.
- [81] P. Bourque and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (Swebok(r)): Version 3.0*. IEEE Computer Society Press, 2014.

- [82] F. Beck and S. Diehl, "On the congruence of modularity and code coupling," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, 2011, p. 354.
- [83] R. Gunstone, "Experiments," in *Encyclopedia of Science Education*, Springer Netherlands, 2015, pp. 418–419.
- [84] G. R. Bergersen and D. I. K. Sjøberg, "Evaluating methods and technologies in software engineering with respect to developers' skill level," in *16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)*, 2012, pp. 101–110.
- [85] A. Čaušević, D. Sundmark, and S. Punnekkat, "Impact of Test Design Technique Knowledge on Test Driven Development: A Controlled Experiment," in *Agile Processes in Software Engineering and Extreme Programming: 13th International Conference, XP 2012, Malmö, Sweden, May 21-25, 2012. Proceedings*, C. Wohlin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 138–152.
- [86] A. Čaušević, R. Shukla, and S. Punnekkat, "Industrial study on test driven development: challenges and experience," in *Proceedings of the 1st International Workshop on Conducting Empirical Studies in Industry*, 2013, pp. 15–20.
- [87] M. Pančur and M. Ciglarič, "Impact of test-driven development on productivity, code and tests: A controlled experiment," *Inf. Softw. Technol.*, vol. 53, no. 6, pp. 557–573, Jun. 2011.
- [88] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development," in *Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06*, 2006, p. 356.
- [89] T. M. Foster, K. Jarema, and A. Poling, "Inferential Statistics: Criticised by Sidman (1960), but Popular in the Journal of the Experimental Analysis of

- Behavior,” *Behav. Chang.*, vol. 16, no. 3, pp. 203–204, Sep. 1999.
- [90] D. H. Barlow and S. C. Hayes, “Alternating treatments design: one strategy for comparing the effects of two treatments in a single subject.,” *J. Appl. Behav. Anal.*, vol. 12, no. 2, 1979.
 - [91] H. M. Collins, “The Meaning of Experiment: Replication and Reasonableness,” in *Dismantling Truth*, H. Lawson and L. Appignanesi, Eds. Weidenfeld, 1989, pp. 82–92.
 - [92] B. Skvorc, “The Best PHP Framework for 2015: Survey Results,” 2015.
 - [93] S. P. Ong *et al.*, “The Materials Application Programming Interface (API): A simple, flexible and efficient API for materials data based on REpresentational State Transfer (REST) principles,” *Comput. Mater. Sci.*, vol. 97, pp. 209–215, Feb. 2015.
 - [94] R. V Krejcie and D. W. Morgan, “Determining Sample Size for Research Activities,” *Educ. Psychol. Meas.*, vol. 30, no. 3, pp. 607–610, 1970.
 - [95] S. Goldthwaite and W. Graylin, “System and method for mobile payment and fulfillment of digital goods,” 2004.
 - [96] A. Griffin, “Metrics for measuring product development cycle time,” *J. Prod. Innov. Manag.*, vol. 10, no. 2, pp. 112–125, Mar. 1993.
 - [97] R. E. Nisbett, H. Zukier, and R. E. Lemley, “The dilution effect: Nondiagnostic information weakens the implications of diagnostic information,” *Cogn. Psychol.*, vol. 13, no. 2, pp. 248–277, Apr. 1981.
 - [98] J. Grigg, “Arrange Act Assert,” 2012.
 - [99] M. Jamro and B. Trybus, “Testing Procedure for IEC 61131-3 Control Software,” *IFAC Proc. Vol.*, vol. 46, no. 28, pp. 192–197, 2013.
 - [100] D. Chelimsky, *The RSpec Book: Behaviour-driven Development with RSpec, Cucumber, and Friends*. Pragmatic Bookshelf, 2010.
 - [101] M. Tufano *et al.*, “When and Why Your Code Starts to Smell Bad,” in 2015

IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 403–414.

- [102] D. Goldberg, “What Every Computer Scientist Should Know About Floating-point Arithmetic,” *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991.
- [103] E. H. McKinney, “Generalized Birthday Problem,” *Am. Math. Mon.*, vol. 73, no. 4, pp. 385–387, 1966.
- [104] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: a dynamic data race detector for multithreaded programs,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.
- [105] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Pearson Education, 2007.
- [106] Y.-I. Chou, “Back Matter: Statistical Analysis,” *Int. Stat. Rev.*, vol. 43, no. 2, 1975.
- [107] V. E. Faherty, *Compassionate Statistics: Applied Quantitative Analysis for Social Services (With Exercises and Instructions in SPSS)*. SAGE Publications, 2008.
- [108] M. Krzywinski and N. Altman, “Points of significance: Significance, P values and t-tests,” *Nat Meth*, vol. 10, no. 11, pp. 1041–1042, Nov. 2013.
- [109] J. Neyman and E. S. Pearson, “The testing of statistical hypotheses in relation to probabilities a priori,” *Math. Proc. Cambridge Philos. Soc.*, vol. 29, no. 4, pp. 492–510, 1933.
- [110] P. Piwowarski, M. Ohba, and J. Caruso, “Coverage Measurement Experience During Function Test,” in *Proceedings of the 15th International Conference on Software Engineering*, 1993, pp. 287–301.
- [111] Q. Yang, J. J. Li, and D. M. Weiss, “A Survey of Coverage-Based Testing Tools,” *Comput. J.*, vol. 52, no. 5, pp. 589–597, Aug. 2009.
- [112] B. George and L. Williams, “A structured experiment of test-driven development,” *Inf. Softw. Technol.*, vol. 46, no. 5, pp. 337–342, Apr. 2004.

- [113] R. Johnson, “J2EE development frameworks,” *Computer (Long. Beach. Calif.)*., vol. 38, no. 1, pp. 107–110, Jan. 2005.

APPENDICES

A. Further Implementation Details

A.1. List of Test Cases

Requirement ID	Level	Test Case Description
TDD01	Feature	User can view published event listing
TDD01	Feature	User cannot view unpublished event listing
TDD01	Unit	Can get formatted date
TDD01	Unit	Can get formatted time
TDD01	Unit	Can get ticket price in pounds
TDD01	Unit	Events with published at date are published
TTL01	Feature	User can purchase published event tickets
TTL01	Unit	Charges with a valid token are successful
TDD02	Feature	Email is required to purchase tickets
TDD02	Feature	Email must be valid to purchase tickets
TDD02	Feature	Ticket quantity is required to purchase tickets
TDD02	Feature	Ticket quantity must be at least one to buy tickets
TDD02	Feature	Payment token is required
TDD02	Feature	Cannot purchase tickets to an unpublished event
TDD02	Feature	Order not created if payment fails
TDD02	Unit	Charges with an invalid payment token fail
TTL02	Feature	User cannot purchase more tickets than remain
TTL02	Unit	Can add tickets
TTL02	Unit	Remaining tickets do not include tickets associated with order
TTL02	Unit	Trying to reserve more tickets than remain throws exception
TDD03	Unit	A ticket can be released
TTL03	Unit	Converting to an array
TDD04	Unit	Creating an order from tickets and a charge
TDD04	Unit	Calculating the total cost
TTL04	Feature	Cannot purchase tickets another user is trying to purchase
TTL04	Unit	Can reserve available tickets
TTL04	Unit	Running a hook before the first charge
TTL04	Unit	A ticket can be reserved
TDD05	Unit	Retrieving the reservation tickets
TDD05	Unit	Retrieving the user email
TDD05	Unit	Reserved tickets are released when a reservation is cancelled
TDD05	Unit	Completing a reservation
TDD06	Unit	Can get charges during a callback
TTL06	Feature	User can view their order confirmation
TTL06	Unit	Retrieving an order by confirmation number
TTL06	Unit	Retrieving a nonexistent order by confirmation number throws exception
TDD07	Unit	Must be twenty four characters long
TDD07	Unit	Can only contain uppercase letters and numbers
TDD07	Unit	Cannot contain ambiguous characters
TDD07	Unit	Confirmation numbers must be unique
TTL07	Unit	Can get details about a successful charge
TDD08	Unit	Ticket codes are at least six characters long
TDD08	Unit	Ticket codes can only contain uppercase letters
TDD08	Unit	Ticket codes for the same ticket ID are the same
TDD08	Unit	Ticket codes for different ticket ID's are different
TDD08	Unit	Ticket codes generated with different salts are different
TDD08	Unit	Ticket can be claimed for an order

A.2. Managing the Database

The application requires some data persistence, which in the case of this research is handled by a database. To create the schema, Laravel supports the use of migrations and they have been used extensively because it serves as a form of version control. Each change to the schema requires a new migration, thus allowing them to be rolled back in the event that the database needs to be returned to a previous state. Perhaps the biggest benefit of using migrations, however, is that they can be turned on or off during testing. This is advantageous when executing tests that do not require any interaction with the database because it causes a significant increase in performance.

Nonetheless, some tests do require the database and often require that there is some data available. Obviously it is inadvisable to have real data stored in the database during testing, so Laravel offers another solution: seeding. Only one seeder class was required in the case of this research, and it is responsible for entering dummy data in to the database for testing purposes. More specifically, it adds one event and 20 associated tickets. The main advantage of this approach is that it saves a considerable amount of time, preventing the need to enter data manually each time a migration is run, and in much the same way as migrations, the seeder(s) can be turned off for better test suite performance.

A.3. Use of Façades

The software artefact makes use of two façade classes, one as part of the ticket code generation implementation and the other as part of the order number generation implementation. In both cases the façade actually provides an accessor to an interface, rather than the concrete generator object, which is then bound to a concrete generator in the application service provider. This is for the purposes of flexibility and extensibility, allowing additional concrete generator objects to be interchanged by reconfiguring one line of code in the service provider (as opposed to everywhere the object appears in the codebase).

B. Experiment Development Log

Session ID	Session Date	Start Time	End Time	Total Time	Requirement ID	Tests	LOC (Tests)	LOC (Application)	Defects	Refactors	Dependencies
SES01	19/03/17	20:58	23:41	163	TDD01	6	44	58	0	8	2
SES02	25/03/17	10:46	01:37	171	TTL01	3	20	82	2	2	2
SES03	25/03/17	17:28	19:19	111	TDD02	8	53	24	0	7	1
SES04	25/03/17	20:45	22:21	96	TTL02	5	42	14	2	1	2
SES05	25/03/17	23:47	00:49	62	TDD03	2	18	12	0	5	0
SES06	26/03/17	14:42	15:12	30	TTL03	1	9	16	1	0	0
SES07	26/03/17	16:21	17:45	84	TDD04	0	0	24	0	5	1
SES08	26/03/17	20:37	21:31	54	TDD04	2	17	15	2	5	2
SES09	26/03/17	22:10	23:49	99	TTL04	4	19	47	5	1	1
SES10	30/03/17	19:33	20:28	55	TDD05	2	23	8	0	3	0
SES11	30/03/17	21:54	23:22	88	TTL05	3	10	44	2	4	2
SES12	31/03/17	22:35	00:07	92	TDD06	4	27	9	1	5	1
SES13	01/04/17	13:12	14:40	88	TTL06	3	19	39	1	1	1
SES14	02/04/17	14:27	15:56	89	TDD07	4	37	12	0	2	2
SES15	02/04/17	18:29	19:46	77	TTL07	1	8	32	2	2	0
SES16	02/04/17	21:05	23:38	153	TDD08	5	43	19	2	5	2
SES17	03/04/17	12:34	13:11	37	TTL08	1	3	10	0	0	0

C. Defect Log

Defect ID	Date	Time	Requirement ID	Finding Mode	Criticality	Urgency	Defect Type	Location	Uniqueness	Changes to
DEF01	25/03/17	12:04	TTL01	Dynamic	2	1	Test Case	PurchaseTicketsTest	Original	PurchaseTicketsTest
DEF02	25/03/17	01:21	TTL01	Dynamic	5	3	Test Case	EventOrdersController	Original	EventOrdersController
DEF03	25/03/17	21:03	TTL02	Static	1	1	Code inspection	Event	Original	Event, EventTest
DEF04	25/03/17	21:59	TTL02	Static	2	2	Code inspection	Event	Duplicate (DEF03)	Event, EventTest
DEF05	26/03/17	15:08	TTL03	Static	3	4	Code inspection	Order	Original	Order, EventOrdersController
DEF06	26/03/17	21:03	TDD04	Dynamic	3	2	Test case	EventOrdersController	Original	EventOrdersController
DEF07	26/03/17	21:04	TDD04	Dynamic	3	2	Test case	EventOrdersController	Duplicate (DEF06)	EventOrdersController
DEF08	26/03/17	22:13	TTL04	Static	1	1	Code inspection	EventOrdersController	Original	EventOrdersController
DEF09	26/03/17	22:44	TTL04	Dynamic	4	5	Test case	PurchaseTicketsTest	Original	PurchaseTicketsTest
DEF10	26/03/17	22:56	TTL04	Dynamic	3	3	Test case	EventOrdersController	Original	Reservation
DEF11	26/03/17	23:42	TTL04	Dynamic	3	5	Test case	PurchaseTicketsTest	Original	PurchaseTicketsTest
DEF12	26/03/17	23:47	TTL04	Static	1	1	Code inspection	Reservation	Original	Reservation
DEF13	30/03/17	22:45	TTL05	Static	5	4	Code inspection	EventOrdersController	Original	Order
DEF14	30/03/17	23:02	TTL05	Static	2	3	Code inspection	EventOrdersController	Original	Order, Reservation
DEF15	31/03/17	23:36	TDD06	Dynamic	1	2	Test Case	FakePaymentGateway	Original	FakePaymentGateway
DEF16	01/04/17	14:35	TTL06	Dynamic	4	5	Test Case	OrderController	Original	OrdersController
DEF17	02/04/17	19:36	TTL07	Dynamic	3	2	Test Case	Reservation	Original	Reservation
DEF18	02/04/17	19:38	TTL07	Dynamic	3	2	Test Case	Reservation	Duplicate(DEF17)	Reservation
DEF19	02/04/17	22:01	TDD08	Static	4	5	Code inspection	Ticket	Original	Ticket
DEF20	02/04/17	22:52	TDD08	Static	2	5	Code inspection	Ticket	Original	Ticket, TicketTest

D. Refactoring Log

Refactor ID	Date	Time	Requirement ID	Type	Change Made
REF01	19/03/17	22:31	TDD01	Structural	Move method
REF02	19/03/17	22:49	TDD01	Structural	Move method
REF03	19/03/17	22:52	TDD01	Structural	Move method
REF04	19/03/17	22:59	TDD01	Local	Rename method
REF05	19/03/17	23:14	TDD01	Local	Rename method
REF06	19/03/17	23:19	TDD01	Local	Rename method
REF07	19/03/17	23:32	TDD01	Local	Add paramter
REF08	19/03/17	23:39	TDD01	Local	Add paramter
REF09	25/03/17	01:29	TTL01	Local	Add parameter
REF10	25/03/17	01:20	TTL01	Structural	Move method
REF11	25/03/17	18:11	TDD02	Structural	Move method
REF12	25/03/17	18:12	TDD02	Structural	Move method
REF13	25/03/17	18:12	TDD02	Structural	Move method
REF14	25/03/17	18:14	TDD02	Structural	Move method
REF15	25/03/17	18:16	TDD02	Structural	Move method
REF16	25/03/17	18:22	TDD02	Local	Rename method
REF17	25/03/17	19:17	TDD02	Local	Rename method
REF18	25/03/17	22:08	TTL02	Structural	Move method
REF19	26/03/17	00:08	TDD03	Local	Rename method
REF20	26/03/17	00:18	TDD03	Structural	Move method
REF21	26/03/17	00:25	TDD03	Structural	Move method
REF22	26/03/17	00:33	TDD03	Structural	Move method
REF23	26/03/17	00:37	TDD03	Structural	Move method
REF24	26/03/17	16:26	TDD04	Local	Add parameter
REF25	26/03/17	17:03	TDD04	Structural	Move method
REF26	26/03/17	17:12	TDD04	Local	Hide method
REF27	26/03/17	17:17	TDD04	Local	Rename method
REF28	26/03/17	17:21	TDD04	Local	Remove parameter
REF29	26/03/17	20:46	TDD04	Structural	Move method
REF30	26/03/17	20:53	TDD04	Local	Rename method
REF31	26/03/17	21:00	TDD04	Local	Add parameter
REF32	26/03/17	21:21	TDD04	Structural	Move method
REF33	26/03/17	21:22	TDD04	Structural	Move method
REF34	26/03/17	23:04	TTL04	Local	Rename method
REF35	30/03/17	20:13	TDD05	Local	Rename method
REF36	30/03/17	20:21	TDD05	Local	Hide method
REF37	30/03/17	20:24	TDD05	Local	Hide method
REF38	30/03/17	21:59	TTL05	Local	Add parameter
REF39	30/03/17	22:01	TTL05	Structural	Move method
REF40	30/03/17	22:23	TTL05	Local	Add parameter
REF41	30/03/17	22:58	TTL05	Structural	Move method
REF42	31/03/17	22:49	TDD06	Structural	Move method
REF43	31/03/17	22:59	TDD06	Local	Reveal method
REF44	31/03/17	23:04	TDD06	Local	Reveal method
REF45	31/03/17	23:26	TDD06	Structural	Move method
REF46	01/04/17	00:03	TDD06	Structural	Move method
REF47	01/04/17	14:00	TTL06	Structural	Move method
REF48	02/04/17	15:29	TDD07	Structural	Move method
REF49	02/04/17	15:45	TDD07	Structural	Move methos
REF50	02/04/17	19:11	TTL07	Local	Add parameter
REF51	02/03/17	19:18	TTL07	Local	Rename method
REF52	02/03/17	21:22	TDD08	Local	Add paramter
REF53	02/03/17	21:38	TDD08	Structural	Move method
REF54	02/03/17	22:25	TDD08	Local	Rename method
REF55	02/03/17	22:27	TDD08	Local	Add parameter
REF56	02/03/17	22:49	TDD08	Local	Add paramter

E. Dependency Log

Dependency ID	Date	Time	Requirement ID	Dependent Class	Dependency Class
DEP01	19/03/17	22:47	TDD01	ModelFactory	Carbon
DEP02	19/03/17	23:02	TDD01	EventsController	Event
DEP03	25/03/17	11:35	TTL01	EventOrdersController	IPaymentGateway
DEP04	25/03/17	11:38	TTL01	EventOrdersController	Event
DEP05	25/03/17	18:54	TDD02	EventOrdersController	PaymentFailedException
DEP06	25/03/17	21:51	TTL02	Event	NotEnoughTicketsException
DEP07	25/03/17	22:13	TTL02	EventOrdersController	NotEnoughTicketsException
DEP08	26/03/17	17:24	TDD04	Event	Order
DEP09	26/03/17	20:49	TDD04	EventOrdersController	Order
DEP10	26/03/17	21:19	TDD04	EventOrdersController	Reservation
DEP11	26/03/17	22:51	TTL04	Ticket	Carbon
DEP12	30/03/17	22:06	TTL05	Event	Reservation
DEP13	30/03/17	23:16	TTL05	Reservation	Order
DEP14	31/03/17	23:19	TDD06	DatabaseSeeder	Carbon
DEP15	01/04/17	13:19	TTL06	OrdersController	Order
DEP16	02/04/17	14:42	TDD07	OrderNumber	IOrderNumberGenerator
DEP17	02/04/17	14:51	TDD07	Order	OrderNumber
DEP18	02/04/17	21:58	TDD08	Ticket	TicketCode
DEP19	02/04/17	22:48	TDD08	HashidsTicketCodeGenerator	Hashids