Test-Driven Development for
Aspect-Oriented Programming

A dissertation submitted to The University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences.

2014

ANDREAS VLACHOU

SCHOOL OF COMPUTER SCIENCE
# Table of Contents

List of Figures ............................................................................................................. 6  
List of Tables .............................................................................................................. 6  
Abstract ....................................................................................................................... 7  
Declaration .................................................................................................................. 8  
Intellectual Property Statement ............................................................................... 9  
Acknowledgements .................................................................................................... 10  
Chapter 1: Introduction .............................................................................................. 11  
  1.1 Overview ............................................................................................................. 11  
  1.2 Aim and Objectives ............................................................................................ 12  
  1.3 Dissertation Outline .......................................................................................... 13  
Chapter 2: Background and Literature Review ....................................................... 14  
  2.1 Overview ............................................................................................................. 14  
  2.2 Test-Driven Development ................................................................................... 14  
      Overview ............................................................................................................. 14  
      Concepts ........................................................................................................... 14  
      Best Practices .................................................................................................. 16  
      Benefits ............................................................................................................ 17  
      Issues ................................................................................................................ 17  
      Summary .......................................................................................................... 18  
  2.3 Aspect-Oriented Programming ........................................................................... 19  
      Overview .......................................................................................................... 19  
      Terminology ...................................................................................................... 20  
      AspectJ ............................................................................................................. 20  
      Benefits ............................................................................................................ 22  
      Issues ................................................................................................................ 23  
      Summary .......................................................................................................... 24  
  2.4 Testing techniques for Aspects ......................................................................... 25  
      Overview .......................................................................................................... 25  
      Test Patterns .................................................................................................... 25  
      AdviceTracer Tool ............................................................................................. 27
Design ........................................................................................................ 48
Implementation .......................................................................................... 48
Testing .......................................................................................................... 49

5.6      Transaction Security ......................................................................... 51
Red Stage .................................................................................................... 51
Green Stage ................................................................................................ 52
Refactor Stage ............................................................................................ 53

5.7      Chapter Conclusion .......................................................................... 53

Chapter 6: Evaluation .................................................................................. 54

6.1      Overview .......................................................................................... 54
6.2      Evaluation Approach ........................................................................ 54
6.3      Reverse-engineering Phase ............................................................... 55
Overview .................................................................................................... 55
Pointcut Testing ......................................................................................... 55
Advice Testing ............................................................................................ 56
Conclusion .................................................................................................. 57

6.4      TDD Evaluation ................................................................................ 58
Overview .................................................................................................... 58
TDD Cycle .................................................................................................. 58
TDD Practices ............................................................................................. 58
Implications for TDD .................................................................................. 60
Conclusion .................................................................................................. 60

6.5      Project Result ................................................................................... 61
Proposed TDD Cycle .................................................................................. 61

6.6      Chapter Conclusion .......................................................................... 62

Chapter 7: Conclusion .................................................................................. 63

7.1      Project Summary .............................................................................. 63
7.2      Reflection ........................................................................................ 63
7.3      Future Work ..................................................................................... 64

Bibliography ............................................................................................... 65
List of Figures

Figure 2.1 - TDD Motto ................................................................. 15
Figure 2.2 - TDD Steps ................................................................. 16
Figure 2.3 Delegated Advice Example ........................................ 26
Figure 2.4 Aspect Flag Example .................................................. 26
Figure 4.1 System Architecture.................................................... 34
Figure 4.2 Server Information View ............................................. 37
Figure 4.3 Accounts View ............................................................ 38
Figure 4.4 Adding Bank Account ................................................ 38
Figure 4.5 Account View ............................................................. 38
Figure 4.6 Transfer View ............................................................. 39
Figure 5.1 System Architecture with Aspects ................................. 43
Figure 5.2 Deposit Constraint Example ....................................... 45
Figure 5.3 Withdrawal Constraint Example .................................. 45
Figure 5.4 Standard Output .......................................................... 47
Figure 5.5 Sample Account Log .................................................. 50
Figure 5.6 Sample Transaction Log ............................................. 50
Figure 5.7 Logging Test Cases ....................................................... 50
Figure 6.1 Changes to TDD Steps ............................................... 59
Figure 6.2 Proposed TDD Steps for creating Aspects ...................... 61

List of Tables

Table 1 System Elements ............................................................ 33
Table 2 Cross-Cutting Concerns ................................................ 40
Abstract

Test-Driven Development (TDD) is an agile, software engineering practice which breaks from the traditional development cycle. Testing is no longer used only for asserting the behaviour of a system but is seen also as a design technique that drives the entire implementation process. While TDD fits well in the object-oriented world, as objects form excellent candidates for unit tests, its supportability for the emerging modularisation techniques is not so clear. Aspect-Oriented Programming (AOP) is one such technique which deals with the separation of cross-cutting concerns in software systems, through the use of stand-alone modules called aspects.

It is imperative for Aspect-Oriented Programming to integrate with Test-Driven Development in order to facilitate its use in real world software projects. The same applies to Test-Driven Development which has to adapt its principles to cope with the specific concepts of Aspect-Oriented Programming.

The project examined the integration of the aforementioned technologies. An AOP banking system was built, in which various testing techniques were employed to test its cross-cutting concerns. The Reverse-Engineering methodology was then applied to the final application to detect whether those techniques could have been utilised in a test-driven way.

The analysis and evaluation of the findings proved that it is feasible to use Test-Driven Development in an Aspect-Oriented Programming environment by modifying its development cycle, despite any complexity issues that might arise.
Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Intellectual Property Statement

i. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the dissertation, for example graphs and tables (“Reproductions”), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/display.aspx?DocID=487), in any relevant Dissertation restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations ) and in The University’s Guidance for the Presentation of Dissertations.
Acknowledgements

I would like to thank my supervisor, Dr John Sargeant, for his excellent supervision and help throughout the project. I am grateful for the guidance he has provided, during both my MSc and BSc dissertations.

I would also like to thank my family for their unconditional support and for providing the funding to complete my MSc. Last but not least, I thank my friends and colleagues for their friendship and motivation over the past year.

I dedicate this work to my Grandfather Andreas and my Aunt Lela, who would have been so proud of what I have achieved. I know you still watch over me.
Chapter 1: Introduction

1.1 Overview

In recent years, the IT community has seen an increased interest in two software development concepts. Test-Driven Development and Aspect-Oriented Programming can be considered as offspring of Object-Oriented Programming, with both focusing on two different areas of software engineering i.e. design and implementation.

TDD is a software development life-cycle that is fundamental part of many Agile methodologies. It is an approach that goes beyond traditional software development, rearranging the way you program. Its principles promote Test-First Programming, where tests are written before the implementation rather than after and most importantly, they have an impact on the actual implementation itself.

Aspect-Oriented Programming is a programming paradigm [1] which aims to increase the expressiveness of Object-Oriented Design. While OOP manages to represent the domain model of an application using high level analysis and design, it has limited supportability for secondary concerns like login or transaction handling. This limitation gave rise to Post-Object Programming mechanisms. AOP is an example of POP, founded on the idea of modularization, which promotes the separation of concerns in stand-alone and interchangeable modules. These modules are called Aspects in AOP. Aspects require the programmer to specify points in the execution of code, called “Pointcuts” and the code to be included, called “Advice”, when a Pointcut is reached.

While AOP might be a well-established and widely used technology, essentially, it will need to integrate with Test-Driven Development in order to cope with software projects that endorse test first programming. On its part, TDD may have to adapt its practices with the aim of facilitating the integration.

Initial background research revealed both the challenges behind the applicability of this adoption and also some possible changes to TDD that might overcome them. An interesting idea is delegating the behaviour of Aspects to separate classes and then unit test those classes. This project has used this suggestion, identified more possible ones
and applied them to determine if they caused any problems, through the built of a banking application.

This banking application is created with cross-cutting concerns in mind. For example, the issues of transaction handling, transaction synchronization and logging are resolved using an aspect-oriented approach while integrating TDD at the same time.

In brief, this project answers the research question of whether the combination of TDD with AOP is applicable and efficient enough for real world software systems.

1.2 Aim and Objectives

The aim of this project is to apply Test-Driven Development (TDD) practices to Aspect-Oriented Programming (AOP), detect any required changes to these practices and classify any issues found during the implementation of these changes.

Objectives:

- Review the literature to gain an understanding on TDD practices and identify the key elements in AOP.
- Design and implement a non-trivial, banking application using methodologies and tools from both TDD and AOP. The application will be a banking system and will be built on the concerns of security, transaction management, logging services and synchronization.
- Identify difficulties and problems arising from the combination of the two technologies, based on the aforementioned application and suggest ways of refactoring TDD practises so that they better support AOP.
- Determine whether the proposed changes lead to any issues or restrict on any level the benefits obtained by each technology separately.
1.3 Dissertation Outline

The report is structured as follows:

Chapter 2: Background and Literature Review
This chapter serves as the literature review of the concepts of TDD and AOP. It describes in detail their best practices and discusses how each one works.

Chapter 3: Problem Analysis
This chapter examines the problem proposed by the project and identifies its challenges. It additionally presents the deliverables, the methodologies and testing techniques employed as part of the project.

Chapter 4: Banking System
This chapter provides an overview of the banking system implemented to support the integration of TDD and AOP.

Chapter 5: Cross-cutting Concerns
This section emphasises the development of cross-cutting concerns in the banking system. It gives an in-depth breakdown of concerns’ design, implementation and testing phases.

Chapter 6: Evaluation
This section analyses the results of the project in terms of each testing technique applied and answers the research question of the project.

Chapter 7: Conclusion
The final chapter summarises the work that has been conducted in respect to the project and states any future research.
Chapter 2: Background and Literature Review

2.1 Overview

This chapter aims to set the background context of the project based on the information found during the research phase. Detailed explanations of both TDD and AOP are provided, as well as the basic concepts, benefits and any related issues, for each technology. In addition, various techniques for testing aspects are presented with examples of their use.

2.2 Test-Driven Development

“If you can’t write tests for what you are about to code, then you shouldn’t even be thinking about coding.” [2]

Overview

Test-Driven Development was introduced by Kent Beck in 2003 in his book “Test Driven Development: By Example” [3]. It is a technique originated from the Test-First Programming (TFP) concept of Extreme Programming, also created by Kent Beck in the 90s.

Taking as basis that the programmer has a basic understanding of the code he is about to produce then it is a logical conclusion that he also knows how to test it. This gives rise to the main concept of TFP which is to ensure code testing. TDD takes this concept a step further. Testing is not only used to verify and validate code but it also drives the actual design and implementation of the code. [4]

This section will introduce the key concepts of Test-Driven Development, its benefits and also its limitations.

Concepts

Test-Driven Development is a combination of TFP and refactoring, following two simple rules:

1. Code is written only if a test has failed
2. Duplication is eliminated

It is done in short iterations, centred on automated unit tests. Each iteration, has three stages based on the motto “Red, Green, Refactor” [3], as seen in Figure 1.
In the “Red Stage”, tests are created that should initially fail or even not compile. This is sensible as code has not been written for the specific test and some objects or features might have not been implemented yet. Failure during this stage cannot be seen as a bad thing but rather as something necessary. It sets the starting point to implement the code that actually passes the test.

**Green Stage**
This stage takes as principle the first rule of TDD and endorses the use of practices like “Keep it Simple” (KISS) and "You Ain't Gonna Need It" (YAGNI). This results in writing as little code as possible in order to pass the test.

In traditional software development, the programmer tends to over-program. Extensibility for example prompts the addition of code that is not needed at the time. This is not the case in TDD. The programmer concentrates on the actual required functionality. The code might not be of great quality and some values might even be hardcoded but that is acceptable as the Refactor Stage will improve it. Once enough code has been produced for the test to pass, the developer can proceed to the next stage.

**Refactor Stage**
This stage is based on the second rule and it illustrates the importance of refactoring in TDD. By Refactoring, is meant the process of enhancing the design and quality of code without modifying its actual behaviour.

At this point, any duplicate code is removed, any hardcoded data is modified to match actual design and any bad coding practices used in the “Green” stage are cleaned up. Readability, performance and maintainability can be improved at this stage as long as
KISS and YAGNI practises still apply. At the end of the refactoring process, the existing test cases should still pass to ensure that the functionality has not been broken. If everything is correct then the programmer can continue to a new iteration based on another requirement/feature.

Figure 2.2 - TDD Steps

Best Practices
A set of best practices for applying Test-Driven Development were identified in accordance to [5], [6] and [7]. These practices enhance the use of TDD and guide the creation of unit tests. A sample of these best practices is provided below:

a) **Independent tests**
Tests should be able to be executed individually, without depending on other tests. If dependencies exist then when a test fails, it is difficult to identify the origin of the fault.

b) **Test only one functionality in each test**
Unit testing is used for asserting one unit at a time. Having multiple assertions in each test which do not relate to the same behaviour, introduces vagueness in the test cases.

c) **Mocks**
Mocks provide tests with the means to focus on the actual functionality tested without worrying about external dependencies, services or states.
Benefits

The main goals of TDD illustrate its major benefits, which are the minimization of defects and “the production of a reliable system” [8]. The repetition of testing and programming provides constant feedback to the programmer. When a bug causes a test to fail then it possibly lies in the code written during the last iteration or any code interacting with it. Taking into consideration that iterations involve as little code as possible, the amount of code that needs to be checked is less than the amount required if the testing was done in the end. As a result, the identification and fixing of bugs becomes easier and thus reduces the amount of time and effort spent on it. Daniel Brown states in his report [9] that “Quality Assurance in TDD becomes “pro-active” rather than “re-active”” and hence limits the density of bugs.

Another advantage of TDD is derived from its “focus on requirements” [9]. Programmers are “forced” by the principles of TDD to concentrate on individual requirements for each iteration and therefore gain a better understanding of them. Only well-defined requirements are implemented, while unnecessary features are excluded. Moreover, isolating functionality in well-defined and tested methods improves the design of the system. The integration of individual features becomes easier and has a higher probability of success [8].

The large amount of tests comprising the test suite of a system can be used as code documentation. Developers, other than the author, can take a look at the test cases for each function and understand what it does and what results it expects to give. This enables developers to work out the actual code fairly easy, helping the maintainability and extensibility of the system in the long term.

Issues

Productivity in TDD has been a subject for discussion. Studies [8] and [10] have shown a slight decrease in productivity as a direct result of the time spent on testing. The number of tests increases proportionally to the scope of the project and so does the time needed to run the tests for each iteration. There are occasions where programmers just wait for tests to complete without being able to do anything else. There is also the problem of over-testing, but there is a wide claim amongst TDD users [10] that this is a matter of experience and not related to TDD.
TDD cannot be applied on every situation. Testing user interface or interactions between the system and external applications are some examples where TDD falls short. In general, full functional tests either prove hard to implement in unit testing or the overhead necessary to configure them would greatly overshadow the benefits [9].

The developer and the tester are usually the same person. Having the same programmer write the tests and then produce the code might result in blind spots in the code [9]. Also, if the programmer does not have sufficient knowledge of the context in which a function is run, there is a chance a bug might appear. For example, if the programmer does not consider a set of values as possible arguments to a certain method then these values will not be validated in either the code or the test cases.

**Summary**

Test-Driven Development modifies traditional software engineering by making testing a key concept in the development lifecycle. It is not only a testing technique but most importantly a design technique. TDD practices have a huge impact on how a system is built and essentially what gets to be built. Although productivity might decline, the gains in code quality and requirement analysis appear to outweigh any issues [8].
2.3 Aspect-Oriented Programming

Overview

Created at Xerox Palo Alto Research Centre Incorporated (Xerox PARC) by Gregor Kiczales and a team of researchers [11], Aspect-Oriented Programming is the basic approach for Aspect-Oriented Software Development. AOP cannot be used as a standalone programming paradigm but rather as an extension to the existing Object-Oriented Programming paradigm [1]. It interacts with OOP to increase its expressiveness and capabilities.

The main idea behind AOP is separation of concerns. By concerns, we mean some functionality that is shared amongst different parts of the system. Take for example the login feature in a system, where a login mechanism is needed every time user-authentication is required. This requirement can appear in many classes at different points of execution and though login may vary depending on each situation, the actual authentication logic is common.

In OOP, a way to deal with the requirement would be to create a Login Class and add calls to the class at various places in the code that need authentication. As a result, maintainability becomes an issue. If a change is to occur then the programmers must keep track of all the parts of the system that would require modification. The issue gets worse as the complexity of the system increases. In AOP this shared requirement is considered a cross-cutting concern.

AOP comprises tools and methods that enable programmers to partition program logic in interchangeable modules called Aspects based on these cross-cutting concerns. So in respect to the example, an Aspect would be responsible for the login feature making the code both clean and reusable.
Terminology

Join Points
Join Points are well-defined points in the program flow [12].

Pointcut
The Pointcut is an expression composed from a set of Join-Points – which specifies points in the program where join points match. This point is where the cross-cutting concern is to be dealt with.

Advice
An advice is the actual code implementation of the cross-cutting concern, invoked when a point-cut is reached. The programmer has the ability to determine the time at which the advice is executed in respect to the associated point-cut (i.e. before, after or around the Join-point).

Aspect
An aspect is the combination of the pointcut and the advice.

AspectJ
AspectJ is an aspect-oriented programming extension for Java. It was created at PARC and it is supported by the Eclipse IDE. AspectJ has become the most widely used tool for AOP due to its simple and usable structure. It has similar syntax to Java but with added functionality to support aspects. Upon compilation an AspectJ program is rewritten to a suitable, valid Java program.

George Howard, in his report [13] compared AspectJ with other Aspect-Oriented languages like AspectC++ and PostSharp. He concluded that AspectJ is the best option for the following reasons:

- Active community (frequent updates)
- Comprehensive documentation and tutorials
- Free-to-use

A simple example in AspectJ based on the “Hello World Program” is provided below to gain a better understanding of the concepts of AOP.
Hello World in AspectJ

```java
// HelloWorld.java
public class HelloWorld {
    public static void main(String[] args) {
        say();
    }

    public static void say() {
        System.out.println("Hello");
    }
} // HelloWorld
```

A simple Java class that does as little as to call a method that prints “Hello”.

```java
// HelloWorldAspect.java
public aspect HelloWorldAspect {
    pointcut callSayMessage() : execution(public static void HelloWorld.say());

    before() : callSayMessage() {
        System.out.println("Good day!");
    }

    after() : callSayMessage() {
        System.out.println("Thank you!");
    }
} // HelloWorldAspect
```

Above is the aspect that advises the Java class.

**A step-by-step explanation of each the code follows:**

```java
public aspect HelloWorldAspect
```

Aspects start with a similar to declaration to Java classes. The difference is the “aspect” word used as type.

```java
pointcut callSayMessage() : execution(public static void HelloWorld.say());
```

Pointcuts are declared using the reserved word “pointcut” followed by the name of the pointcut. After “:” the joint-points are set. In this case, the point is reached when the method `say()` inside `HelloWorld` class is executed.

```java
before() : callSayMessage() {
    System.out.println("Good day!");
}
```

The declaration of an advice. The `before()` indicates that this is a `before advice`, which runs the associated code before the joint-point proceeds. In this case the print statement is executed before the `say()` method is executed.
This is an advice as well. It is similar as the previous one but with the difference that the code is executed after the `say()` method. *(After advice)*

The result when you compile and run the program is:

```
Good day!  ←  Printed by before() Advice
Hello  ←  Printed by the say() method
Thank you!  ←  Printed by the after() Advice
```

More realistic code can be found in Appendix A which contains parts of the banking system.

**Benefits**

**High System Cohesion**

Modularization has a positive impact on the structure of the system. Centralizing shared functionality increases cohesion, as it strips cross-cutting concerns from the core architecture of the program. Duplicating code across the system becomes unnecessary, enhancing the quality of the code and reducing the chance of getting a bug or error.

**Code Reusability**

Implementing functionality in individual modules, as part of AOP, promotes reusability. Aspects are “more loosely coupled than equivalent conventional implementations” [14]. This enables them to be reused in different projects without the need of excessive modification.

**Easier Extensibility / Maintainability**

Aspect-Oriented Programming provides an easy way to add extent or maintain a program. Aspects enable the addition or modification of existing functionality without making any changes to the core system. This proves very handy, especially when the source code cannot be accessed directly.
Issues

Complexity

Much of the criticism towards AOP focuses on its complexity. Indeed AOP, with all its tools and methods, can be difficult to grasp. Novice programmers might find it too complex and even expert developers will need both time and effort to master it [14]. Even AspectJ, despite its simplicity and usability, can be tricky. While its syntax is similar to Java, its declarations do not have similar usage in Java. Pointcuts or advices only exist in the context of Aspects. Empirical knowledge of Java cannot help here.

Execution Order and Code Behaviour

In traditional OOP, programmers can follow the code execution order just by looking directly at the code. This is not the case with Aspect-Oriented Programming. You can never look at the core program and be sure that it will not be cross-cut by an aspect at any point.

This complex execution order gives rise to another issue. Without knowing the code order, programmers cannot verify the behaviour of the code. An example to this difficulty is a call to a method that increases a given number by one. Though in traditional OOP it would be simple to verify, in AOP this becomes non-trivial. An Aspect might modify the behaviour of the method or even replace the entire method. In order to be able to reason about a piece of code, the programmer will need to have knowledge of all the aspects that advice the program.

Gary Pollice stated in his recent publication on Aspects [15] that “the way to think about correctness in AOP code is the inverse of how we consider it for object-oriented programming (OOP) code”. This describes exactly the issue. To verify the behaviour in OOP, you look the code first and then the context. In AOP, you go the other way around. First, you check the context (all aspects that advise the code) and then the actual code.

Testing

Another important issue with AOP is testing. Though in some aspect-oriented languages like PostSharp there is an annotation reference of the advising aspect in the base code, in others like AspectJ this does not apply. Unit testing has difficulties asserting aspects
in AspectJ because their behaviour in not referenced in the base program. Even unit testing of simple methods can fail due to the previously mentioned issue of code behaviour. Special configurations are required to include aspects when testing the system.

**Over-using Aspects**

AOP has seen an increase in popularity in recent year due to the easiness in which it can alter the behaviour of systems. This ability has led to an over-use of aspects. Modularization is no longer the only reason to apply AOP. Programmers tend to use aspect to fix problems within programs that would otherwise need modification of the actual source code.

**Summary**

Aspect-oriented Programming is a technology that promotes the idea of modularization of the cross-cutting concepts found in many applications. An example language is AspectJ which utilizes the power of aspects in the context of Java. With all its capabilities, AOP will have to be approached with care so that it does not get abused.
2.4 Testing techniques for Aspects

Overview
With the concepts of TDD and AOP been established, the background research continued to related work in the area. Only three studies were found to be closely related to the area of the project. The first study [16] examined TDD efficiency in modularization techniques in respect to compilation and testing time and the last two [17] and [18] proposed the use of the AdviceTracer tool in a test-driven approach to develop pointcuts. Neither the former nor the latter actually investigated ways of using TDD for AOP, though the tool looked promising. As a result of the limited related work, the research expanded to studies which concentrated on testing aspects [19] [20] [21], with the purpose of detecting which ones can be used in TDD.

The section will provide an introduction to the techniques identified and give examples of their use.

Test Patterns
Nicholas Lesiecki, in his article “AOP@Work: Unit test your aspects” [19], based on his experience with AspectJ, proposed the use of patterns to support aspect testing. These patterns, when applied to aspects, they allow the assertion of its different parts.

a) “Pattern: Test delegated advice logic”

Aims: Aspect Functionality

Overview
The pattern promotes the extraction of advice login from the aspect and delegates it to a separate class. The class can then be tested with common unit testing.

Benefits
- **Isolates advice code in separate class** - Test failure indicates a problem with the delegated code and not the aspect itself
- **Achieve even more separation of concerns** – The new class can be reused by the system even without the need to set the associated aspect.

Drawback
- Can only be used if the advice logic is extractable.
Without the delegation, the code is in the advice. After the delegation, the code is moved to the new class and the aspect just need to call the corresponding method in the class. That method can then be tested to assert if it does what it is supposed to do.

**b) “Pattern: Test flags set by Aspects”**

**Aims:** Aspect Specification

**Overview**

The pattern suggests the use of flags (variables that are set according to the occurrence of particular events). When a pointcut is reached, the associated advice - in addition to its relevant code - also raises a flag in a separate class. The flag/variable can then be asserted to check if it was set appropriately.

**Benefits**

- **Simplicity** – Only requires a simple variable and get/set methods

**Drawback**

- **Indirect testing** – The test can’t tell if the correct aspect raised the flag (unless stated explicitly)

**Example**

The advice calls the class that holds the flag variable and raises the corresponding flag. The test class can then check that the aspect was executed by checking that the flag was raised.
**AdviceTracer Tool**

AdviceTracer [22] is a tool created by the Triskel Team [23] that provides a neat mechanism for testing pointcuts. The tool is comprised of two classes:

1. **AdviceTracer Class:** The class responsible for the Pointcut/Advice tracing.
   **Methods:**
   - `setAdviceTracerOn()`: Starts the tracing process.
   - `setAdviceTracerOff()`: Stops the tracing process.
     - The methods which match the pointcuts under test should be called while the process is active.
   - `addTracedAdvice(String adviceName)`: Adds the advice name to the collection of traced advices.
   - `setTracedAdvices(Collection<String> adviceNameList)`: Sets the traced advices’ collection. Only the advices in the collection are traced. An empty collection indicates that every advice execution is traced.

2. **Assert Class:** The class that provides the assertion mechanisms for checking the execution of advices/pointcuts.
   **Methods:**
   - `assertAdviceExecutionEquals(int expectedNumber)`: Passes if the number of traced advices executions equals to the expectedNumber.
   - `assertExecutedAdvice(String adviceName)`: Passes if the specified advice name has been executed.
   - `assertExecutedAdviceAtJoinpoint(String adviceName, String joinpoint)`: Passes if the specified advice has been executed at the specified joinpoint.

The tool is used inside the test code and it does not require any changes to the actual program apart from naming each Advice by adding the annotation `@AdviceName("..")`.

An example use of the tool follows.
Example

Given a Sample Object:

```java
public class SampleObject {
    public void sampleMethod() {
        // ...
    }
}
```

And a Sample Aspect:

```java
public aspect SampleAspect {
    pointcut samplePointcut() : execution(void SampleObject.sampleMethod());
    @AdviceName("SamplePointcut")
    before(): samplePointcut() {
        // ...
    }
}
```

The SampleAspect declares a pointcut that matches the execution of `sampleMethod()` inside the `SampleObject`. A before advice is set for the pointcut. The advice is given a name by using the annotation `@AdviceName`.

The test class can then use the `AdviceTracer` to check whether the Aspect was executed correctly.

```java
public class Test {
    @Test
    public void testSampleAspectPointcut() {
        SampleObject so = new SampleObject();
        AdviceTracer.setAdviceTracerOn();
        so.sampleMethod();
        AdviceTracer.setAdviceTracerOff();
        Assert.assertExecutedAdvice("SamplePointcut");
    }
}
```

The testing order based on the above example is as followed:

1. The `SampleObject` is created.
2. The "AdviceTracer.setAdviceTracerOn()" is then called to start the tracing. Because no advice name was added to the traced list, the process will trace the execution of every advice.
3. The method that matches the pointcut is executed. I.e. “sampleMethod()”.
4. The "AdviceTracer.setAdviceTracerOff()" must then be called to stop the tracing.
5. The assertion "Assert.assertExecutedAdvice()" is called to assert that the advice was executed correctly.

Summary
A number of techniques exist that can tackle the testing issues of Aspects. Each one can be applied at a different phase of programming and have a different impact on the design of the system. Testing patterns affect the whole design while the AdviceTracer tool affects almost entirely the testing part.

2.5 Background Conclusion
This chapter provided the necessary background knowledge to support the project context. Detail reviews were presented for both Test-Driven Development and Aspect-Oriented Programming, along with an explanation of their key concepts and their advantages and disadvantages. Testing techniques for aspects were introduced along with examples of their use.

The following chapter describes the problem which this project tries to tackle.
Chapter 3: Problem Analysis

3.1 Overview
Having set the necessary background context, this chapter will analyse the problem arising from the research question introduced in the first chapter (i.e. “The combination of TDD with AOP is applicable and efficient enough for real world software systems”). It examines the integration challenge of the two technologies and establishes the key targets that guided the development of the project. Finally, the list of deliverables is provided, as well as the solution approach taken by the project.

3.2 Integration Challenge
Testing aspects, as already mentioned in Section 2.3, is a serious issue for Aspect-Oriented Programming, especially when AspectJ is used. Consequently, this also becomes a major challenge for the combination of TDD and AOP.

A number of questions come up as a result from the issue:

- **How can a programmer write a test for an aspect when he cannot access the aspect code from within a test class?**
- **How can a developer be sure that an aspect was called correctly and did what it was supposed to do?**

And in the case of TDD:

- **If you cannot test something, then how can you write a test that drives its development?**

In order to tackle the challenge, we need to analyse the problem. To do so, it is necessary to break down aspects.

An aspect is a combination of a “**Pointcut**” and an “**Advice**”. This means that both of them need to be asserted as part of aspect testing. As a result, we have the key targets that steered this project:

i. **Testing Pointcuts** (i.e. are the pointcuts matching the correct join-points?)

ii. **Testing Advice Code** (i.e. Does the advice code behaves as expected?)

By achieving the above targets, we can then proceed to answer the question of whether TDD can be applied for Aspect-Oriented Programming.
3.3 Deliverables

In accordance with the objectives described in section 1.2 and consistent with the project proposal [24], the project delivers the following artefacts:

a) A background review on Test-Driven Development, focusing on its processes and practices
b) A background review on Aspect-Oriented Programming, the AspectJ extension and their various implementation and compiling techniques.
c) A banking application that utilises both Aspect-Oriented Programming and Test-Driven Development
d) A set of proposed changes, if any, to the TDD practices, in order to straightforwardly apply it to AOP.
e) A set of possible issues, if any, as a result of the modifications to TDD in the previous deliverable.

3.4 Methodology

The methodologies applied for the project are divided into two categories, one based on background research and one on software implementation.

Background Research Methodology

Background research started with the identification of the project context. Once the context was specified, the actual reading began. Literature review focused around Test-Driven Development and Aspect-Oriented Programming. Research then continued to different techniques for testing aspects. Detailed summary of the two technologies and the testing techniques was presented in Chapter 2.

Software Implementation Methodology

Attempting to use Test-Driven Development for Aspect-Oriented Programming in a traditional, incremental development proved to be a non-trivial task. The integration issues, in addition to the complexity of AOP can greatly increase the risk of stalling development. Therefore, Software Reverse Engineering was selected instead.

“Software Reverse Engineering (SRE) is the practice of analysing a software system, either in whole or in part, to extract design and implementation information.” [25]
There are two forms of SRE [26] depending on the purpose for it is used:

- **Re-documentation**: Uses tools to generate documentation of the implementation of the system in the form of diagrams. E.g. class diagrams

- **Design Recovery**: It tries to extract the design steps by examining the implemented software.

SRE in the project was employed in the following phases to achieve Design Recovery:

1. **System Implementation**
   The banking application was developed, utilising aspects where they are needed.

2. **System Testing**
   Tests were produced for the system. In particular, tests for the aspects were created with their logic centred on the integration challenges explained in the previous section.

3. **Reverse Engineering**
   In this phase, every test was analysed to determine if it could have be written prior to the implementation and if it could have been used in a test-driven way to guide the actual implementation, in respect to the TDD principles. The procedure resembles a reversed Waterfall model. From the Implementation phase, you try to reach a valid Design Phase. In this case, the valid Design Phase is the one that uses TDD.

3.5 **Chapter Conclusion**

This chapter investigated the problem of the project and identified/explained its major challenge. It then stated the project deliverables and described the methodologies applied to the project.

The following chapter examines the banking system that was implemented as part of the project.
Chapter 4: Banking System

4.1 Overview
The chapter aims to give an overview of the Banking System, concentrating on the base code before proceeding to define its cross-cutting concerns. The overall system architecture is provided, along with explanation of its core components.

4.2 System Architecture
The system development started with an analysis of the main elements that comprise a banking application. These elements include:

<table>
<thead>
<tr>
<th>User (UserMode)</th>
<th>The person using the banking system. Each user has a corresponding mode while he accesses the system. Each mode has different privileges and access rights. E.g. an admin can access any account and do any operation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank Account</td>
<td>Each account is associated with a user. Different types of account are available with each one allowing different operations by default.</td>
</tr>
<tr>
<td>Transaction</td>
<td>A specific operation that is performed on an account or a number of accounts. In general transactions may or may not produce a result, but for this project, we consider that they always return one.</td>
</tr>
</tbody>
</table>

Table 1 System Elements

When the initial analysis was completed and based on the above elements, the different components of the system were designed and implemented in an iterative process.

In order for the application to work for multiple clients, a Server was created that is responsible for communicating with users and allowing admins to do some basic operations to accounts.

Figure 4.1 presents the architecture of the system in the form of a class diagram. Operations and attributes are omitted in order to be more comprehensive. The detailed class diagram can be found in Appendix B.

The following sections provide an overview of the main system components.
Figure 4.1 System Architecture
Bank Accounts

Bank accounts have a hierarchical order. An abstract class establishes the common functionality shared among different accounts and any extra account-specific functionality is set in the subclasses. Sample code from the BankAccount abstract class is provided below.

```java
public abstract class BankAccount ...
...
public BankAccount(String accountNum, Integer pass,
                     Hashtable<TransactionType, Boolean> ops) {
    accNum = accountNum;
    credential = pass;
    balance = 0;
    availableOperations = ops;
} // BankAccount

public void deposit(int amount) { ...
} // deposit

public void withdraw(int amount) { ...
} // withdraw

public int getBalance() { ...
} // getBalance

public abstract AccountType getAccountType();
...
```

Two types of accounts (Basic Account and Business Account) were created for better testing the system along with three basic operations (Deposit – Withdraw – getting Balance).

Responsible for the creation of the bank accounts is a Factory. This AccountFactory class assigns a unique number to the account and sets the default available operations for each type of account. The availableOperations variable is a HashTable with the type of transaction (e.g. Balance – Deposit – Withdraw - Transfer) as the key and a boolean as the value that states if the transaction is available. By default, the Balance/Deposit/Withdrawal Transactions are available to both account types, while the Transfer Transaction is only available to Business Accounts.

A Controller (AccountsController.java) is used to manage the list of accounts. Any modification to the list or an access to an account is done via the controller.
Transactions

Transactions are the most important part of the system. They incorporate the most crucial functionality of the system and most importantly, all aspects are related to them. This importance had an impact on both design and implementation as different approaches were tried until the final one was chosen.

An interface (Transaction.java) sets the methods which have to be implemented.

```java
public interface Transaction {
    Result executeTransaction();
    Integer getClientPassword();
    Global.UserMode getUserMode();
    Global.TransactionType getTransactionType();
    Integer getAmount();
    ArrayList<String> getAffectingAccNumbers();
    Integer getTransactionID();
}
```

In total four transactions were created, one for each basic account operation (BalanceTransaction – DepositTransaction – WithdrawTransaction) and one composite (TransferTransaction) that consists of a withdrawal operation on one account followed by a deposit operation on a second account.

The transactions are also split into two categories, Critical transactions and non-critical ones. Critical are the transactions which change the balance of an account (they require a write operation) i.e. Deposit/Withdraw/Transfer transactions. BalanceTransaction is considered a non-critical because it is just a read operation.

Each client is assigned a transaction controller (TransactionController) with the user mode and password provided as parameters. The controller can then create and execute any desired transaction and return the result of the execution.

The result of a transaction is also an important element. The way transactions either complete or fail have an impact on how aspects actually work and interact with the main application.

In the early approach, a transaction could fail due to a number of reasons (e.g. breaking an operation constraint) and it would throw an exception stating the reason of failure. In the end, a more flexible approach was utilised. Upon the execution of a transaction, a
result will always be created no matter what. The result contains two string variables, named status and info. The status indicates whether the transaction “COMPLETED” or “FAILED” and the info contains any relevant information about the transaction e.g. the balance returned by a BalanceTransaction or the reason that caused the transaction to fail.

As a result, all aspects take into consideration this approach and any aspect advising Transactions makes sure that it always returns a corresponding Result. Further explanation on how this is done will be provided in the next chapter.

Server
The Server part of the system is used for:

- Creating and managing Bank Accounts.
- Testing different Transactions.
- Supporting multiple users and so raising the need for synchronization handling.

And finally
- For providing a simple interface to the application

Because the main effort was focused on the underlying transaction handling, aspect-oriented programming and test-driven development, the graphical interface was kept as simple as possible.

The GUI comprises of three views:

1) A Server Information view as seen in Figure 4.2 containing client related data like the number of online users and connection details. It serves as an indicator of whether multiple users can and have connected to the application.
2) An **Accounts** view which displays the list of bank accounts.

![Figure 4.3 Accounts View](image)

The interface provides the tools to add an account or remove an existing one. Adding an account requires the selection of its type (in this case Basic or Business). The server then proceeds to create a default account of the chosen type and displays a confirmation message with the details of the account.

![Figure 4.4 Adding Bank Account](image)

In addition to Add and Remove, there is the option to view an account. It displays the balance of the account and allows the deposit or withdrawal from the account. Again the interface is quite simple, as it is mainly used for testing Transactions.

![Figure 4.5 Account View](image)
3) A **Transfer** view which is used to transfer an amount from one account to another.

Accounts can be chosen from dropdown lists, with information displayed underneath for the selected accounts.

![Transfer View](image)

**Figure 4.6 Transfer View**

The server is considered an admin tool and so all transactions executed in any of the views are under **admin mode**.
4.3 Cross-cutting Concerns

A number of cross-cutting concerns were identified for the system, based on its architecture. The cross-cutting functionality is closely related to Transactions and in particular to the fact that a transaction must always provide a result. They range from simple operation checking to concurrency handling and are being as realistic as possible to actual banking systems. The table below lists all four concerns:

<table>
<thead>
<tr>
<th>Cross-Cutting Functionality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Constraints</td>
<td>The system has to be able to enforce specific constraints to specific operations. For example, a withdrawal cannot be made on an account that has insufficient funds.</td>
</tr>
<tr>
<td>Transaction Logging</td>
<td>The system has to keep a log of all transaction related activities. E.g. deposit, withdraw</td>
</tr>
<tr>
<td>Transaction Synchronization</td>
<td>The system has to impose a synchronization mechanism. I.e. An account should not be modified by two transactions at the same time.</td>
</tr>
<tr>
<td>Transaction Security</td>
<td>The system needs to be able to enforce safety and security. This is a major concern because it requires both correct authentication and authorization. E.g. A user is authenticated against the accounts he attempts to access/modify and he is authorised to commit a specific transaction</td>
</tr>
</tbody>
</table>

Table 2 Cross-Cutting Concerns

4.4 Chapter Conclusion

This chapter described the basic functionality and architecture of the Banking System. Its main components were explained giving special attention to Transactions, which are the most important elements of the application. The chapter then proceeded to list the cross-cutting concerns of the system.

The next chapter concentrates on the development of these cross-cutting concerns.
Chapter 5: Cross-cutting Concerns

5.1 Overview

The chapter presents in detail the cross-cutting concerns defined in the previous chapter, in respect to the architecture of the base application. For each concern, a specific Aspect was created to deal with it. The development process is thoroughly explained, with special attention given to the Implementation and Test Phases that incorporate the techniques introduced in Section 2.4. As already mentioned, the aspect development cycle was reverse-engineered and the results of the process were evaluated to check whether TDD can be supported in AOP. While this methodology was used for the majority of the aspects, a test-driven approach was attempted during the development of the TransactionSecurity aspect in order to get an actual experience of TDD and AOP.

5.2 Aspect Enhanced System Architecture

With the base application implemented, the development continued with aspects. As mentioned in the previous chapter, four cross-cutting concerns were identified based on:

(a) Transaction Constraints (b) Transaction Logging (c) Transaction Synchronization
(d) Transaction Security

Before proceeding with the design, it was decided to create an aspect which sets all necessary pointcuts. The aspect – called Transactions – was declared abstract, so all others aspects would just need to inherit it in order to use the pointcuts. In total 5 pointcuts were specified as seen in the code below. Most of them are shared between more than one aspect so this approach brings flexibility to the code and reusability. In addition to the abstract aspect, precedence was declared for all aspects that advice the same code so that they can work properly.
The first two pointcuts – `deposit(...)` and `withdraw(...)` – are used for the Transaction Constraints and are join-points inside the BankAccount objects.

The third pointcut – `bank_operations(...)` – is the combination of the previous two and is for logging purposes.

The last two pointcuts match the `execute()` method inside the Transaction interface. Their difference is that the `transactions(...)` pointcut matches all Transactions that implement the interface while `critical_transactions(...)` only matches the critical ones.

In the parenthesis, for each pointcut, we state the parameters that are passed to the advice. For example take the following pointcut:

```
deposit(BankAccount account, int amount) && target(account) && args(amount)
```

And let say we have a `BasicAccount` named “`sampleAccount`” and the deposit method was called with a parameter stating the amount to deposit:

```
sampleAccount.deposit(amount);
```

At this line the joint-point is reached and the pointcut is matched. The `sampleAccount` object is taken as the `target(account)` and the `amount` is taken as the `args(amount)`. Both are then passed to the advice that uses the pointcut. The same applies to the other pointcuts, with each one taking its own parameters as required.

Figure 5.1 illustrates the architecture of the system including the aspects created to manage cross-cutting concerns.
Figure 5.1 System Architecture with Aspects
5.3 Transaction Constrains

Design
The TransactionConstraints aspect is meant as an operation handler. It validates an operation before it can carry on and enforce specific constraints. The system is protected from an undesired state cause by an erroneous input. For example, although trying to deposit a negative number is not considered a mathematical error, the system does not accept it and so it must not be allowed to complete. The rest of the constraints include:

- Trying to withdraw money from an account that has insufficient funds
- Trying to withdraw a negative amount.

The aspect must catch this invalid input and inform the system that the operation cannot continue. The information is in the form of an AccountOperationException with the reason passed as the exception message.

Implementation
The code below shows the actual implemented aspect class. There are two advices, one for the deposit and one for the withdraw pointcut. Inside the advices, the amount and/or account balance are checked against the constraints and if a constraint is found to be broken, an exception is thrown with a corresponding message indicating the reason.

```java
public aspect TransactionConstraints extends Transactions{

   before(BankAccount account, int amount) : deposit(account, amount) {
      if ( amount < 0 )
         throw new AccountOperationException("Deposit amount must not be negative.");
   } // before deposit

   before(BankAccount account, int amount) : withdraw(account, amount) {
      int accountBalance = account.getBalance();
      if ( accountBalance < amount )
         throw new AccountOperationException("The account contains insufficient funds.");

      if ( amount < 0 )
         throw new AccountOperationException("Withdrawal amount must not be negative.");
   } // before withdraw

} // TransactionConstraints
```

The exceptions must then be caught inside the transaction class and should be treated accordingly.
try {
    // ...
    account.deposit(amount);
    return new Result("COMPLETED", "");
} catch (AccountOperationException exception) {
    // Transaction Failed because it broke a TransactionConstraint
    return new Result("FAILED", exception.getMessage());
} // catch

The following figures show example messages from the application that inform about the constraints being broken.

![Deposit Constraint Example](image)

**Figure 5.2 Deposit Constraint Example**

![Withdrawal Constraint Example](image)

**Figure 5.3 Withdrawal Constraint Example**

**Testing**

Testing the aspect did not require any of the testing techniques mentioned in Section 2.3 as the standard JUnit framework was sufficient. An ExpectedException rule was used in the test methods to check if a constraint was broken. The rule asserts true if the expected type of exception provided as parameter is thrown during the execution of the test. It can also assert that the correct exception was thrown by checking the exception message. In this test case, the rule checks that the “account.deposit(-10)” throws an AccountOperationException with the given exception message due to the invalid amount provided as parameter.

```java
@Rule
public ExpectedException exception = ExpectedException.none(); // rule initialization
@Test
public void testDepositException() { // ...
    exception.expect(AccountOperationException.class); // Asserts type of exception
    exception.expectMessage("Deposit amount must not be negative"); // Asserts the message
    account.deposit(-10);
} // testDepositException
```
5.4 Transaction Synchronization

Design

The TransactionSynchronization aspect is responsible for achieving concurrency. The system allows the connection of multiple users and so it should provide the mechanisms to deal with synchronization issues.

Having many users accessing the same accounts involves the risk of getting an account in an incorrect state. While reading the balance of an account does not impose any risks, withdraw and deposit, if done at the same time, can interfere with one another.

For example, if two users login on the same account with balance of £100 and one user tries to deposit £10 while the other tries to withdraw £20, the system should provide a way of coping with it and showing the correct balance in the end.

The mechanism selected to deal with the issue has the following criteria:

1. Only one user is permitted to make changes to an account at any given time. If a second client tries to change the account, the transaction fails and a corresponding message is provided.

2. All accounts affected by a transaction must get locked prior to the execution else if even one account is already locked to another client then the transaction returns a failed result.

Implementation

An around advice was implemented to act as a wrapper around the execution method of the critical transactions. Based on the criteria, there are two possible outcomes:

- **At least one criterion is not satisfied:** The method is not allowed to proceed and the around advice must instead return an appropriate Result with a relevant status and info. As was stated before, an execution should always return a Result even if it has not completed.

- **Both criteria are met:** In this case, the advice proceeds with the transaction and returns the result.

The “**Pattern: Test delegated advice logic**” was used for the locking mechanism. The code was moved to a separate AccountLocker class that takes the list of accounts which need locking and unlocking.
When the lockAccounts() method of AccountLocker is called inside the advice, it returns true provided it could lock all accounts or false if it could not.

```
Create account locker for the list of affecting accounts
If the locker locks all accounts
    the result is the return value of the transaction
else
    a result is created with appropriate values
the locker unlocks all locked accounts
return the result
```

Pseudo code of TransactionSynchronization

**Testing**

The phase involved testing:
- The account locker class
- The aspect as a whole.

Delegating the code to the account locker made the test process a lot easier. Both the locking and unlocking functionalities were asserted using standard unit testing.

For the entire aspect testing, a slightly more complex strategy was used with a mock version of the UserThread object. The mock is given an operation, which is executed when the UserThread is initialised. In addition, a delay was introduced to the deposit operation to better support the test cases by allowing the operations to clash.

Each criterion was asserted individually in a separate test. The first checked a deposit and a withdrawal from the same account, as seen in the sample test code provided below. The second involved the deposit to an account followed by a transfer again to the same account.

In both cases, one operation should fail due to its inability to lock the desired account. The tests assert the results. They pass if the statuses of the results are not equal i.e. One completes and one fails. Standard output was also used to check the assertion by printing out the results as illustrated in Figure 5.4.

```
New connection with client# 1
New connection with client# 2
Deposit on account: acc0  Status: FAILED   Info: Another transaction on account acc0 is in process! Try again later!
Deposit on account: acc0  Status: COMPLETED Info:
-------------------------------
New connection with client# 1
New connection with client# 2
Deposit on account: acc0  Status: COMPLETED Info:
-------------------------------
Deposit on account: acc0  Status: FAILED   Info: Another transaction on account acc0 is in process! Try again later!
Transfer from: acc1 to acc0 Status: FAILED   Info: Another transaction on account acc0 is in process! Try again later!
```

Figure 5.4 Standard Output
5.5 Transaction Logger

Design

The logging concern of the system is achieved through the TransactionLogger aspect. The system must be able to log individual transactions and any relative info for persistence purposes. It was also decided to provide a log for specific operation on each account.

For example, when a Transfer transaction is executed, the transaction should get logged along with its details (account numbers, amount, time, status). Apart from the transaction, the withdrawal from one account and the deposit to the other one are also logged separately in their own log files. The account logging has to provide the state of the account prior and after the operation for better monitoring.

Implementation

The aspect is comprised of four advices:

1. LogBeforeOperation
2. LogAfterOperationNoError
3. LogAfterOperationWithError
4. LogTransaction

The (1), (2) and (3) are for account-specific logging and are called after the bank_operations pointcut (deposit and withdraw methods). They log the state of an account before and after the execution of the operation. They also include the status of the operation along with any relative information.

The last advice (4) is used for transaction logging. It is called after the execution of a critical transaction. Based on the result of the execution, the advice generates a log message that is send to a persistence class which outputs the message to the corresponding file.

The order, in which the advices are called, when a transaction gets executed, is the following:

1. The (1) advice is called before a deposit or withdrawal and logs the operation.
2. Then operations (2) or (3) are executed, depending on the result of the operation. If the operation completes correctly then the (2) advice is called, else if a transaction constraint was broken and an exception was thrown then the (3) advice is executed. In each case a corresponding log is generated.
3. The (4) advice is called after the execution of the transaction and logs its details.
Testing

The AdviceTracer Tool was employed to assert whether all pointcuts are correct and lead to the appropriate advice execution. In order for the tool to work, the @AdviceName annotation was set for all advices in the TransactionLogging aspect. This annotation provides a name to the advice so that it can be traced by the tool.

Two test cases were set, one for a transaction that includes a failed operation and one for a transaction that completes without errors.

The sample code below explains how the tool was used to assert the execution of the advice.

```java
AdviceTracer.addTracedAdvice("LogBeforeOperation");
AdviceTracer.addTracedAdvice(adviceName);
AdviceTracer.addTracedAdvice("LogTransaction");

AdviceTracer.setAdviceTracerOn();
tr.executeTransaction();
AdviceTracer.setAdviceTracerOff();

Assert.assertAdviceExecutionsEquals(3);
List<TraceElement> advList = AdviceTracer.getExecutedAdvices();
assertEquals("LogBeforeOperation", advList.get(0).getAdvice());
assertEquals(adviceName, advList.get(1).getAdvice());
assertEquals("LogTransaction", advList.get(2).getAdvice());
```

While checking the execution of the advice was plausible using the tool, asserting the log messages was a bit more difficult. Delegating the code to a separate class was not as straightforward as in the synchronization aspect, as it required writing additional, more complex code. For this reason, it was decided to do manual testing of the log files. This proved much more efficient and also gave an actual view of how the messages were displayed. Figures 5.7 and 5.8 provide sample views of the logs.
Figure 5.5 Sample Account Log

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Account</th>
<th>Type</th>
<th>Prior Balance</th>
<th>Operation</th>
<th>Status</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/03/2014</td>
<td>10:54:58</td>
<td>acc0</td>
<td>Deposit</td>
<td>0</td>
<td>Deposit</td>
<td>Pending</td>
<td>100</td>
</tr>
<tr>
<td>2/03/2014</td>
<td>10:54:59</td>
<td>acc0</td>
<td>Deposit</td>
<td>100</td>
<td>Deposit</td>
<td>Completed</td>
<td>100</td>
</tr>
<tr>
<td>3/03/2014</td>
<td>10:54:59</td>
<td>acc0</td>
<td>Withdrawal</td>
<td>100</td>
<td>Withdrawal</td>
<td>Pending</td>
<td>10</td>
</tr>
<tr>
<td>4/03/2014</td>
<td>10:54:59</td>
<td>acc0</td>
<td>Withdrawal</td>
<td>90</td>
<td>Withdrawal</td>
<td>Completed</td>
<td>20</td>
</tr>
<tr>
<td>5/03/2014</td>
<td>10:54:59</td>
<td>acc0</td>
<td>Withdrawal</td>
<td>80</td>
<td>Withdrawal</td>
<td>Completed</td>
<td>30</td>
</tr>
<tr>
<td>6/03/2014</td>
<td>10:54:59</td>
<td>acc0</td>
<td>Deposit</td>
<td>80</td>
<td>Deposit</td>
<td>Pending</td>
<td>100</td>
</tr>
<tr>
<td>7/03/2014</td>
<td>10:54:59</td>
<td>acc0</td>
<td>Withdrawal</td>
<td>180</td>
<td>Withdrawal</td>
<td>Completed</td>
<td>10</td>
</tr>
<tr>
<td>8/03/2014</td>
<td>10:54:59</td>
<td>acc0</td>
<td>Deposit</td>
<td>100</td>
<td>Deposit</td>
<td>Completed</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 5.6 Sample Transaction Log

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Type</th>
<th>Account</th>
<th>Description</th>
<th>Amount</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/03/2014</td>
<td>10:54:59</td>
<td>ADMIN</td>
<td>acc0</td>
<td>Deposit</td>
<td>100</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>2/03/2014</td>
<td>10:54:59</td>
<td>ADMIN</td>
<td>acc0</td>
<td>Withdrawal</td>
<td>10</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>3/03/2014</td>
<td>10:54:59</td>
<td>ADMIN</td>
<td>acc0</td>
<td>Transfer</td>
<td>10</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>4/03/2014</td>
<td>10:54:59</td>
<td>CLIENT</td>
<td>acc0</td>
<td>Deposit</td>
<td>100</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>5/03/2014</td>
<td>10:54:59</td>
<td>CLIENT</td>
<td>acc0</td>
<td>Withdrawal</td>
<td>10</td>
<td>COMPLETED</td>
</tr>
<tr>
<td>6/03/2014</td>
<td>10:54:59</td>
<td>CLIENT</td>
<td>acc0</td>
<td>Transfer</td>
<td>10</td>
<td>FAILED</td>
</tr>
</tbody>
</table>

Figure 5.7 Logging Test Cases

Finished after 0.674 seconds

Runs: 2/2  Errors: 0  Failures: 0

- Test_Suite.Test_TransactionLogging [Runner: JUnit-4] (0.593 s)
  - test_Logging_Transaction_Failed_Operation (0.077 s)
  - test_Logging_Transaction_Completed_Operation (0.516 s)
5.6 Transaction Security

Having created all of the previous aspects based on the reverse-engineering methodology, it was decided to combine all the testing techniques and attempt to use TDD for building the final TransactionSecurity aspect. Because testing pointcuts and advice execution was possible with both the AdviceTracer tool and the “Test Flags set by Aspects” pattern, two approaches were tried out. As for testing the advice code, the “Test delegated code” pattern was chosen.

A detailed description of the TDD stages follows.

Red Stage

In this stage the tests were written for each of the aspect elements. In the end of the stage all tests failed as no code was implemented yet, according to the TDD concept.

a) Testing pointcuts/Advice Execution

For both approaches a similar strategy was use. For every Transaction a corresponding method was called that asserted the pointcut.

“Test Flags set by Aspects” pattern

This approach required to check that the flag was raised after the execution of the transaction

```java
tr.executeTransaction();
assertEquals(true, SecurityHandler.transactionEvaluated()); // Assert that flag was raised
```

AdviceTracer tool

The tool approach is almost the same as in the TransactionLogger. The difference is that just one advice has to be traced and thus no order testing is needed.

```java
AdviceTracer.addTracedAdvice("TransactionSecurity");
AdviceTracer.setAdviceTracerOn();
tr.executeTransaction();
AdviceTracer.setAdviceTracerOff();
Assert.assertEquals(AdviceExecutionsEquals(1));
List<TraceElement> advList = AdviceTracer.getExecutedAdvices();
assertEquals("TransactionSecurity", advList.get(0).getAdvice());
```
b) **Testing advice code**

Testing the advice code is similar to the TransactionSynchronization aspect. The tests are written for the methods of the class (SecurityHandler.java) in which the advice logic will be delegated. The methods include authenticating a transaction (correct user password) and authorizing (allowed to execute the transaction). Due to the variety of Transactions and different UserModes available, tests cases were created for every possible scenario and asserted that the methods returned the correct Boolean value.

**Green Stage**

In this stage as little code as possible was written to pass the test cases. The **aspect** and the **SecurityHandler** class were created along with the pointcuts and the security methods (i.e. authentication and authorization). The **authentication** method retrieved the password provided with transaction and used it to authenticate the user against the affecting account. The **authorization** method verified that the transaction was available for the affecting account. In both cases, a Boolean value is returned according to the validation process. Also if the UserMode of the transaction is Admin then a **true** value is returned without any more validation. The methods are called inside the aspect, which depending on the returned values either returns an appropriate message or proceeds with the transaction.

In order for the Flag pattern to work, code needed to be introduced inside both the aspect and the SecurityHandler. The flag corresponds to a Boolean variable inside the SecurityHandler which indicates if the evaluation has been called. In addition, the flag also required methods just for test purposes:

- A set method for raising the flag.
- A getter method for retrieving the flag.
- A reset method for resetting the flag

The aspect calls the setter method to raise the flag and which the tests use to assert that the aspect has been executed.

At the end of this stage, all test cases from the Red stage passed as they should.
Refactor Stage

In the Refactor stage, the code quality was improved without affecting the behaviour of the aspect and any hardcoded data was modified to match the actual design. An optimisation was also included to speed up the evaluation procedure. If the UserMode of the transaction is “ADMIN” then the transaction proceeds without going through the validation.

The final version of the aspect is shown below.

```java
public aspect TransactionSecurity extends Transactions {

  @AdviceName("TransactionSecurity")
  Result around (Transaction transaction) : transactions(transaction) {
      SecurityHandler.setEvaluated();

      if (transaction.getUserMode() == UserMode.ADMIN)
          return proceed (transaction);

      if (!SecurityHandler.authenticate(transaction))
          return new Result("FAILED", "Not authenticated for transaction");

      if (!SecurityHandler.authorise(transaction))
          return new Result("FAILED", "Not authorized for transaction");

      return proceed(transaction);
  } // before basic_transactions
} // TransactionSecurity
```

5.7 Chapter Conclusion

This chapter described in detail the development process of the aspects created to handle the cross-cutting concerns of the system. Both the testing techniques and the standard JUnit framework were used to both test and drive the implementation of each aspect. The basic incremental approach was taken for most aspects in order to combine it with the reverse-engineering methodology. TDD was also tried out for the last aspect bringing together the techniques and experience from the previous development.

The next chapter analyses the results of the development process and tries to answer the research question of the project.
Chapter 6: Evaluation

6.1 Overview

According to the project phases explained in Section 3.4, the last phase of the project is the Reverse-Engineering which corresponds to the evaluation of the system. This chapter classifies the different possible outcomes of the project, followed by the reverse-engineering of the banking application. The TDD practice is then evaluated to detect any required changes and state any arising issues. Finally, a verdict is provided on whether the integration of TDD and AOP is feasible, together with a proposed TDD/AOP cycle.

6.2 Evaluation Approach

All possible outcomes of the project were identified early on, in order to produce an accurate and thorough evaluation plan. These outcomes were extracted from the questions which the project tried to answer.

The questions, as taken from the project’s proposal [24], are:

“How do TDD practices need to change when using AOP? What problems (if any) does this cause?”

The above questions make the assumption that TDD practices require modification when used with AOP. The project takes a broader view and does not assume this. It considers all possible scenarios.

The three possible outcomes identified were:

1. No integration of TDD and AOP
   This is the worst case scenario where the testing cannot drive the development of aspects even with the modification of TDD practices.

2. Full integration without any changes
   This is the best case scenario where TDD can be used as it is without changing any of it principles and without any resulting problems.

3. Partial integration with the need of modifying TDD
   This scenario states that practices of TDD need changes to manage integration with AOP. It can be further split in two cases:
   a) No problems caused from the modification
   b) Problems arise from modification.
6.3 Reverse-engineering Phase

Overview
In section 3.2, the key targets of the project were set. I.e. **Testing Pointcuts** and **Testing Advice Code**. The development of the banking application was focused on achieving these two targets by utilizing testing techniques and the JUnit framework. In this phase, the final implementation and test results are reverse-engineered to identify if they could have been used in a test-driven way in the first place. Finally, the findings are used to reason on the outcome of the integration.

The process is split between the two targets and for each one all relative technologies utilised are evaluated.

Pointcut Testing
Pointcut testing can be achieved by either the flag pattern or the AdviceTracer tool. Both techniques are quite similar in term of how testing is done. In the first case, the rising of the flag is asserted while in the second, the tool asserts that the appropriate advice is executed while tracing is on.

Their main different lies in the way each technology modifies the base code so that it can be employed. On one hand, the Advice Tool only requires the inclusion of a name for each advice using the @AdviceName annotation. On the other hand, the Flag pattern requires the addition of a new Boolean flag in the aspect code and extra methods for handling the flag.

The experience gained from the development of the aspects and especially in the final TDD attempt indicated that flags and the AdviceTracer can equally be utilized in a Test-Driven Development.

The AdviceTracer tool though is preferred over the flags due to the following reasons:

- **Simplicity & Efficiency**: The pointcut tests pass just by creating the pointcut/advice and adding the name annotation. The added code needed to support the trace process is negligible and so it requires less code and time to employ it.
Abilities: The tool not only asserts that the correct advices were called but also asserts the order in which they are executed, without any extra modification to the base code.

Advice Testing

Testing the advice code involved using both the testing techniques and the existing JUnit framework, depending on the implementing aspect. The “Test delegated advice logic” pattern was particularly used and proved to be really helpful. By extracting the advice logic in a separate class, it allowed the use of unit testing, which was impossible when the logic was still in the aspect. With unit testing being available, aspect testing becomes a lot easier. This, as a result, enables the test-driven development of the advice code.

The main disadvantage of the pattern is its dependence on the extractability of the advice logic. When the code can be delegated straightforwardly to a separate class then there are not any problems. If the code is too complex or too little then the gains from the delegation are overshadowed by the time needed to extract and test the code. An example is the TransactionLogger aspect. The advice code for logging transactions and operations could not be extracted easily and so the pattern could not be applied.

Apart from the pattern, the standard JUnit framework was used to test specific aspect behaviour. The ExpectedException rule can be employed to assert whether a code execution throws a particular type of exception together with the correct exception message. For example, in the case of the TransactionConstraint aspect, the rule implicitly asserted its overall behaviour (i.e. it validates transactions and throws exceptions when necessary) without explicitly testing the actual code of the aspect.

Mock objects were also used in TransactionSynchronization testing to assert the behaviour of the aspect in a multithreading environment. They helped mock out the state of the system and focus on the actual functionality.

In general, the delegation pattern provides an effective way of testing the advice logic and can be applied easily in a test-driven environment that drives the implementation of aspects. Furthermore, the ExpectedException rule and the mock objects are ideal
mechanisms to assist the testing of aspects which are meant for exception handling or require test-specific behaviour.

**Conclusion**

Summing up the reverse-engineering phase, it is obvious that pointcut testing and advice testing is achievable in TDD through the application of the aforementioned techniques. Of course, there are some limitations and restrictions imposed by the nature of the aspects but overall the integration of TDD and AOP is possible. Whether the integration is also desirable is evaluated in the next section, where any changes to TDD are detected and any problems are discussed.
6.4 TDD Evaluation

Overview
With the feasibility of the integration set, the evaluation can proceed with the analysis of the TDD and its practices. Any necessary changes to TDD are identified and reported, stating any resulting issues.

TDD Cycle
The TDD attempt, during the final aspect implementation, required the modification of the standard TDD steps in order for the development to progress correctly. In standard TDD a new cycle is applied for each new feature but this cannot be applied in AOP. New functionality implies the creation of a new aspect which is a more complex unit.

For a proper aspect development, the TDD cycle has to take into consideration both the pointcut part and the advice code part of the aspect. Both have to be tested and implemented before continuing to a new functionality.

Figure 6.1 illustrates how the cycle has to change in order to cope with aspect-oriented programming. In terms of phases, the following sequence demonstrates a sample cycle:

1) **Red Phase for Pointcut**: Test is written for the pointcut - advice execution
2) **Green Phase for Pointcut**: The aspect is created with the appropriate pointcut-advice declaration. The test passes.
3) **Red Phase for Advice Code**: Tests are written asserting the advice code
4) **Green Phase for Advice Code**: Just enough code to pass the test.
5) **Refactor Phase**: The code is refactored without changing its behaviour

TDD Practices
Looking back through the entire implementation phase of system, it is obvious that the best practices mentioned in Section 2.2 still exist when TDD is integrated with AOP. The aspect takes the place of objects and each unit test asserts a particular functionality (i.e. pointcut, advice code etc.). While the parts comprising an aspect are logically connected, their tests remain independent. Moreover, mocks can be employed to assist in aspect testing in the same way they do in TDD. Overall, the use of the TDD practices did not cause any issues.
Figure 6.1 Changes to TDD Steps
Implications for TDD

The proposed changes to TDD were evaluated and the next implications were detected:

- **Increased Complexity**: The addition of extra steps in the TDD cycle, combined with the AOP complexity leads to an increased complexity in TDD. The programmer not only has to follow the new cycle, but he also has to decide which techniques are most suited for each aspect element and the implementation details for the chosen one. For example if the programmer decides to use the advice delegation pattern, he must then extract the advice code, which is not always straightforward.

- **Loss in Productivity**: The use of TDD by itself results in a decrease in productivity. The loss gets worse when integrated with AOP due to the complexity issue mentioned above and due to the time needed for testing each aspect element before implementing it. The programmer has to write the test for the pointcut, write some code, stop to assert the code and then do the same for the advice code. This procedure takes more time than in the typical development process.

- **Tests have become harder to read.** One of the advantages of TDD is that the test code can be used as documentation for the application. Now the test code incorporates tools and patterns which the programmer needs to know in order to understand its purpose.

- **Complex when used on an existing system.** Trying to apply TDD in an existing AOP application can be a complicated task. It is necessary for the programmer to have detailed knowledge of all possible joint-points before using them in a new aspect development. Any wrong use of pointcuts or any unchecked advice execution can lead to an unknown failure of both the system and the test suite.

**Conclusion**

The evaluation of TDD has revealed an essential modification to the TDD cycle that is required to cope with the structure of aspects. No other changes were identified for any of the TDD practices. Finally, based on the required change a number of implications were detected and listed.
6.5 Project Result

Combining the findings of the reverse-engineering process and the evaluation of the TDD practice, it can be deduced that:

*The integration of TDD and AOP is possible with some modification of TDD.*

*(Outcome 3)*

Complexity issues arise from the combination of the two technologies but they are not sufficient to become a major problem for the development process.

During the development of the **base application**, TDD can be applied **non-trivially** without any changes. When an aspect is required to be implemented then a **complete TDD cycle for building aspects** is proposed with specific testing techniques used at each corresponding step. This cycle enables the best integration between the two technologies.

**Proposed TDD Cycle**

Figure 6.2 is similar to Figure 6.1 with the inclusion of the techniques at the test phases.
Test Driven Development process for creating Aspects

1) The test cycle starts with writing the pointcut tests using the AdviceTracer tool.

2) The Aspect is created along with the pointcut and the advice declaration. An advice name is provided using the @AdviceName annotation.

3) The pointcut test passes.

4) Test is written for the delegated advice code.

5) The advice logic is extracted to a separate class and code is written until the advice test passes.

6) The code is refactored to enhance quality and remove duplication and unnecessary hardcoded data.

6.6 Chapter Conclusion

To test the integration of AOP and TDD, a reverse-engineering process was performed on the banking application. In addition, TDD was analysed to detect changes in its practice and cycle. The results of both evaluations were satisfactory, as they revealed that TDD can be applied in AOP using the proposed development cycle.

The next chapter concludes the report and reflects on the project and any future work.
Chapter 7: Conclusion

7.1 Project Summary

The project set to determine whether Test-Driven Development can be applied in Aspect-Oriented Programming environment or if it requires changes in its practices to support the integration. Thorough background research was done for both technologies in order to gain an understanding of their concepts and principles.

The test issues of AOP and the limited related work forced the project to take a novice approach. A number of testing techniques were classified with the purpose of finding ways to test aspects and then detect if those tests could have been used in a test-driven way.

The above approach required the development of a banking application, which contained various cross-cutting concerns (transaction handling, synchronization, security and logging). Upon the implementation of the application, the reverse-engineering methodology was applied to evaluate the testing techniques. The TDD practice was also analysed to identify any necessary modifications and any resulting issues. The findings were combined to conclude on the outcome of the integration.

The project proved that TDD can be used for AOP with some changes to the TDD cycle and the introduction of testing techniques, despite the increase in complexity.

Additionally, an extra contribution was achieved at the end of the project. A TDD cycle was proposed that incorporated specific techniques, which when utilised can accomplish the most efficient way to apply TDD in AOP.

7.2 Reflection

The project helped the author enrich his IT knowledge with the concepts of TDD and AOP. Especially during the implementation of the banking system, the direct engagement with the AspectJ extension and the TDD principles enabled the author to gain an actual practical experience of the two technologies. Furthermore, the use of the reverse-engineering methodology along with different testing techniques has greatly enhanced his research, analytical and evaluating skills.
7.3 Future Work

While the project succeeded in providing an answer to the research question, more work could have been done which would have improved the project deliverables. A set of these extra tasks was identified which could be implemented in the future, using this project as stepping stone. The set consists of:

a) **An complete Test-Driven development of an Aspect-Oriented Application**

TDD was partially used for developing aspects. Given the TDD cycle proposed in this report, a complete TDD of an Aspect-Oriented application can be done which will provide more and better feedback for the integration of the two technologies.

b) **TDD supportability in other languages**

The project used the AspectJ extension language for implementing aspects. More languages are available for AOP which can also be evaluated to detect whether they support TDD.

c) **AspectJ Extension**

Using the testing techniques identified in the project, an investigation can begin on a tool for AspectJ that automates the Test/Implementation Phases of pointcuts and advice code.
Bibliography


[13] G. W. Howarth, An Investigation into Aspect-Oriented Programming,
Manchester, 2012.


Appendix A: Sample Banking System Code

**TransactionSynchronization.aj**

```java
public aspect TransactionSynchronization extends Transactions {
    @AdviceName("TransactionSynchronisation")
    Result around(Transaction transaction): critical_transactions(transaction){
        Result result = null;
        ArrayList<String> affectingAccounts =
            transaction.getAffectingAccNumbers();
        AccountLocker locker = new AccountLocker(affectingAccounts);

        if ( locker.lockAccounts() )
            result = proceed(transaction);
        else
            result = new Result("FAILED", locker.getError());

        locker.unlockAccounts();

        return result;
    } // around bank_operations
} // TransactionSynchronization
```

**TransferTransaction.java**

```java
public class TransferTransaction implements Transaction {
    private UserMode userMode;
    private String fromAccNumber, toAccNumber;
    private Integer transactionID, password;
    private int amount;
    private ArrayList<String> affectingAccNumbers;

    public TransferTransaction(int trID, UserMode mode, Integer pass, String fromAccNum, String toAccNum, int am) {
        transactionID = trID;
        userMode = mode;
        password = pass;
        fromAccNumber = fromAccNum;
        toAccNumber = toAccNum;
        amount = am;
        affectingAccNumbers = new ArrayList<>();
        affectingAccNumbers.add(fromAccNum);
        affectingAccNumbers.add(toAccNum);
    } // TransferTransaction

    @Override
    public Result executeTransaction() {
        try {
            AccountsController accountController = AccountsController.getInstance();

            // Retrieves the transfer from account
            BankAccount fromAccount = accountController.getAccount(fromAccNumber);

            // Retrieves the transfer to account
            BankAccount toAccount = accountController.getAccount(toAccNumber);

            fromAccount.withdraw(amount); // Withdraws the amount
            toAccount.deposit(amount);   // Deposits the amount
        }
    }
}
```
return new Result("COMPLETED", ""); // Returns the result
} // try
catch (AccountOperationException exception) {
    // Transaction Failed Because it broke a BankConstraint
    return new Result("FAILED", exception.getMessage());
} // catch
} // executeTransaction

@Override
public Integer getClientPassword() {
    return this.password;
} // getClientPassword

@Override
public UserMode getUserMode() {
    return this.userMode;
} // getUserMode

@Override
public TransactionType getTransactionType() {
    return TransactionType.TRANSFER;
} // getTransactionType

@Override
public Integer getAmount() {
    return this.amount;
} // getAmount

@Override
public ArrayList<String> getAffectingAccNumbers() {
    return this.affectingAccNumbers;
} // getAffectingAccNumber

@Override
public Integer getTransactionID() {
    return this.transactionID;
} // getTransactionNumber

Result.java

public class Result {
    private String status;
    private String info;

    public Result(String s, String i) {
        this.status = s;
        this.info = i;
    } // Pair

    public String getStatus() {
        return this.status;
    } // getKey

    public String getInfo() {
        return this.info;
    } // getValue
} // Pair
SecurityHandler.java

```java
public class SecurityHandler {
    private static boolean evaluated = false;

    public static boolean authenticated(Transaction tr) {
        if (tr.getUserMode() == UserMode.ADMIN)
            return true;

        String accNumToAuthenticate = tr.getAffectingAccNumbers().get(0);
        AccountsController accountController = AccountsController.getInstance();
        BankAccount accountToAuthenticate = accountController.getAccount(accNumToAuthenticate);

        Integer clientPass = tr.getClientPassword();
        if (clientPass == null)
            return false;
        return accountToAuthenticate.evaluateCredentials(clientPass);
    }

    public static boolean authorised(Transaction tr) {
        if (tr.getUserMode() == UserMode.ADMIN)
            return true;

        String accNumToAuthorise = tr.getAffectingAccNumbers().get(0);
        AccountsController accountController = AccountsController.getInstance();
        BankAccount accountToAuthorise = accountController.getAccount(accNumToAuthorise);

        return accountToAuthorise.getOperationAvailable(tr.getTransactionType());
    }

    public static void setEvaluated() {
        evaluated = true;
    }

    public static boolean transactionEvaluated() {
        return evaluated;
    }

    public static void resetEvaluation() {
        evaluated = false;
    }
}
```

/*
 * The following 3 methods are for testing purposes only.
 * They should not be used for any other purpose.
 * They are meant for a single transaction evaluation
 * They do not take into consideration synchronisation/concurrency.
 * evaluated is used as a flag that indicates whether the corresponding
 * aspect was called and the transaction was evaluated
 * transactionEvaluated() returns the flag, setEvaluated() raises the
 * flag and resetEvaluation() resets the value
 */
```
AccountLocker.java

```java
public class AccountLocker implements Runnable{
    private int numOfLockedAccounts;
    private ArrayList<String> accounts;
    private AccountsController accountController;
    private String error;
    private boolean lockCalled;
    private boolean outputOn = false;

    public AccountLocker (ArrayList<String> accs) {
        numOfLockedAccounts = 0;
        accounts = accs;
        accountController = AccountsController.getInstance();
        error = "";
        lockCalled = false;
    }

    public boolean lockAccounts() {
        lockCalled = false;
        BankAccount account;
        numOfLockedAccounts = 0;
        String accountNum;

        for ( int i = 0; i < accounts.size(); i++ ) {
            accountNum = accounts.get(i);
            account = accountController.getAccount(accountNum);

            output("Attempting to get Lock on Account: " + accountNum);
            if ( !account.lock().tryLock() ) {
                numOfLockedAccounts = i;
                error = "Another transaction on account " + accountNum
                            + " is in process! Try again later!";
                output(error);
                return false;
            }
        // if
        output("\tLock on Account: " + accountNum);
        numOfLockedAccounts++;
    } // for
    error = "No Errors";
    lockCalled = true;
    return true;
}

public void unlockAccounts() {
    String accountNum;
    BankAccount account;
    for ( int i = 0; i < numOfLockedAccounts; i++ ) {
        accountNum = accounts.get(i);
        account = accountController.getAccount(accountNum);
        account.lock().unlock();
        output("Released Lock on Account: " + accountNum);
    } // for
}
```
public String getError() {
    try {
        while (!lockCalled) {
            Thread.sleep(100);
        } // while
        return error;
    } catch (InterruptedException e) {
        return e.getMessage();
    } // catch
} // getError

private void output(String msg) {
    if (outputOn)
        System.out.println(msg);
} // output

@Override
public void run() {
    lockAccounts();
} // run
} // AccountLocker

UserThreadMock.java

public class UserThread extends Thread {
    private Socket socket;
    private int clientNumber;
    private TransactionController controller;
    private String accNumber;
    private Result result;
    private TransactionType selectedTransaction;
    private String toAccountNumber = null;

    public UserThread(Socket socket, int clientNum, String accNum) {
        this.socket = socket;
        this.clientNumber = clientNum;
        log("New connection with client# " + clientNumber);
        accNumber = accNum;
        controller = new TransactionController(UserMode.ADMIN);
        selectedTransaction = TransactionType.DEPOSIT;
    } // UserThread

    private boolean resultReceived;
    public void run() {
        resultReceived = false;
        switch (selectedTransaction) {
            case DEPOSIT :
                result = deposit(10);
                break;
            case WITHDRAW :
                result = withdraw(10);
                break;
            case TRANSFER :
                result = transfer(toAccountNumber, 10);
                break;
            default: break;
        } // switch
        resultReceived = true;
    } // run
public void selectOperation(TransactionType transaction, String toAccount) {
    selectedTransaction = transaction;
    toAccountNumber = toAccount;
} // selectOperation

public int getClientNumber() {
    return this.clientNumber;
} // getClientNumber

private void log(String message) {
    System.out.println(message);
} // log

public Result getResult() throws InterruptedException {
    while (!resultReceived) {
        Thread.sleep(500);
    } // while
    return result;
} // getResult

private Result deposit(int amount) {
    return controller.deposit(accNumber, amount);
} // deposit

private Result withdraw(int amount) {
    return controller.deposit(accNumber, amount);
} // withdraw

private Result transfer(String toAccNum, int amount) {
    return controller.transfer(accNumber, toAccNum, amount);
} // transfer
} // UserThread