Test-Driven Development for Aspect-Oriented Programming
Progress Report

Andreas Vlachou

Supervisor:
Dr John Sargeant

2013-2014
Abstract

Test-Driven Development has emerged as a quite new and fresh way of developing applications. Tests are no longer only used as means to validate and verify the specification and functionality of a system. They have become a design technique that drives the actual implementation of a system. Test-Driven Development fits well in the objects-oriented world, as objects form excellent candidates for unit tests. But what happens with Aspect-Oriented Programming? The separation of cross-cutting concerns has endorsed the need for modularization. Security and transaction management are merely some the concerns that Aspect-Oriented Programming tries to deal by utilizing 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 will examine the integration of the aforementioned technologies. A number of test patterns will be used along with existing Test-Driven Development principles. The findings will be analysed to identify any necessary changes to the Test-Driven Development principles. Any possible issues, as a result of the changes, will be reported.

This report describes the necessary literature review to support the project and explains the project plan along with the chosen research methodologies. Finally, it presents the progress done on the project so far in respect to the deliverables of the project.
Table of Contents

Abstract .......................................................................................................................... 2

Table of Contents .......................................................................................................... 3

List of Figures ................................................................................................................ 4

List of Tables .................................................................................................................. 4

Glossary .......................................................................................................................... 5

Chapter 1: Introduction .................................................................................................. 6
  1.1 Overview .................................................................................................................. 6
  1.2 Aim and Objectives .................................................................................................. 6
  1.3 Report Outline ......................................................................................................... 7

Chapter 2: Background and Literature Review .............................................................. 8
  2.1 Overview .................................................................................................................. 8
  2.2 Test-Driven Development ....................................................................................... 8
    2.2.1 Overview ........................................................................................................... 8
    2.2.2 Concepts ........................................................................................................... 8
    2.2.3 Benefits ............................................................................................................ 10
    2.2.4 Issues ............................................................................................................... 10
    2.2.5 Summary ......................................................................................................... 11
  2.3 Aspect-Oriented Programming ................................................................................ 12
    2.3.1 Overview ........................................................................................................... 12
    2.3.2 Terminology ...................................................................................................... 12
    2.3.3 AspectJ .............................................................................................................. 13
    2.3.4 Benefits ............................................................................................................ 15
    2.3.5 Issues ............................................................................................................... 15
    2.3.6 Summary ......................................................................................................... 16
  2.4 Background Conclusion ........................................................................................ 16

Chapter 3 – Project Progress ....................................................................................... 17
  3.1 Overview .................................................................................................................. 17
  3.2 Deliverables ............................................................................................................. 17
  3.3 Progress on Deliverables ....................................................................................... 17
  3.4 Unit Test Patterns for Aspects .............................................................................. 18
    3.4.1 “Pattern: Test delegated advice logic” ............................................................... 18
    3.4.2 Pattern: Test flags set by Aspects ................................................................. 19
List of Figures

Figure 1 - TDD Motto .................................................................................. 8
Figure 2 - TDD Steps .................................................................................. 9
Figure 3 - Delegation Example .................................................................. 18
Figure 4 - Flag Example ........................................................................... 19
Figure 5 - Class Diagram .......................................................................... 20

List of Tables

Table 1 - Progress on Deliverables .......................................................... 17
Table 2 - Cross-cutting concerns of banking application ...................... 20
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP</td>
<td>Aspect-Oriented Programming</td>
</tr>
<tr>
<td>TDD</td>
<td>Test-Driven Development</td>
</tr>
<tr>
<td>TFP</td>
<td>Test-First Programming</td>
</tr>
<tr>
<td>POP</td>
<td>Post-object programming</td>
</tr>
<tr>
<td>KISS</td>
<td>Keep-It-Stupid-Simple. Principle that promotes simplicity. Only make code complex when it is explicitly stated</td>
</tr>
<tr>
<td>YAGNI</td>
<td>You-Aint-Gonna-Need-It. Principle which states that functionality should be included only when need</td>
</tr>
</tbody>
</table>
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 computer programming like software design and software 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 testing comes before the implementation rather than after.

Aspect-Oriented Programming is a programming paradigm which aims to increase the expressiveness of Object-Oriented Design. [1] While OOP manages to represent the domain model of an application using high level of 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 “Pointcut” 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 will 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. A quite interesting idea is delegating the behaviour of Aspects in separate classes and then unit test those classes. This project will use this suggestion, identify more possible ones and apply them to determine if they cause any problems. Everything will be verified in the context of a banking application with cross-cutting concerns.

In brief, this project will try to answer 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:

1. Review the literature to gain an understanding on TDD practices and identify the key elements in AOP.
2. 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.
3. Identify difficulties and problems arising from the combination of the two technologies, based on the aforementioned application and suggest ways of refactoring TDD practices so that they better support AOP
4. Determine whether the proposed changes lead to any issues or restrict on any level the benefits obtained by each technology separately.

1.3 Report 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 – Project progress
This chapter lists the deliverables and the progress made so far in the project. It addition, it presents the candidate test patterns found so far. Finally, it illustrates some example code from the project application and describes how it will expand in the future.

Chapter 4: Research Methodology & Project plan
This chapter explains the methodology followed to conduct the research and states how the project application will be used to evaluate the integration of TDD and AOP. Furthermore, it details on the project plan, providing significant milestones and deadlines in respect to the project deliverables.

Chapter 5 – Conclusion
This final chapter provides an overall summary of the report and the future work.
Chapter 2: Background and Literature Review

2.1 Overview

This chapter aims to set the background context of the project based on the research done so far. Detail explanations of both TDD and AOP are provided below, presenting the basic concepts of each one as well as their benefits and any related issues.

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]

2.2.1 Overview

Test-Driven Development was introduced by Kent Beck in 2003 in his book “Test Driven Development: By Example” [3]. It is a technique based on 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.

2.2.2 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.

![Figure 1 - TDD Motto](image-url)
Red Stage

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 in 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, the existing test cases should continue to 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 - TDD Steps
2.2.3 Benefits

The main goal of TDD illustrates its major benefit, which is the minimization of bugs. [5] 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 that “Quality Assurance in TDD becomes “pro-active” rather than “re-active”” [6] and hence limits the density of bugs.

Another advantage of TDD is derived from its “focus on requirements” [6]. 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, which becomes more reliable. The integration of individual features is easier and has a higher probability of success. [5]

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 them to work out the actual code fairly easy, helping the maintainability and extensibility of the system in the future.

2.2.4 Issues

Productivity in TDD has been a subject for discussion. Studies have shown a slight decrease in productivity [7] 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 TDD people claim this is a matter of experience and not related to TDD. [7]

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. [6]

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 [6]. If the programmer does not have sufficient knowledge over 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 code or test.
2.2.5 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 has been built. Although productivity might decline, the gains in code quality and requirement analysis appear to outweigh any issues [5].
2.3 Aspect-Oriented Programming

2.3.1 Overview

Created at Xerox Palo Alto Research Centre Incorporated (Xerox PARC) by Gregor Kiczales and a team of researchers, Aspect-Oriented Programming is the basic approach of Aspect-Oriented Software Development [8]. AOP cannot be used as a standalone programming paradigm but rather as an extension to the existing Object-Oriented Programming paradigm. It interacts with OOP to increase its expressiveness and capabilities. [1]

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 based on each situation, the actual authentication is common.

In OOP, a way to deal with the requirement would be to create a Login Class and insert a call to it in the various places in the code that is needed. This results in maintainability problems. If a change occurs then the programmers must keep track of all the parts of the system that would require modification, which becomes a serious issue in complex systems. In AOP this shared requirement is considered a cross-cutting concern.

AOP comprises of tools and methods that enables 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 reusable and clean.

2.3.2 Terminology

Join Points
Join Points are well-defined points in the program flow [9].

Pointcut
The Pointcut is an expression - composed from a set of Join-Points – which specifies points in the program where all join points match. This point is where 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 (e.g. before, after or around the Join-point).

Aspect
An aspect is the combination of the pointcut and the advice.
2.3.3 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 re-written to a suitable, valid Java program.

George Howard, in his report “An Investigation into Aspect-Oriented Programming” [10], 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
```

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

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

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

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

Above is the aspect that advice 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 with the difference of the - aspect – word used as type.

```java
pointcut callsSayMessage() : call(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 reach when there is a call to the method say() inside HelloWorld class.

```java
before() : callsSayMessage() {
    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 say() method is called.

```java
after() : callsSayMessage() {
    System.out.println("Thank you");
}
```
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 sample code can be found in Appendix A which contains parts of the banking system.
2.3.4 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. As a result, duplicating code across the system becomes unnecessary thus 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” [11]. 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.

2.3.5 Issues

Complexity

Much of the criticism towards AOP focuses on its complexity. Indeed AOP, with all its tools and methods, can be very difficult to grasp. Novice programmers might find it too complex and even expert developers will need both time and effort to master it. [11] Even AspectJ, with all its simplicity and usability, can be tricky. While it uses Java-like syntax, 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’s 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, a programmer will need to have knowledge over all the aspects which advice the program.
Gary Pollice stated in his recent publish on Aspects [12] 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. Firstly, you check the context (all aspects that advice 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 reference in the base program. Even unit testing of simple methods can fail due to the aforementioned 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.

### 2.3.6 Summary

Aspect-oriented Programming is a quite new 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 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.

The following chapter will discuss the project progress made to the interim product in terms of the deliverables.
Chapter 3 – Project Progress:

3.1 Overview

This chapter list the deliverables of the project and presents in brief the progress made on them so far. It then goes on with a more detail description of two important parts of the final product. The first consists of two testing patterns used in unit testing of aspects which are chosen as candidate patterns for Test-Driven Development. The second illustrates the work done in respect to the banking application with the help of UML notation.

3.2 Deliverables

At the end of the project a list of artefacts will be produced. These artefacts will include:

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.

3.3 Progress on Deliverables

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) &amp; (B)</td>
<td>The deliverables are part of the background study and so they are completed at more than 90%. Any more research will focus on finding available testing patterns for aspects that can be used in TDD.</td>
</tr>
<tr>
<td>(C)</td>
<td>The deliverable is at the initial stage with about 30-40% of code written so far.</td>
</tr>
<tr>
<td>(D)</td>
<td>The deliverable will start immediately after the previous one (C), when the process of reverse engineering begins.</td>
</tr>
<tr>
<td>(E)</td>
<td>The final deliverable is part of the evaluation done at the end of the project.</td>
</tr>
</tbody>
</table>

Table 1 Progress on Deliverables
3.4 Unit Test Patterns for Aspects

Test cases have to be able to assert both the specification and the functionality of an aspect. Unit testing Aspects can be both tricky and complex due to the issues already mentioned in this report. However there exist a way of leveraging the complexity. The following patterns utilize the use of ordinary Java objects to simplify and ease the test process.

3.4.1 “Pattern: Test delegated advice logic”

Aims: Aspect Functionality

Overview
The pattern proposed by Nicholas Lesiecki [13] promotes the extraction of advice login from the aspect and delegate it to a separate class. The class can then be tested like common units.

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.

Example use in banking system

![Diagram showing Before and After Delegation](image)

Without the delegation, the authentication code is in the advice. After the delegation, the advice call the corresponding method in the AuthenticationClass. That method can then be tested to assert if the authentication process is correct.
### 3.4.2 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 use in banking system**

![Diagram](image)

Figure 4 - Flag Example

The authentication code inside the advice calls the class (ProgramStateClass) that holds the flag variable and sets the corresponding flag. The programmer, in the test case, can call the method which the aspect advices and then check whether the flag was raised, indicating that the authentication process was called.
3.5 **Project Application**

A necessary part of the project and an important deliverable is the banking application, which is required in order to investigate the integration of TDD and AOP.

A number of cross-cutting concerns have been identified as part of the application.

<table>
<thead>
<tr>
<th>Cross-Cutting Functionality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Activity logging</strong></td>
<td>The system has to keep a log of all activities related to each account. E.g. deposit, withdraw</td>
</tr>
<tr>
<td>2. <strong>Customer Authentication</strong></td>
<td>The system has to authenticate the customer before doing account related operations.</td>
</tr>
<tr>
<td>3. <strong>Account Operation Constraints</strong></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>4. <strong>Error handling</strong></td>
<td>The system needs an error handling mechanism that does not break the system.</td>
</tr>
<tr>
<td>5. <strong>Synchronization</strong></td>
<td>The system will have to impose a synchronization mechanisms. i.e. an account should not be modified by two clients at the same time</td>
</tr>
</tbody>
</table>

Table 2 – Cross-cutting concerns of banking application

A class diagram of the system implemented so far follows along with a brief description of each component.
**BankAccount:** An account interface stating the necessary methods for an account implementation.

**BasicAccount:** A class that implements the BankAccount Interface.

**OperationHandling:** An abstract aspect that declares the point-cuts based on the account operations. When either deposit() or withdraw() is executed then its corresponding pointcut is matched. The point-cuts are set in a separate abstract aspect so that aspects that extends them can take advantage of the point-cuts.

**ActivityLogger:** The aspect responsible for the logging concern. Advices are set before and after each point-cut (in OperationHandling) in order to log all information about the account and the operation in execution.

**BankConstraints:** The aspect responsible for enforcing constraints on account operations. Before the point-cuts set in in OperationHandling, the aspect calls a specific advice to validate the operation.

**ErrorHandling:** The aspect responsible for the exception managing concern. A point-cut is declared which is reached when an AccountOperationException is thrown.

**AccountOperationException:** A custom exception extending the Exception class. It is used to handle particular errors caused by invalid operations.

Sample code from the BankConstraints aspect is provided bellow. More code is available in Appendix A.

```java
package Aspects;

import BankOnAspect.AccountOperationException;

// Aspect that validates the account operations based on constraints
// amount must be >= 0 and the account must contain balance >= amount
public aspect BankConstraints extends OperationHandling{
    // Advice to check deposit(..)
    before(BankOnAspect.BankAccount account, int amount) : deposit(account, amount) {
        if ( amount < 0 )
            throw new AccountOperationException("Deposit amount must not be negative");
    }

    // Advice to check withdrawal
    before(BankOnAspect.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");
    }
}
```
3.5.1 Future Work

Based on the cross-cutting concerns shown in Table 2, at least two more functionalities will be implemented as part of the system. The functionality are:

- **SynchronizationAspect**: This aspect will be responsible for imposing locks. Before an operation it will check if the lock is available. If it is it will allow the operation to proceed, if not it will throw an exception. After the operation has finished, it will release the lock.

- **AuthenticationAspect**: This aspect will be responsible for authenticating the client before each operation. It will enforce that only the actual client can access/modify an account.

3.6 Summary

This chapter described the final product in terms of the deliverables and discussed the work done so far. Detailed information was focused on two of the interim products that will directly contribute towards achieving the project’s goal.

The next chapter will examine the research methodologies chosen for the project and will illustrate the project plan followed.
Chapter 4: Research Methodology & Project plan

4.1 Overview

This chapter describes the research methodology undertaken as part of the project. It details on the steps followed so far, as well as the steps which follow next. The chapter then discusses the evaluation procedure chosen to validate the project’s results. Finally, it presents the project plan with key tasks, important milestones and specific deadlines.

4.2 Research methodology

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

4.2.1 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. Detailed summary of the two technologies was presented in Chapter 2.

When an understanding of their concepts was established, the research continued to related work in the area. Similar studies, applying TDD to AOP, were not found and so the research expanded to unit testing of aspects, with the purpose of detecting testing patterns for aspects that can be used in TDD. Section 4.2 provides a description of the discovered patterns.

4.2.2 Software Implementation Methodology

Attempting to use TDD for AOP in traditional incremental development involves a high risk of stalling development. As already mentioned in Section 2.3.5, testing aspects proves to be quite difficult due to a number of issues, like the lack of referencing aspects inside the base code. These issues make the creation of tests, prior to the code, very complex. The programmer has to write tests that not only assert the behaviour of the aspect but the specification as well. This is non-trivial, especially in the case of a novice Aspect-Oriented programmer. Therefore, for all the above reasons, 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.” [14]

There are two forms of SRE [15] based on the purpose 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 is employed in the following phases to achieve Design Recovery:

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

2. **System Testing**
   Tests are produce for the system. In particular, tests for the aspects will be created with their logic centred on these questions:
   i. Are the pointcuts matching the correct join-points?
   ii. Does the advice reference the right pointcut?
   iii. Does the advice behave as expected?

3. **Reverse Engineering**
   In this phase, every test will be 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.

4.3 **Evaluation Approach**

All possible outcomes of the project have to be identified, in order to produce an accurate and thorough evaluation plan. These outcomes can be extracted from the questions which the project will try to answer.

The questions, as taken from the project’s proposal [16], are:
   a) “How do TDD practices need to change when using AOP?
   b) What problems (if any) does this cause?”

The above questions take 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 are:

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.
   The changes will be analyses in terms of their complexity, by measuring the implementation time and the lines of code.
## 4.4 Milestones

A set of milestones and deadlines has been specified that illustrates the project plan.

<table>
<thead>
<tr>
<th>Time Period</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>17/31 January:</td>
<td>Examine project description and establish the project context.</td>
</tr>
<tr>
<td>25 Jan/15 Feb:</td>
<td>Explore Test-Driven Development concepts.</td>
</tr>
<tr>
<td>16 Feb/1 March:</td>
<td>Explore Aspect-Oriented Programming concepts.</td>
</tr>
<tr>
<td>20 Feb/1 March:</td>
<td>First interaction with AspectJ.</td>
</tr>
<tr>
<td>1 March/ 6 March:</td>
<td>Writing Initial Report.</td>
</tr>
<tr>
<td>6 March:</td>
<td><strong>Deadline:</strong> Submit Initial Report</td>
</tr>
<tr>
<td>7 March/31 March:</td>
<td>Background read on integrating TDD with AOP and identify issues</td>
</tr>
<tr>
<td>19 March/26 March:</td>
<td>Determine the software implementation methodology for the banking application.</td>
</tr>
<tr>
<td>27 March:</td>
<td><strong>Milestone:</strong> Design of the banking application begins.</td>
</tr>
<tr>
<td>27 March:/3 April:</td>
<td>Identify system requirements and potential uses of aspects in the program.</td>
</tr>
<tr>
<td>27 March/ 9 April:</td>
<td>Design and implementation of the application functionality in the form of iterations</td>
</tr>
<tr>
<td>7 April/ 27 April:</td>
<td><strong>Easter Break</strong></td>
</tr>
<tr>
<td>7 April/ 1 June:</td>
<td>Identify candidate patterns for testing aspects (The task runs along with other tasks in same time period)</td>
</tr>
<tr>
<td>16 April/ 9 May</td>
<td>Writing Progress Report</td>
</tr>
<tr>
<td>9 May:</td>
<td><strong>Deadline:</strong> Submit Progress Report</td>
</tr>
<tr>
<td>10 May / 21 May</td>
<td><strong>Exams Period</strong></td>
</tr>
<tr>
<td>22 May / 10 June:</td>
<td>The implementation of the system continues</td>
</tr>
<tr>
<td>10 June:</td>
<td><strong>Milestone:</strong> System testing begins</td>
</tr>
</tbody>
</table>
10 June / 30 June: Candidate patterns and existing TDD principles are applied to the system, while refactoring the code to facilitate the test cases. (The task will progress in an iterative way)

15 June/ 5 July: Identify necessary changes to TDD principles

6 July: Milestone: Project evaluation begins

6 July/ 20 July Measure complexity of changes and classify issues caused by changes.

15 July / 4 September: Writing Final Report

5 September: Deadline: Final Report submission

4.5 Summary

This chapter described the methodologies selected for background research and software implementation. Furthermore, it explained the evaluation plan and illustrated the project plan, pointing out milestones and deadlines.

The next chapter will conclude the report.
Chapter 5 – Conclusion
The integration of Test-Driven Development with Aspect-Oriented Programming proves to be a complex process. The lack of related work in the area and the issues surrounding aspect testing, especially in the case of AspectJ, makes the project even more challenging.

This report served as an introduction to the project. It established the background needed to gain an understanding of the project context. Both TDD and AOP were discussed and their concepts explained. The progress made in terms of the deliverables and the interim product was provided and finally, the choice of methods followed for research, software implementation and evaluation was described. A detail project plan was also included to illustrate important tasks, milestones and deadlines.

5.1 Future Work
In the days to come, the project will continue with the design and implementation of the banking application. Work will then concentrate on testing the system. The aforementioned test patterns will be used, as well as the principles of TDD. During this period any necessary changes to TDD will be collected. The changes (if any) will then be analysed to identify any resulting issues. In parallel with the evaluation phase, the dissertation will be written up until its submission in September.
Bibliography

Appendix A

BankAccount.java

```java
package BankOnAspect;

public interface BankAccount {
    void deposit(int amount);
    void withdraw(int amount);
    int getBalance();
    int getAccNum();
}
``` // BankAccount

BasicAccount.java

```java
package BankOnAspect;

public class BasicAccount implements BankAccount {
    private int balance;
    private int accNum;

    public BasicAccount(int accountNum) {
        this.accNum = accountNum;
        balance = 0;
    } // BasicAccount

    @Override
    public void deposit(int amount) {
        balance += amount;
    } // deposit

    @Override
    public void withdraw(int amount) {
        balance -= amount;
    } // withdraw

    @Override
    public int getBalance() {
        return this.balance;
    } // getBalance

    @Override
    public int getAccNum() {
        return this.accNum;
    } // getAccNum
} // BasicAccount
```
**OperationHandling.aj**

```java
package Aspects;

public abstract aspect OperationHandling {

    // Matches execution of deposit method in BankAccount
    pointcut deposit(BankOnAspect.BankAccount account, int amount) :
        execution(void BankOnAspect.BankAccount.deposit(int)) && args(amount) && target(account);

    // Matches execution of withdraw method in BankAccount
    pointcut withdraw(BankOnAspect.BankAccount account, int amount) :
        execution(void BankOnAspect.BankAccount.withdraw(int)) && args(amount) && target(account);

} // OperationHandling
```

**ErrorHandling.aj**

```java
package Aspects;

import BankOnAspect.AccountOperationException;

public aspect ErrorHandling {

    pointcut handleException(AccountOperationException exception) :
        handler(AccountOperationException) && args(exception);

    before(AccountOperationException exception): handleException(exception) {
        String errorMessage = exception.getMessage();
        // Do something with message
    }
}
```

**AccountOperationException**

```java
package Aspects;

@SuppressWarnings("serial")
public class AccountOperationException extends Exception {

    public AccountOperationException(String message) {
        super(message);
    // AccountOperationException

    public String getMessage() {
        return this.getMessage();
    // getMessage
    }
}
```

Page 30 of 30