Aspect-Oriented Built-in Testing for Services

Thesis

Remco Steelink

Aspect-Oriented Built-in Testing for Services

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Remco Steelink
born in Amsterdam, the Netherlands



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2011 Remco Steelink. All rights reserved.

Aspect-Oriented Built-in Testing for Services

Author: Remco Steelink
Student id: 1047809
Email: R.Steelink@Student.TUDelft.nl

Abstract

Services are used a lot nowadays in order to deploy large scale systems that should never be shut down such as internet shops, online banking applications, and surveillance systems. Because services can be updated at any given time, testing is required during runtime by its clients to validate that the updated service still performs as required. However, testing during runtime may interfere with the nominal operations of the service and its environment. Furthermore, only the methods exposed by the service through its interface are available to the clients, which reduces the ability to control and observe the service for testing purposes, i.e. its testability. Built-in testing can be used to increase the testability, and to ensure that other clients are not hindered while the service is being tested. However, in order to increase the testability of the service, methods will have to be added to the service, which decreases the modularity and evolvability of the service. In order to maintain the modularity of the service, aspects can be used to separate the test facilities and test cases from the service. An approach has been devised to achieve built-in testing for services through the use of aspects. Our feasibility study and evaluation demonstrate that and how aspects can be used to achieve built-in testing for services. Further evaluation is, however, required to assess how our approach holds up when applied to a large real world service-oriented system.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Phil. H.-G. Gross, Faculty EEMCS, TU Delft
Committee Member:	Dr. L. Hollink, Faculty EEMCS, TU Delft

Preface

It has been quite some time since I started with my graduation project. Things went slowly for many reasons, but the main reason was that I had to deal with some personal issues. Although it took me eleven years to finally graduate, including a four-year break, I actually managed to do it! One thing I have learned is that there is no point to look back, since you cannot change the past. Especially the last couple of months I saw how fragile life really is and that we should not waste time. Life might be totally different tomorrow and for that reason we should enjoy every moment of it while we can. Many people around the world do not get the opportunity to study, so I am very grateful that I did get that chance.

With those life lessons out of the way, I would first like to thank my supervisor, Hans-Gerhard Gross, for his guidance throughout these years. Whenever I felt lost, unsure what to do next, his insights and expertise helped me continuing my work. I cannot thank him enough for being so patient with me. I would also like to thank my parents for always being there for me, supporting me in every way and giving me the chance to achieve this.

Remco Steelink
Purmerend, the Netherlands
September 23, 2011

Contents

Preface iii

Contents v

List of Figures vii

List of Tables ix

List of Listings xi

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 Outline . . . 3

2 Background and Related Work 5
- 2.1 Services and Testing . . . 5
  - 2.1.1 Test Phases . . . 6
  - 2.1.2 Service Related Tests . . . 7
  - 2.1.3 Testability . . . 8
  - 2.1.4 Test Isolation . . . 9
- 2.2 Built-in Testing . . . 9
- 2.3 Aspect-Oriented Testing . . . 10

3 Aspect-Oriented Built-in Testing for Services 13
- 3.1 Approach . . . 13
  - 3.1.1 Mechanism to Enable and Disable Testing . . . 14
  - 3.1.2 Service Test Cases as Aspects . . . 16
  - 3.1.3 Test Isolation . . . 19
  - 3.1.4 Invoking Unit and Integration Tests on the Service . . . 21
  - 3.1.5 Overhead . . . 21

4 Feasibility Study **23**
4.1 Goal . . . 23
4.2 Design . . . 23
4.2.1 Test Setup . . . 24
4.3 Results . . . 27
4.3.1 Mechanism to Enable and Disable Testing . . . 28
4.3.2 Service Test Cases as Aspects . . . 30
4.3.3 Test Isolation . . . 36
4.3.4 Invoking Unit and Integration Tests on the Service . . . 39
4.3.5 Overhead . . . 40
4.4 Discussion of the Feasibility Study Results . . . 43
4.4.1 Mechanism to Enable and Disable Testing . . . 43
4.4.2 Service Test Cases as Aspects . . . 44
4.4.3 Test Isolation . . . 44
4.4.4 Invoking Unit and Integration Tests on the Service . . . 45
4.4.5 Overhead . . . 45
4.5 Concluding Remarks . . . 46

5 Evaluation and Threats to Validity **47**
5.1 Evaluation . . . 47
5.1.1 Mechanism to Enable and Disable Testing . . . 48
5.1.2 Service Test Cases as Aspects . . . 48
5.1.3 Test Isolation . . . 48
5.1.4 Invoking Unit and Integration Tests on the Service . . . 49
5.1.5 Overhead . . . 49
5.2 Threats to Validity . . . 49

6 Conclusions and Future Work **51**
6.1 Summary and Contributions . . . 51
6.2 Conclusions . . . 52
6.3 Future work . . . 53

Bibliography **55**

A Example of Using Aspects **59**

B Schema of Test Setup Database **61**

C Glossary **63**

List of Figures

2.1 Service-Oriented Architecture 6
2.2 Simplified V-model 7
2.3 Weaving test code as aspects into a service to create a testable service 11

3.1 Class diagram: Aspect-oriented built-in testing for services 22

4.1 JAX-WS Architecture 24
4.2 Class diagram: Test setup consisting of four JAX-WS services, two clients and a database 25
4.3 Class diagram: Test setup aspected with BIT technology 41

B.1 Schema of the database used in the test setup 61

List of Tables

4.1 Average execution time in milliseconds of `getBook` after 5x1000 calls 42
4.2 Required code for each concept, `x` is variable 43

List of Listings

3.1 Pseudocode: Mechanism to enable and disable testing 15
3.2 Pseudocode: Publishing test of a service 16
3.3 Pseudocode: Discovery test of a service 16
3.4 Pseudocode: Binding test of a service 17
3.5 Pseudocode: Composition test of a service 17
3.6 Pseudocode: Execution test for variable `dummy` 18
3.7 Pseudocode: Execution test for variable `dummy` with locking 18
3.8 Pseudocode: Test isolation by skipping an operation 19
3.9 Pseudocode: Test isolation by saving and restoring a variable 19
3.10 Pseudocode: Inform another service that testing is enabled 20
3.11 Pseudocode: Invoking unit and integration tests on the service 21
3.12 Pseudocode: Measuring performance of the service 22
4.1 Interface of JAX-WS service `BookManager` 25
4.2 Implementation of JAX-WS service `BookManager` 26
4.3 Client including testing information by modifying the HTTP request headers 28
4.4 Service enabling test mode for clients tagging their messages with testing information 28
4.5 Condition `testingEnabled` added to a pointcut 29
4.6 Aspect for delaying the execution of `loanBook` 29
4.7 Integration test case: Availability of the database required by `BookManager` 30
4.8 Discovery and binding of service `Logger` 30
4.9 Integration test case: Discovery of service `Logger` 31
4.10 Integration test case: Binding of service `Logger` 31
4.11 Integration test case: Response time of `Logger` should not exceed 5 seconds 32
4.12 Unit test case: Method `loanBook` 33
4.13 Unit test case: Method `loanBook` with locking 34
4.14 Unit test case: Control flow of method `loanBook` 35
4.15 Integration test case: Expected response from `Logger` 36
4.16 Test isolation by skipping variable assignment and database update 37
4.17 Test isolation by restoring the variable and database 37
4.18 Test isolation for service `Logger` 39

4.19 Test isolation by deleting inserted log after executing `addLog` 39
4.20 Adding test operations to the interface of `BookManager` 39
4.21 Implementation of the test operations for `BookManager` 40
4.22 Client code for the performance test comparison 41
A.1 Adding two numbers with method `calculate` and printing the result 59
A.2 Printing messages before and after execution of method `calculate` with an aspect 59
A.3 Changing behavior of method `calculate` with an aspect 60

Chapter 1

Introduction

Services are used a lot nowadays in order to deploy large scale systems that should never be shut down such as internet shops, online banking applications, and surveillance systems. Services extend the concepts of components, where complex problems are decomposed into smaller more amenable parts, increasing their reusability [26]. Both, components and services, use encapsulation to reduce the overall complexity of the software development process and they may only be accessed through their interfaces. Services are more loosely coupled than components, which means that they have a minimum amount of interdependencies with other services, and their coupling is dynamic by nature. Unlike components, services are not bound to a particular platform and can be invoked between platforms. Services can be discovered and bound at any given time by other services and they can interact with each other through service contracts. Services provide a better way to expose business functions and are, therefore, useful to develop applications that support the business process of organizations. A service composition is typically distributed, e.g. web services, under different ownership and, hence, not directly available to the user of the service.

The characteristics of services present some issues that can affect the quality of the software that uses these services [9]. One issue comes from the fact that the same service is used in many different contexts. A service might function flawlessly when used by one client but fail when used by another client, simply because services have not initially been designed to interact in any arbitrary environment, nor have they been tested for every possible context. For example, the service might return a date in a certain format that is not expected by the client. Another quality issue arises from the fact that services are encapsulated. Since they hide their inner workings it is hard to control and observe their behavior. When the source code is available this is not really a problem, but usually it is not available to the service user. Even if it would be available, the user could easily get lost in too much detail. Lastly, ownership of the service is also an issue since the service user does not own the service. This might cause problems when the functionality of the service is updated, because this update does not necessarily have to be visible through its interface. The service provider might simply not always be aware of who is using their service, making it hard to notify those users.

The single most important technique for addressing quality issues in component-based systems, and hence service-oriented systems, is testing [2, 9, 10, 12, 29]. With testing, oper-

ation scenarios (test cases) are defined to detect mismatches between the observed behavior and the service specifications, i.e. failures. Different test approaches exist for defining those test cases. A good test approach will have to take the dynamic nature of services into account and it should be possible to test the service during runtime, while ensuring continued nominal operations for other clients [19]. Since it is too costly to recreate (part of) the test cases each time a service is integrated into a new environment, test cases should be reused as much as possible.

Built-in testing (BIT) is such an approach that can facilitate making the service more controllable and observable for testing purposes [21, 32]. Test facilities, e.g. code that sets a service in a certain state for testing, and test cases are added to the service to provide more control over the service and to provide the means to observe possible problems of the service. Furthermore, by the use of test interfaces, BIT enables a service to test and be tested by its environment during runtime. Nominal operations for other clients during runtime testing can be ensured with BIT by changing the behavior of methods.

One way to achieve BIT is by using aspects for the required test facilities and test cases. Aspects can be used to separate concerns that cut across other concerns [25]. An example of a cross-cutting concern is a variable that is locked before use, preventing simultaneous adjustment of this variable. The code that locks the variable is removed and implemented as an aspect. In the same way, test facilities and test cases can be implemented as aspects and separated from the original service. One of the main advantages of using aspects is that the original source code does not have to be modified, even when trying to access private members of a class. The test facilities and test cases are located in another file and they are woven into the compiled service. The same level of testability, which influences the degree to which a specific software entity can be tested, is retained compared with other BIT approaches [6]. Modularity and evolvability are less difficult to achieve by using this approach.

1.1 Problem Statement

Since services are dynamic and often have to be tested during runtime, built-in testing can be a suitable approach to include test facilities and test cases with the service, which also increases the reusability of these test facilities and test cases. The traditional approach for BIT is to extend the base class with the test facilities and test cases, or by adding the test facilities and test cases directly to the base class. A disadvantage of extending the base class, however, is that private members are not visible, and a disadvantage of adding the code directly to the service is that it makes the services less modular and harder to evolve.

In order to maintain the modularity and evolvability of services, aspects can be used to achieve built-in test capabilities for services. There are several problems that have to be solved for this to be a viable approach for testing services. First of all, services can produce a variety of failures, such as failures that occur when a service is not found at the specified location. Test cases, written as aspects, will have to be created to this end. The way aspects are applied to the service also raises an issue. Because aspects are woven into the code by the aspect compiler, they are always present and applied to the operations. This is not really

an issue if a copy of the service is used for testing, but this might take too many resources when the service is complex. If the same service is used for both nominal mode and test mode, then a mechanism is required to turn testing on and off depending on the client that is using the service. This is especially important when testing is done during runtime as other clients should be able to use the service normally without any interference through testing by other clients. In order to achieve this, the service will have to know which client wants to test a specific method and apply the aspects to that specific method for that client only. Furthermore, certain method invocations and variable or state changes are not allowed while testing is performed during runtime since they could interfere with nominal operations. This is also referred to as test isolation [18, 31]. The tested operations will either have to be skipped completely or the changes made to certain variables will have to be reverted back to their original value after testing. Another feature that is required is the ability to invoke unit and integration tests on the service in order to check if the service functions as determined in its specifications. Lastly, the overhead of using aspects for the BIT facilities will have to be determined. Using aspects for the test facilities and test cases might not be interesting if the performance of the service is lowered too much or if it requires too much additional coding by the development team. Consequently, the following research questions will be treated in this thesis:

How can aspects be used to achieve built-in testing for services?

- Which mechanism can be used to differentiate between nominal mode and test mode?
- To which extent can test cases for services be implemented as aspects?
- To which extent can test isolation be achieved during runtime testing with aspects?
- How can unit and integration tests on a service be invoked?
- How much overhead is generated when aspects are used to achieve built-in testing for services?

The answers to these questions are given throughout the thesis and evaluated with a feasibility study.

1.2 Outline

The remaining chapters in this thesis are structured as follows. Chapter two gives a background on the topic of testing and the work that is related to this thesis.

The information gathered in chapter two is used in chapter three to devise our approach to achieve built-in testing through the use of aspects for services. A section is devoted to each sub research question to detail our approach in that regard.

The fourth chapter presents a feasibility study where our approach has been applied to a small service-oriented system, consisting of four web services, two clients and a database. For each sub research question a section is used to show the results. Furthermore, a discussion on these results is given.

An evaluation of our approach on an abstract level and the threats to the validity of our approach follows in chapter five.

Finally, in chapter six this thesis is summarized, our contributions are presented, conclusions are drawn, and suggestions of how this work can be followed up are given.

Chapter 2

Background and Related Work

Before actually explaining our approach, a background on the subject and the work that is related to our thesis will be given in this chapter. First, testing for services in general is introduced, detailing the different test phases, service related tests, test isolation, and the notion of testability. Furthermore, several different BIT approaches are introduced. Finally, the principles of aspects and how they can be used for testing are discussed.

2.1 Services and Testing

Let us first start with giving a definition of a service [5]:

A service is a course-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model.

Services are built from a business perspective and are based on high-level business concepts and not so much on the technical details (low-level). They offer a product, i.e. service, to their customers, i.e. clients. Services are often distributed, written in different languages, and communicate with their clients through messages over a network. A service is registered by the service provider and discovered by the service client, which can be a user or another service. After the service has been discovered, the client can bind the service and invoke operations on the interface of the service. Figure 2.1 depicts the architecture of a system consisting of services [13].

The characteristics of services present issues that can affect the quality of the software that uses these services [9], such as the fact that the service user does not own the service. When the functionality of the service is updated, it might cause problems for the service user, because this update does not necessarily have to be visible through its interface. Furthermore, services are harder to control and observe since they are encapsulated and only their interface is available. Their dynamic coupling also allows services to join and leave at any given time, which means that the service might not always be available.

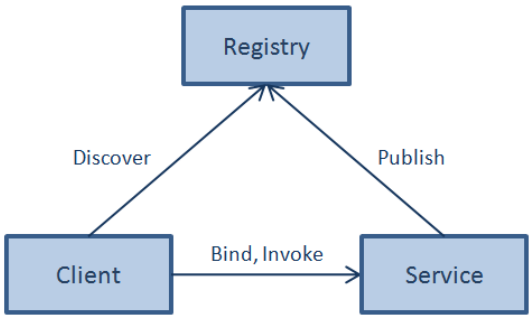


Figure 2.1: Service-Oriented Architecture

In order to address these quality issues and to assess that the service operates according to its specifications, testing is required. Testing is defined in the IEEE glossary of software engineering [22] as:

(1) The process of operating a system or component/service under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component/service. (2) The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.

In other words, testing is used to detect failures, i.e. observed misbehaviors. Services are tested by both the provider and the user of the service.

2.1.1 Test Phases

Testing is usually done during different phases in the development process. For the classification of the test phases for service-oriented testing, the V-model will be used [8]. Although a somewhat dated development model, the V-model sufficiently details the development phases and related testing activities. It is shown in figure 2.2.

On the left side are the user requirements and the design documents, and on the right side is the actual implementation of these documents. The V-model defines 3 fundamental test phases: unit testing, integration testing and system acceptance testing. A unit is tested following its specification stated in the design models, the integrated software (integrated units) is tested according to its system design and lastly the final system is tested according to the user requirements. When changes are made to the system once it is operational, regression testing is done to ensure that the system is still running according to its specifications.

Since the goal of testing in every test phase is to detect failures, it is important that test cases are well defined. A test case is an input, together with expected results and a set of execution conditions, on which the software entity is tested [22]. An example test case would be to validate that a huge value, e.g. 999999999999999, does not cause an overflow error. Since a single test case is usually not enough to cover all criteria, a test suite is required. A test suite is a collection of test cases for testing a specific software entity.

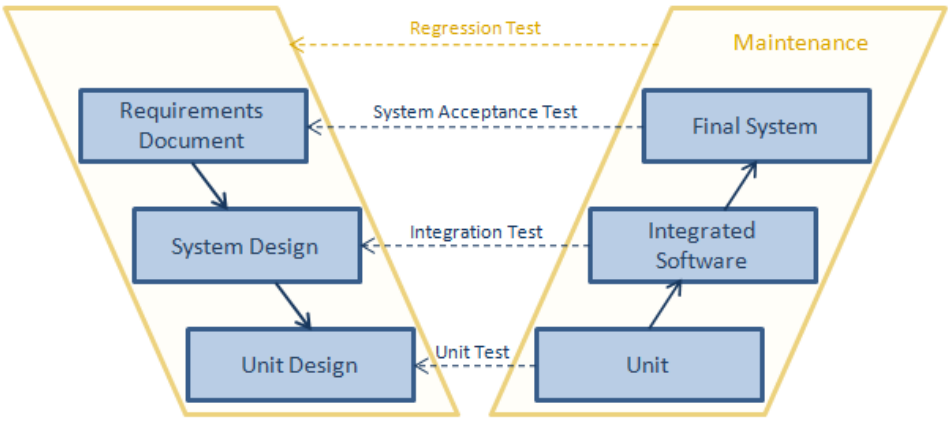


Figure 2.2: Simplified V-model

For our thesis we will only focus on unit and integration testing. For unit testing, services are tested independently of the applications they will be used in. This is usually done by the service provider. Integration testing is done by the service provider in cases where the service requires another service to function correctly, and by the service user in cases where the service is integrated in an application. Furthermore, integration testing will usually have to be done during runtime since only then they can be fully tested.

2.1.2 Service Related Tests

Some failures that are produced by services are related to the service life-cycle, such as failures that occur when the service is not correctly deployed. According to Brüning et al. [7] five types of possible faults can be described in a service-oriented system, namely: publishing, discovery, composition, binding and execution faults.

- *Publishing faults* can be faults where the description of the service is incorrect or when the service is not successfully deployed. Resources for the service might also be missing, e.g. a link to a database. The service is started but will fail to perform correctly.
- *Discovery faults* are faults where a certain service cannot be found or when the wrong service is found.
- *Binding faults* can be related to access restrictions. Certain services might not be run without having the correct certificate. It can also occur when the wrong service is bound without the client noticing.
- *Composition faults* occur when services are incompatible and cannot be connected with each other or if a contract is not met.
- *Execution faults* happen when the result of a service does not match the expected result. This can be caused by incorrect input or a faulty service.

Although this fault taxonomy has some shortcomings according to Greiler et al. [20], such as the lack of a description of faults that occur when a service is being updated, it gives us an insight of the test cases that have to be implemented as aspects to detect the failures that these faults cause.

2.1.3 Testability

An important aspect in software testing is testability [22], which expresses the degree to which a specific software entity can be tested. In other words, testability is the extent to which the software can hide faults and the extent to which tests can uncover faults. Since services are encapsulated, testability plays an important role in testing them.

Two important key concepts in testability are controllability and observability [16]. Controllability is the ability to feed the software with certain input to set it in a certain state. Not every state might be of interest to the developer. Usually certain combinations of states are of interest. Observability is the extent to which the output state and failures can be observed. Sometimes existing output states might not be enough in cases that errors are not propagating to lead to a failure. Additional output states might be required in order to increase observability.

Binder discusses six testability factors which may either facilitate or hinder testing [3]:

- *Representation*. This encompasses the completeness of the software design documentation, such as the requirements specification of the software.

- *Implementation*, which depends on the used programming language and paradigms. For example, when using an object-oriented language, inheritance might cause problems. When a class that has a lot of subclasses is changed, all of its subclasses will have to be tested too.

- *Built-in test*. The ability to equip software with, for example, extra set/get methods to set a software system into a certain state to facilitate testing.

- *Test suite*, which is a collection of test cases and plans to use them. This can make testing easier, especially since they can be reused for regression testing.

- *Test tools*, which are tools to automate testing in a certain way, such as test frameworks and scripts.

- *Test process*. The capability and maturity of the staff is important, but also the amount of spent resources on testing and the integration of testing throughout the development process.

Services reduce controllability and observability, which in turn reduces their testability. Without access to their inner workings, it is harder to trace failures back to faults.

2.1.4 Test Isolation

In order to ensure nominal operations during runtime for other clients, test isolation will have to be used [18, 31]. An option for achieving test isolation is to skip certain operations that hinder other clients or insert test data into, for example, a database. In cases where operations cannot be skipped, another option would be to restore the state of the service after testing is completed. The downside of this option is that other clients might have to be locked out temporary of using the service during the test, since they might encounter test data otherwise. In some cases, execution of other users will therefore have to be halted, which is often undesirable. Depending on the service, this can be solved by cloning the service and executing the tests on the cloned service. The downside to this solution is that in cases where a system consists of many services, it might be too expensive to clone them. Alternatively only the involved methods could be cloned and adjusted for testing purposes. In the worst case certain tests cannot be conducted while the service is operational. In all cases the service provider will have to identify the parts of the service that are sensitive to testing and determine the best approach to achieve test isolation for these parts.

2.2 Built-in Testing

Built-in testing increases the testability of a service by adding test cases and other test facilities to the service. This gives an insight about, for example, the state of the service and the means to control the state of the service. Reusability of the test facilities and test cases is also increased since they are included in the service, which saves development time and cost when dealing with highly dynamic services. Furthermore, BIT can be used to ensure test isolation by changing the behavior of methods during a test.

One way of achieving BIT is by adding testing and testable interfaces to the service, also called built-in contract testing by Gross [21]. With this approach, a service can test itself, its environment, and it can be tested by its environment. The testable interface extends the service so that another service can test it and the testing interface acquires the testable interface of another service to check if it meets its expectations. Test cases are built into the services at development time or even at deployment time. One major advantage of this approach is that it can be used during runtime, where services change all the time, to check if the environment of the service still performs as required by the service. A disadvantage of this approach is that private members of the base service are not visible to the testable, inherited service. In order to solve this, methods could be added to the original service, but this makes the service less modular.

A framework based on BIT to allow runtime integration testing of components, named ATLAS, has been developed by by González et al. [17]. Although it is based on components, it takes the constant joining and leaving of components into account, which is similar to how services behave. Their approach adds two interfaces to a component, one interface to add and remove tests, and one interface to provide testability features. However, they avoided runtime testing problems by duplicating the component that has to be tested, which is not always a good solution considering, for example, the extra cost and the complexity of the system.

Atkinson et al. show how verification effort in component-based software engineering can be reduced through the use of BIT [4, 31]. They introduce their own infrastructure, named MORABIT, which supports runtime testing through test isolation, resource monitoring, and test scheduling. Their approach helps pin-pointing the location of faults and automates testing. Test cases can be added or removed without modifying the source code. They also add two modes to a component, normal mode and maintenance. A disadvantage, which was seen before with other BIT implementations, is that private members remain hidden from the tester.

Bruel et al. use aspects to develop built-in tests for components [6]. Aspects can be used to separate concerns that cut across other concerns. An example of a cross-cutting concern is a variable that is locked before usage, preventing simultaneous adjustment of this variable. The code that locks the variable is removed and implemented as an aspect. In the same way, test cases can be implemented as aspects. In contrary to, for example, the built-in contract testing [21], private members of the component are visible with this approach. However, it has not been developed with runtime testing in mind. Their research shows that aspects offer great testability of the component, and modularity and evolvability are less difficult to achieve.

2.3 Aspect-Oriented Testing

Often certain design decisions cross-cut through the basic functionality of a system. Kiczales et al. developed aspect-oriented programming (AOP) to solve the problem of cross-cutting concerns by capturing them in so called aspects, increasing modularity [25].

An AOP extension for the programming language Java is AspectJ [11, 24]. Aspects in AspectJ are implemented by defining pointcuts, advice code and normal Java declarations. A pointcut is a set of join points, which are well-defined points in the execution of a program. Advice code is code that is run when a pointcut has been reached. There are three different advices in AspectJ: *before*, *after* and *around*, which determine when the advice code is executed. Furthermore, new methods and variables can also be inserted into the class by using aspects.

Let us explain the use of aspects with the example shown in listing A.1 in appendix A. The method `calculate` adds two given numbers and prints the result. When we want to print a message before and after execution of this method, we will first have to define a pointcut that triggers when the method is executed. The before and after advice will then use this pointcut to execute their code. The complete aspect can be found in listing A.2. The pointcut contains a single join point, namely the execution of the method `calculate`.

If we want to change the actual behavior of the method, then we will have to use the around advice. Listing A.3 gives an example where the value of both arguments of method `calculate` are increased by 1. The pointcut also had to be changed to be able to access the arguments of the method. Executing this piece of code will print the number 5 instead of 3. AspectJ offers many more different types of join points and other operations which can be used to create suitable aspects, all of which are listed in the AspectJ quick reference [14].

The terminology used for the AspectJ example, such as pointcuts, join points and advice code, are standard. They are also used in other AOP extensions for other programming languages. The differences come from the different internal workings of the programming languages. Aspects are implemented in different ways and some AOP extensions provide extra features, like AspectLUA [27], which allows the programmer to modify pointcuts and advice code during runtime.

Test cases and test facilities also cross-cut through the basic functionality of a system and can be implemented as aspects. Stamey and Saunders explain how aspects can be used to create unit tests and how to trace failures back to faults [30]. The *before*, *after* and *around* advices are used to validate the test results and to print the location of the fault when a test case fails. Bruel et al. implement test cases as new methods instead, without the *before*, *after* and *around* advices [6]. They only use these advices to capture transitions, for example, to update the state of a stack whenever the number of items in the stack increases or decreases.

Not much literature is available on the subject of creating test cases with aspects. However, the research by Stamey and Saunders, and Bruel et al. provides enough information on how to use aspects to implement test cases. Every variable and method can be modified and new variables and methods can be inserted. The weaving of aspects with test facilities and test cases into a service to create a testable service, is done by the aspect compiler as can be seen in figure 2.3.

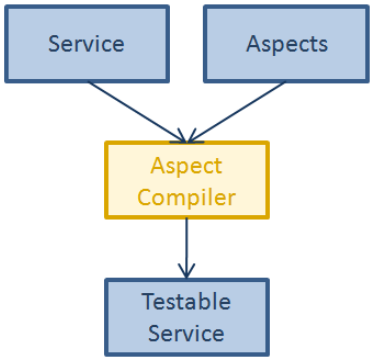


Figure 2.3: Weaving test code as aspects into a service to create a testable service

Chapter 3

Aspect-Oriented Built-in Testing for Services

In this chapter, the information and insights obtained in chapter 2, and experiments with services and test cases are used to create our approach. The aim of our approach is to equip services with built-in testing through the use of aspects. The fault taxonomy discussed in section 2.1.2 is used to guide us with the creation of the test cases. The approach by Bruel et al. [6] to achieve BIT with aspects, discussed in section 2.2, is used as a baseline for our approach, which extends the techniques proposed by Bruel.

3.1 Approach

As identified in the previous chapter, many BIT approaches exist, which all add testable and testing interfaces to the component or service. These are used to set the component or service into a certain state and to execute test cases on them. The main problem with most of these approaches is that private members are not visible unless the original service is modified. However, modifying the original service makes it less modular and harder to evolve.

Bruel et al. have acknowledged this fact and come up with aspects to develop built-in tests to maintain the modularity and evolvability of a component [6]. With aspects they insert new methods and variables to define the test cases and use the *before*, *after* and *around* advices to capture state transitions. These aspects are then compiled into the component to create a testable component. In contrast to components, their subject of study, services present different challenges as was discussed in section 2.1. Therefore, the approach by Bruel et al. has shortcomings if we want to apply it to services instead of components:

- The most important shortcoming is that the service would either be in nominal mode or in test mode when the same service is used for both testing and normal use. This could be solved by making a copy of the service for the purpose of testing, but this might be too expensive for the service provider. Instead, a mechanism is required to turn testing on and off depending on the client that is using the service. This is

especially important when testing is done during runtime as other clients should be able to use the service normally without any interference from testing clients.

- The second shortcoming of their approach is that since it is based on component-based systems, service specific test cases, according to the taxonomy discussed in section 2.1.2, have not been considered. Furthermore, the test cases are inserted as new methods. Since only the service its interface is accessible, these methods would also have to be added to the interface to be able to call them. Instead, test cases should be implemented by modifying the method that has to be tested by using *before*, *after* and *around* advices. In this case the original method can be called and no modification of the interface is required for each test case.

- The third shortcoming is that runtime testing has not been considered, which is important when dealing with services, since they can be updated at any given time. When a service has been updated, the associated clients will have to perform their integration tests. In order to ensure that the integration tests do not hinder nominal operations, test isolation is required. Different test isolation techniques have been discussed in section 2.1.4.

- The fourth shortcoming of their approach is that a test suite has not been included in the service. When the service or its environment are updated, test cases will have to be executed to validate that the service still performs correctly in its context.

- The final shortcoming is that Bruel et al. did not assess the overhead of their approach. Although a performance loss is expected, since extra code is added, it is essential to determine the extent of performance overhead incurred when applying BIT with aspects. Services will often have to respond within a certain amount of time, which means that BIT code should not cause too much of a performance loss. Furthermore, we will have to assess how much additional code is required for our approach to determine the development overhead.

Since it would be pretty hard to define our approach without having adequate experience with services and test cases, we implemented many services with test cases in Java to conduct experiments. Our approach is the result of these experiments, and in the next five sub-sections we will explain in more detail how the shortcomings of the approach by Bruel et al. have been addressed.

3.1.1 Mechanism to Enable and Disable Testing

Since the aspects are woven into the service, test mode is always active even for nominal operations. One way to switch test mode off is by removing the aspects after testing is finished. This can be done by removing the source file containing the aspects and by recompiling the service. However, most of the time testing will have to be done during runtime and the system cannot be taken offline to be tested. Furthermore, the clients and the service itself should be able to test whenever needed.

Keeping the aspects in place means that we have to ensure that clients can normally use the service while it is being tested by other clients. One option is to use a global variable on the service itself to toggle testing, but this will put the entire service either into nominal mode or test mode, which means that other clients cannot use it normally while tests are performed. Instead, a mechanism is needed to differentiate between the clients who use the service and the ones that test it. The idea behind this mechanism is to enable and disable pointcuts on demand, depending on who is using the service. Since pointcuts work by checking if a join point has been reached, we can add another condition to find out if testing has been enabled and apply the advices accordingly.

Now that we have described how pointcuts can be modified to disable and enable test code, we need a way to inform the service that testing should be enabled. Since services communicate with messages, we can tag these messages with testing information. The idea of tagging these messages originates from the same mechanism that is used to authenticate with services. Login names and passwords are tagged along with the message. The service then obtains this tagged login information and only responds if it is correct. In our case, the condition in the pointcut will check if the message is tagged with testing information to determine if testing should be enabled. Listing 3.1 shows the pseudocode for tagging and processing a message with testing information.

```
1  // client code: sending message
2  someMethod
3  {
4      discover service
5      bind service
6
7      tag message with testing information
8      invoke method on service
9  }
10
11 // service code: retrieving message
12 POINTCUT testMethod : testingEnabled() AND execution of method
13
14 boolean testingEnabled()
15 {
16     read message from client
17     IF message is tagged with testing information
18         RETURN true
19     ELSE
20         RETURN false
21 }
22
23 AROUND testMethod
24 {
25     setup test case
26     proceed with normal execution of method
27     IF test case does not meet expected result
28         THROW EXCEPTION
29 }
```

Listing 3.1: Pseudocode: Mechanism to enable and disable testing

The client tags the message with testing information and invokes a method on the service, while the service is aspected with a method, named `testingEnabled`, to process this

message. The *around* advice, where the test case is set up and executed, is only executed if `testingEnabled` returns true, i.e. the message is tagged with testing information, in the pointcut `testMethod`. In other words, the method invoked on line 8 is altered by the *around* advice on lines 23–29 when testing has been enabled.

3.1.2 Service Test Cases as Aspects

The fault taxonomy, discussed in section 2.1.2, is used as a guideline to help us with the creation of the test cases for services. For each type of fault we will show an example test case implemented with aspects.

Publishing Faults

An example of a publishing fault is a missing resource, such as a link to a database. The pseudocode in listing 3.2 illustrates a test case which throws an exception when the database is missing.

```
1 POINTCUT publishingTest : testingEnabled() && JOIN POINT connect to database
2
3 AFTER publishingTest
4 {
5     IF database connection returns NULL
6         THROW EXCEPTION
7 }
```

Listing 3.2: Pseudocode: Publishing test of a service

The test is started by connecting to the database after which the test case throws an exception if the connection to the database could not be established.

Discovery Faults

Discovery faults are faults that occur when, for example, a service cannot be found. The pseudocode for such a test case is described in listing 3.3.

```
1 POINTCUT discoverService : testingEnabled() && JOIN POINT obtain service ↩
      reference
2
3 AFTER discoverService
4 {
5     IF service not found
6         THROW EXCEPTION
7 }
```

Listing 3.3: Pseudocode: Discovery test of a service

After obtaining the reference to the service, the test case checks if the service was found and throws an exception if it was not found.

Binding Faults

When the required methods are missing on the acquired service, a binding failure occurs.

```
1  POINTCUT bindService : testingEnabled() && JOIN POINT bind service
2
3  AFTER bindService
4  {
5      FOR each required method defined in the service description
6          invoke method on service
7          IF method does not exist
8              THROW EXCEPTION
9  }
```

Listing 3.4: Pseudocode: Binding test of a service

Listing 3.4 illustrates how each required method on the acquired service is invoked to validate that it exists.

Composition Faults

A composition fault occurs when, for example, certain criteria is not met. A service might expect a response from another service within a certain amount of time. The pseudocode for such an integration test case can be found in listing 3.5.

```
1  POINTCUT compositionTest : testingEnabled() && JOIN POINT execution of ↩
       method
2
3  AROUND compositionTest
4  {
5      start timer
6      invoke method on service
7      stop timer
8      response time = start - stop
9      IF response time exceeds criteria
10         THROW EXCEPTION
11 }
```

Listing 3.5: Pseudocode: Composition test of a service

Before the method on the other service is invoked, a timer is started. When the service responded, the timer is stopped and the response time is calculated. An exception is thrown if the response time exceeds the allowed response time.

Execution Faults

Execution faults occur when a certain method does not perform as expected. Aspects can also be used for this purpose, for example, to compare the value of a variable before execution with the value of a variable after execution of a method. The unit test in listing 3.6 checks if an execution fault is present.

```
1  POINTCUT testVariable : testingEnabled() && JOIN POINT execution of method
2
3  // option 1
4  BEFORE testVariable
5  {
6      store dummy as temp
7  }
8
9  AFTER testVariable
10 {
11     IF dummy != temp + 1
12         THROW EXCEPTION
13 }
14
15 // option 2
16 AROUND testVariable
17 {
18     store dummy as temp
19     proceed with normal execution of method
20     IF dummy != temp + 1
21         THROW EXCEPTION
22 }
```

Listing 3.6: Pseudocode: Execution test for variable dummy

This unit test can either be done with the *before* and *after* advices, or the *around* advice. In both cases the value of the variable dummy is stored before the execution of a method and compared with the value after the execution to validate that it has increased by one.

A problem with comparing certain variables is that some variables represent shared resources. In other words, they are used by other clients during our tests and can make a test case fail. There are several solutions to this problem. The variable could be locked for the entire duration so that no other client can change it. The obvious disadvantage to this is that the other clients will have to wait, which might be a long time if the test takes a large amount of time. Another solution is to duplicate the code of the method in an *around* advice and use different variables for testing, which might not be an optimal solution since it might take up too many resources. Listing 3.7 shows how the method using variable dummy from listing 3.6 is locked.

```
1  POINTCUT testVariable : JOIN POINT execution of method
2
3  // option 1
4  BEFORE testVariable
5  {
6      lock
7      IF testingEnabled()
8          store dummy as temp
9  }
10
11 AFTER testVariable
12 {
13     IF testingEnabled()
14     {
15         IF dummy != temp + 1
16             THROW EXCEPTION
17     }
```

```
18      unlock
19  }
20
21  // option 2
22  AROUND testVariable
23  {
24      lock
25      IF testingEnabled()
26          store dummy as temp
27
28      proceed with normal execution of method
29
30      IF testingEnabled()
31      {
32          IF dummy != temp + 1
33              THROW EXCEPTION
34      }
35      unlock
36  }
```

Listing 3.7: Pseudocode: Execution test for variable `dummy` with locking

Two changes have to be implemented in order to lock the method. The first change is the actual lock and unlock mechanism at the start and end of the method execution. The second change is the movement of the `testingEnabled` condition to the advices themselves. This is required because clients who use the service without testing enabled should still hit the lock and unlock mechanisms for these to work properly.

3.1.3 Test Isolation

Test isolation is needed to ensure that other clients are not hindered while the service is being tested and to not cause any undesired effects to occur. For our approach we will achieve test isolation in different ways. In some cases it is possible to skip an entire operation completely, which can be done with the *around* advice, as can be seen in listing 3.8.

```
1  POINTCUT skipOperation : testingEnabled() && JOIN POINT some operation
2
3  AROUND skipOperation
4  {
5      empty
6  }
```

Listing 3.8: Pseudocode: Test isolation by skipping an operation

In other cases it is required to save and restore certain variables before and after the execution of an operation. The test case described in listing 3.7 would fail if the variables were restored before the test case was evaluated. In listing 3.9 we show how we have modified the test case to restore the values after the test case has been conducted.

```
1  POINTCUT testVariable : JOIN POINT execution of method
2
3  // option 1
4  BEFORE testVariable
```

```
5  {
6      lock
7      IF testingEnabled()
8          store dummy as temp
9  }
10
11 AFTER testVariable
12 {
13     IF testingEnabled()
14     {
15         IF dummy != temp + 1
16             THROW EXCEPTION
17         restore dummy with temp
18     }
19     unlock
20 }
21
22 // option 2
23 AROUND testVariable
24 {
25     lock
26     IF testingEnabled()
27         store dummy as temp
28
29     proceed with normal execution of method
30
31     IF testingEnabled()
32     {
33         IF dummy != temp + 1
34             THROW EXCEPTION
35         restore dummy with temp
36     }
37     unlock
38 }
```

Listing 3.9: Pseudocode: Test isolation by saving and restoring a variable

In case of a complex service-oriented system, test cases can cause a ripple effect through the entire system. Take for example a service that invokes an operation on another service which is connected to a database. When invoking that operation, the service also has to be aware of the fact that testing is enabled to avoid that the database is updated with test data. To this end, the same mechanism discussed earlier to enable and disable testing can be used.

```
1 POINTCUT bindService : testingEnabled() && JOIN POINT bind service
2
3 AFTER bindService
4 {
5     tag message with testing information
6 }
```

Listing 3.10: Pseudocode: Inform another service that testing is enabled

As shown in listing 3.10, the testing information is included after the service has been bound. Whenever a method is invoked on that specific service, it will be informed by the tagged message that testing should be enabled.

3.1.4 Invoking Unit and Integration Tests on the Service

A testing interface is required to execute a test suite on the service. This can be achieved by adding two methods to the interface of the service with the use of an aspect. One method is inserted to run unit tests on the service, and another method is inserted to run integration tests on the service. The actual implementation of these methods deals with setting up the test environment, invoking methods and evaluating the test results. The pseudocode for this test suite can be found in listing 3.11.

```
1  add runUnitTests() to interface of service
2  add runIntegrationTests() to interface of service
3
4  boolean runUnitTests()
5  {
6      invoke methods on self
7
8      RETURN unit tests passed or failed
9  }
10
11 boolean runIntegrationTests()
12 {
13     discover service
14     bind service
15
16     setup test environment
17     invoke methods on service
18
19     RETURN integration tests passed or failed
20 }
```

Listing 3.11: Pseudocode: Invoking unit and integration tests on the service

3.1.5 Overhead

The performance of a service depends on the computer hardware and software it is deployed on and the network it is connected to. When testing is considered, and services are equipped with test facilities and test cases, a performance drop can be expected. After all, extra lines of code are added and executed in addition to the original source code. The performance is pretty hard to compare since we would have to compare it against another test approach. Since there are many test approaches, it does not make much sense to pick a random one just for the sake of performance testing. Instead, we will compare the performance of the original service without aspects to the performance of the service with aspects without testing enabled. In other words, both services are used normally by the clients. The time can be measured by subtracting the time before execution from the time obtained after execution. Since doing this only once would not give accurate results, we will repeat this process several times and take the average time. We can then compare these values to determine the potential performance loss. Listing 3.12 shows the pseudocode for such a performance test.

```
1  double performanceTest
2  {
3      discover service
4      bind service
5
6      start timer
7      FOR 1 to 1000
8          invoke method on service
9      stop timer
10     elapsed time = start - stop
11
12     RETURN elapsed time / 1000
13 }
```

Listing 3.12: Pseudocode: Measuring performance of the service

The development overhead is determined by the extra lines of code that are required to achieve BIT with aspects. Since services are deployed by the service provider, the service provider will have to equip their services with test facilities and test cases.

Figure 3.1 depicts the class diagram of a service aspected with BIT technology.

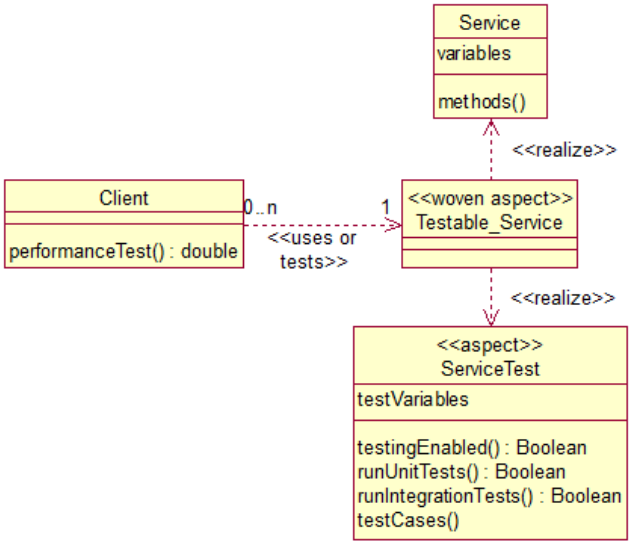


Figure 3.1: Class diagram: Aspect-oriented built-in testing for services

With aspects we added new methods to check if testing is enabled, and to run unit and integration tests on the service. Furthermore, the method executions are modified with the *before*, *after* and *around* advices to implement the test cases. Clients, which can be another service or a client application, bind these testable services and can either invoke tests on them or use them normally.

Chapter 4

Feasibility Study

This chapter describes how our approach, detailed in chapter 3, has been applied to a small service-oriented system. The chapter starts with the goal and design of our feasibility study. After that we will show the results of the implementation of our approach. The chapter concludes with a discussion of these results.

4.1 Goal

The goal of our feasibility study is to provide a proof of concept of our approach for achieving BIT for services with the use of aspects, and to help answer the sub research questions listed in section 1.1. Furthermore, we aim to answer some specific questions that are related to the actual test setup:

- Which part of the message can we tag with testing information to let the service know that testing should be enabled?
- If there is a performance loss, how is this caused?
- What is the complexity of the instrumentation of the service with aspects and can it be automated?

4.2 Design

Web services are the most extreme case of services, since they are highly dynamic, and they are therefore suitable for our feasibility study. A well-known API for developing web services is the Java API for XML Web Services (JAX-WS) [28]. Since JAX-WS is based on Java and uses XML for the communication between clients and services, JAX-WS services have increased platform independence, which is the main reason why our system is based on JAX-WS. Furthermore, AspectJ is well-known and can be used to equip our system with test facilities and test cases through the use of aspects. Our web services are deployed on a Apache Tomcat application server [15].

JAX-WS has been designed to simplify and automate the creation and use of web services. The service developer starts out with the basic implementation of the web service, including the use of annotations to let the compiler know which methods should be exposed to the clients. The developer will then use a tool, supplied by Java, to generate the necessary ties, and stubs if the service is using another service. When these stubs and ties are generated, the service can be deployed. Before explaining JAX-WS in more detail, we will provide an illustration of the architecture of JAX-WS in figure 4.1.



Figure 4.1: JAX-WS Architecture

The client and service use the Simple Object Access Protocol (SOAP) to send XML messages over HTTP. These messages are generated and translated automatically by the JAX-WS Runtime. The stubs and ties are generated by a tool, supplied by Java, to map ordinary Java method invocations to XML messages and vice versa. Once deployed, the web service is described by a Web Services Description Language (WSDL) file. This file is automatically generated by the application server and obtainable by a client. When developing the client, the WSDL file is imported with a supplied tool and the stubs are then automatically generated. These stubs have to be used by the client to bind the web service and to invoke operations on it. An advantage of JAX-WS is that since it uses SOAP for the communication, clients can be written in other languages than Java. For example, gSOAP is a toolkit that can be used to map the XML messages to c/c++ code [23].

4.2.1 Test Setup

Unfortunately there were not any large JAX-WS service-oriented systems available for us to experiment on. Therefore, we had to create our own service-oriented system, which resulted in a small test setup that simulates a basic library where people can get a membership, loan and return books. Furthermore, the test setup provides the necessary tools for the employees to manage their library. The test setup consists of four JAX-WS services, two clients and a database. The JAX-WS services are: `MemberManager`, `BookManager`, `Logger` and `Statistics`. These services are only accessible by the clients through their interfaces, which describe the exposed methods. Furthermore, there is a database with four tables that is used by the services. A detailed schema of this database can be found in listing B.1 in appendix B. Lastly, there are two clients, a `library` which acquires the `MemberManager` and `BookManager` services, and a `visualizer` which acquires the `Statistics` service. Figure 4.2 depicts the test setup.

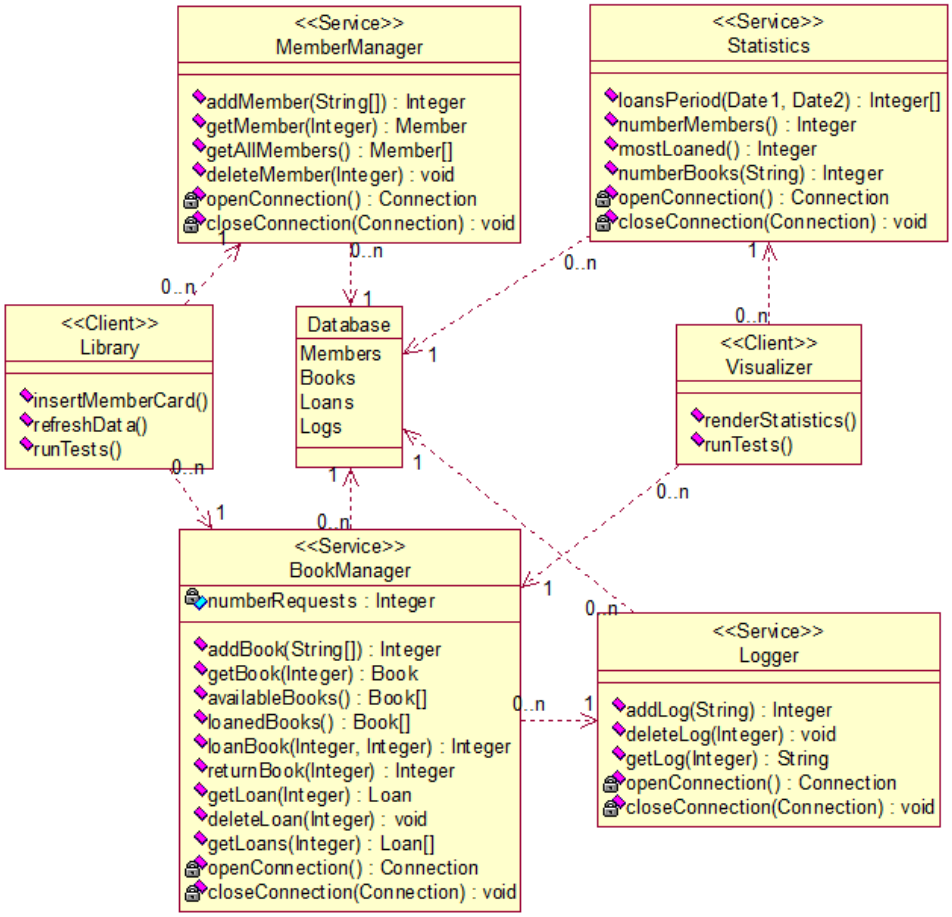


Figure 4.2: Class diagram: Test setup consisting of four JAX-WS services, two clients and a database

For the purpose of illustrating how our approach has been applied, we will only take a part of the test setup. This part consists of the `BookManager` service, the `Logger` service, required by the `BookManager` service, a custom client for the purpose of invoking tests, and the database. The `BookManager` service will be used to give an idea about the implementation of a JAX-WS service. Its interface is displayed in listing 4.1. It contains the methods that are exposed to the clients.

```
1  package bookManagerPackage;
2
3  import javax.jws.WebMethod;
4  import javax.jws.WebService;
5
6  import bookManagerPackage.impl.Book;
7  import bookManagerPackage.impl.Loan;
8
9  @WebService
10 public interface IBookManager
```

```
11 {
12     @WebMethod int addBook(String author, String title, String genre);
13     @WebMethod Book getBook(int bookID);
14     @WebMethod Book[] availableBooks();
15     @WebMethod Book[] loanedBooks();
16     @WebMethod int loanBook(int memberID, int bookID);
17     @WebMethod int returnBook(int bookID);
18     @WebMethod Loan getLoan(int loanID);
19     @WebMethod Loan[] getLoans(int memberID);
20 }
```

Listing 4.1: Interface of JAX-WS service `BookManager`

The `@WebService` annotation is used to mark the interface as an endpoint interface, i.e. an interface that is used by the clients. The `@WebMethod` annotation is used to expose a method to the clients who use the service. In listing 4.2 the actual implementation of `BookManager` is given. Since listing all the methods would take up several pages, we will only show how method `loanBook` is implemented.

```
1  package bookManagerPackage.impl;
2
3  import javax.jws.WebService;
4  import java.sql.*;
5  import java.util.ArrayList;
6  import java.util.Date;
7
8  import bookManagerPackage.IBookManager;
9  import loggerpackage.impl.*;
10
11 @WebService(endpointInterface = "bookManagerPackage.IBookManager")
12 public class BookManager implements IBookManager
13 {
14     private int numberRequests = 0;
15
16     ...
17
18     @Override
19     public int loanBook(int memberID, int bookID)
20     {
21         numberRequests++;
22         Connection connection = openConnection();
23         int loanID = -1;
24
25         Date dateExpires = new Date();
26         Date now = new Date();
27         dateExpires.setTime(dateExpires.getTime() + 20*24*60*60*1000);
28
29         try
30         {
31             PreparedStatement insertLoan = connection.prepareStatement("↩
                   INSERT INTO loans (memberID, bookID, dateAdded, dateExpires)↩
                    VALUES(?,?,?,?)", Statement.RETURN_GENERATED_KEYS);
32
33             insertLoan.setInt(1, memberID);
34             insertLoan.setInt(2, bookID);
35             insertLoan.setDate(3, new java.sql.Date(now.getTime()));
36             insertLoan.setDate(4, new java.sql.Date(dateExpires.getTime()));
37             insertLoan.executeUpdate();
38
```

```
39              ResultSet result = insertLoan.getGeneratedKeys();
40              while (result.next())
41                  loanID = result.getInt(1);
42
43              Statement updateBook = connection.createStatement();
44              updateBook.executeUpdate("UPDATE books SET available='0' WHERE ↩
                    bookID='" + bookID + "'");
45
46              LoggerService loggerService = new LoggerService();
47              ILogger logger = loggerService.getLoggerPort();
48              logger.addLog("Member: " + memberID + ", LOANED Book: " + bookID↩
                     + "");
49
50              insertLoan.close();
51              updateBook.close();
52              result.close();
53          }
54          catch (Exception e)
55          {
56              System.out.println("Error: " + e);
57          }
58          closeConnection(connection);
59
60          return loanID;
61      }
62
63      ...
64  }
```

Listing 4.2: Implementation of JAX-WS service `BookManager`

On line 9 the required stubs are included to be able to access service `Logger`. When method `loanBook` is called by a client, the number of requests to this service is increased by 1, a connection to the database is made, the loan is inserted into the database and the book is updated, the service `Logger` is discovered and bound (lines 46 and 47), the action is logged, and the id of the inserted loan is returned. A client using this service would discover, bind and invoke operations on it in the same way the `Logger` service is acquired and used on lines 46–48.

4.3 Results

In this section the results from applying our approach, detailed in chapter 3, to the service-oriented system described in listing 4.2 are presented. For illustration purposes we will use the method `loanBook`, shown in listing 4.2, as an example throughout this section.

We validated all test cases and test facilities by injecting faults into the service-oriented system, such as the removal of service `Logger` and the database, the modification of `Logger` by renaming method `addLog`, and the simulation of a long response time by adding a delay to the `addLog` method.

For each sub research question we defined in section 1.1, a sub-section is used to explain how we implemented our approach.

4.3.1 Mechanism to Enable and Disable Testing

Since aspects are woven into the service, test mode is always active even for nominal operations. In order to differentiate between the clients who use the service normally and the ones that test it, clients that want to test the service will have to tag their messages with testing information. This testing information is then processed by the service and the test cases are activated only for those clients.

One of the feasibility study questions was to determine which part of the message could be tagged with testing information. JAX-WS provides a way to manipulate the exchange of information between clients and services by modifying the HTTP request headers with metadata. In our case the testing information is described as metadata and added to the headers. Listing 4.3 shows the modification of the HTTP request headers after service `BookManager` has been bound.

```
1 Map<String, Object> rctx = ((BindingProvider)bookMngr).getRequestContext();
2 Map<String, List<String>> headers = new HashMap<String, List<String>>();
3 headers.put("Testing", Collections.singletonList("true"));
4 rctx.put(MessageContext.HTTP_REQUEST_HEADERS, headers);
```

Listing 4.3: Client including testing information by modifying the HTTP request headers

First the request context of the service that has to be tested, in this case `BookManager`, is obtained which allows the metadata to be configured. The testing information is stored in `HashMap`, which is then inserted into the HTTP request headers. Whenever a method invocation occurs on this specific service, the testing information is sent along with it.

The service will have to process this tagged message to determine if testing should be enabled. This is done by adding the `testingEnabled` method to the service with an aspect, as illustrated in listing 4.4.

```
1  static WebServiceContext BookManager.wsctx;
2  declare @field: WebServiceContext BookManager.wsctx : @Resource;
3
4  public static boolean BookManager.testingEnabled()
5  {
6      MessageContext mctx = BookManager.wsctx.getMessageContext();
7      Map<String, List<String>> headers = (Map<String, List<String>>) mctx.get↩
           (MessageContext.HTTP_REQUEST_HEADERS);
8      List testList = (List) headers.get("Testing");
9
10     String testing = "false";
11
12     if (testList!=null)
13         testing = testList.get(0).toString();
14
15     if (testing.equals("true"))
16         return true;
17     else
18         return false;
19 }
```

Listing 4.4: Service enabling test mode for clients tagging their messages with testing information

A resource to store the context of the message, i.e. the client that is using the service, is added to the service on lines 1 and 2. The HTTP request headers containing the testing information is retrieved at lines 6–13 of the code. Testing is enabled when the value of the variable equals `true`. Testing can be disabled by the client at any given time by removing the metadata from the HTTP request headers.

The activation of the test cases is done by modifying the pointcuts with a call to the `testingEnabled` method.

```
1 pointcut testLoanBook() : if(BookManager.testingEnabled()) && execution(* ↩
      BookManager.loanBook(..));
```

Listing 4.5: Condition `testingEnabled` added to a pointcut

As an example in listing 4.5, the test case that is associated with the execution of method `loanBook` from service `BookManager` will only be run if `testingEnabled` returns true.

In order to validate that this approach works when multiple clients use a service at the same time, a small scenario was created where two clients call method `loanBook`. The first client will test the method and the second client will normally use this method while the first client is testing it. In order to mimic this simultaneous use, a delay of ten seconds has been built into the method as an aspect which is only activated when the method is being tested by the first client. Furthermore, after this delay, the aspect checks if testing is still enabled and prints a certain message to let the first client know if it is. The code of this aspect is shown in listing 4.6.

```
1  pointcut testLoanBook() : if(BookManager.testingEnabled()) && execution(* ↩
       BookManager.loanBook(..));
2
3  Object around() : testLoanBook()
4  {
5      long t0, t1;
6      t0 =  System.currentTimeMillis();
7
8      do
9      {
10         t1 = System.currentTimeMillis();
11     }
12     while ((t1 - t0) < (10000));
13
14     if (BookManager.testingEnabled())
15         System.out.println("Testing is still enabled!");
16
17     return proceed();
18 }
```

Listing 4.6: Aspect for delaying the execution of `loanBook`

The result of this scenario is that testing is indeed still enabled after the second client used the service. The same applies when the client is testing a service that is coupled to other services. All these services will be in test mode for the first client only and the second client can use all the services without noticing that testing is enabled. In other words, the modification of the HTTP request headers can reliably be used for our purpose.

4.3.2 Service Test Cases as Aspects

The fault taxonomy described in section 2.1.2 is used to guide us with the creation of the service test cases. Example test cases are given for each type of fault.

Publishing Faults

Publishing faults are mostly detected by the JAX-WS runtime, since it generates and validates the service description files automatically. This means that, for example, a service cannot expose more methods than the service description lists. A missing resource, such as the database required by `BookManager`, can be detected with the test case in listing 4.7.

```
1  pointcut testDbConn() : if(BookManager.testingEnabled())
2                          && execution(* BookManager.openConnection(..));
3
4  after() returning(Connection connection) : testDbConn()
5  {
6      try
7      {
8          Assert.assertNotNull(connection);
9      }
10     catch (Throwable e)
11     {
12         integrationTestsPassed = false;
13         System.out.println("Error at: "+ thisJoinPoint.getSourceLocation());
14         System.out.println(e);
15     }
16 }
```

Listing 4.7: Integration test case: Availability of the database required by `BookManager`

By invoking the `openConnection` method on `BookManager`, the integration test case checks if the connection was successfull. If a connection to the database cannot be made, an exception is thrown.

Discovery and Binding Faults

The discovery and binding tests are invoked by the sequence of methods displayed in listing 4.8.

```
1  LoggerService loggerService = new LoggerService();
2  ILogger logger = loggerService.getLoggerPort();
3
4  logger.addLog("Integration Test");
```

Listing 4.8: Discovery and binding of service `Logger`

First the `Logger` service is discovered, then it is bound and finally a method is called on the service. If the discovery test fails, then there is no point trying to bind it and invoke methods on it. The discovery test case can be found in listing 4.9.

```
1  pointcut discoverLogger() : if(BookManager.testingEnabled())
2                              && call(LoggerService.new(..));
3
4  private static boolean discoveryError = false;
5
6  Object around() : discoverLogger()
7  {
8      discoveryError = false;
9
10     try
11     {
12         return proceed();
13     }
14     catch (Throwable e)
15     {
16         discoveryError = true;
17         integrationTestsPassed = false;
18         System.out.println("Discovering Logger has failed at: " + ↩
               thisJoinPoint.getSourceLocation());
19         System.out.println(e);
20         return null;
21     }
22 }
```

Listing 4.9: Integration test case: Discovery of service `Logger`

This integration test case is triggered when a service is being discovered and throws an exception if it is not found. The next integration test case, shown in listing 4.10, binds the service and checks if `addLog` exists.

```
1  pointcut bindLogger() : if(BookManager.testingEnabled())
2                          && call(* getLoggerPort(..));
3  pointcut testAddLog(String s) : if(BookManager.testingEnabled())
4                                  && call(* addLog(..)) && args(s);
5
6  Object around() : bindLogger()
7  {
8      try
9      {
10         if (!discoveryError)
11             return proceed();
12         else
13             return null;
14     }
15     catch (Throwable e)
16     {
17         integrationTestsPassed = false;
18         System.out.println("Binding Logger has failed at: " + thisJoinPoint.↩
               getSourceLocation());
19         System.out.println(e);
20         return null;
21     }
22 }
23
24 Object around(String s) : testAddLog(s)
25 {
26     try
27     {
28         return proceed(s);
```

```
29      }
30      catch (Throwable e)
31      {
32          integrationTestsPassed = false;
33          System.out.println("Method addLog does not exist on service Logger: ↩
                " + thisJoinPoint.getSourceLocation());
34          System.out.println(e);
35          return "test failed";
36      }
37  }
```

Listing 4.10: Integration test case: Binding of service `Logger`

The first aspect on lines 6–22 is triggered when trying to bind the service, as described by the pointcut on line 2. If there were no discovery errors, then the method `getLoggerPort` is executed to bind the service. Normally we would have to check ourselves if a method exists and throw an error if does not exist, but the JAX-WS runtime does this instead for us by comparing the generated stubs to the methods defined in the WSDL file. If the JAX-WS runtime would not have checked this for us, then we would have to use the second aspect on lines 24–37. The aspect tries to execute the method and throws an error if there is a problem.

Composition Faults

A composition fault occurs when `Logger` does not respond within a certain amount as required by `BookManager`. The integration test case in listing 4.11 shows how the timeout is measured. Before the execution of `addLog` a timer is started and after execution the timer is stopped after which the response time is calculated. When the response time exceeds five seconds, an error is printed.

```
1   pointcut testAddLog(String s) : if(BookManager.testingEnabled())
2                                   && call(* addLog(..)) && args(s);
3
4   long t0, t1, responseTime;
5
6   before(String s) : testAddLog(s)
7   {
8       t0 = System.currentTimeMillis();
9   }
10
11  after(String s) : testAddLog(s)
12  {
13      t1 = System.currentTimeMillis();
14      responseTime = t1 - t0;
15
16      if (responseTime > 5000)
17      {
18          integrationTestsPassed = false;
19          System.out.println("Logger did not respond within 5 seconds: " + ↩
                thisJoinPoint.getSourceLocation());
20      }
21  }
```

Listing 4.11: Integration test case: Response time of `Logger` should not exceed 5 seconds

Execution Faults

Execution faults occur when the service does not produce the expected result. A typical unit test case would be to check whether the variable `numberRequests` has been increased by 1 and that the loan has been correctly inserted into the database when `loanBook` has been executed. Note that the variable `numberRequests` is private, but since we can define our aspect as `privileged` we can access this variable without a `get` method. The test case is shown in listing 4.12.

```
1  pointcut testLoanBook(int m, int b) :
2                               if(BookManager.testingEnabled())
3                               && execution(* BookManager.loanBook(..))
4                               && args(m, b);
5
6  before(int memberID, int bookID) : testLoanBook(memberID, bookID)
7  {
8      old_numberRequests = ((BookManager) thisJoinPoint.getThis()).↩
           numberRequests;
9  }
10
11 after(int memberID, int bookID) returning(int loanID) : testLoanBook(↩
       memberID, bookID)
12 {
13     new_numberRequests = ((BookManager) thisJoinPoint.getThis()).↩
           numberRequests;
14     Loan loan = ((BookManager) thisJoinPoint.getThis()).getLoan(loanID);
15
16     try
17     {
18         Assert.assertEquals(old_numberRequests+1, new_numberRequests);
19         Assert.assertEquals(memberID, loan.memberID);
20         Assert.assertEquals(bookID, loan.bookID);
21     }
22     catch (Throwable e)
23     {
24         unitTestsPassed = false;
25         System.out.println("Error at: "+ thisJoinPoint.getSourceLocation());
26         System.out.println(e);
27     }
28 }
```

Listing 4.12: Unit test case: Method `loanBook`

The *before* advice on lines 6–9 stores the value of `numberRequests` before the execution of `loanBook`. The *after* advice on lines 11–28 stores the value of `numberRequests` after execution of `loanBook` and checks if it has been increased by 1. Furthermore, the inserted loan is retrieved from the database and its values are compared with the expected values.

A situation that can occur with the previous test case is that another client also increments `numberRequests` while the service is being tested. This leads to an error being thrown by the test case. In order to solve this, we can lock out other clients from using the methods that increment `numberRequests` as listing 4.13 shows. This will, however, hinder other clients since they will have to wait until the test is complete.

```
1  private Lock lock = new ReentrantLock();
2
3  pointcut testLoanBook(int m, int b) :
4                               execution(* BookManager.loanBook(..))
5                               && args(m, b);
6
7  before(int memberID, int bookID) : testLoanBook(memberID, bookID)
8  {
9      lock.lock();
10
11     if (BookManager.testingEnabled())
12         old_numberRequests = ((BookManager) thisJoinPoint.getThis()).↩
               numberRequests;
13 }
14
15 after(int memberID, int bookID) returning(int loanID) : testLoanBook(↩
       memberID, bookID)
16 {
17     if (BookManager.testingEnabled())
18     {
19         new_numberRequests = ((BookManager) thisJoinPoint.getThis()).↩
               numberRequests;
20
21         try
22         {
23             Assert.assertEquals(old_numberRequests+1, new_numberRequests);
24             Assert.assertEquals(memberID, loan.memberID);
25             Assert.assertEquals(bookID, loan.bookID);
26       }
27       catch (Throwable e)
28       {
29         unitTestsPassed = false;
30         System.out.println("Error at: "+ thisJoinPoint.getSourceLocation());
31         System.out.println(e);
32       }
33     }
34     lock.unlock();
35 }
```

Listing 4.13: Unit test case: Method `loanBook` with locking

Before the method is executed it is locked, and after execution it is unlocked. Furthermore, we have moved the `testingEnabled` check to the advices, because clients who do not test the service should still hit the locking mechanism for it to work.

The control flow of a method execution can also be tested with aspects. In the case of method `loanBook` there should be two database updates before a call is made to method `addLog` on service Logger. Listing 4.14 illustrates how the test case from the previous listing is extended with the means to record the control flow and to test if it is correct.

```
1  private String[] loanBookCFlow = new String[3];
2  private int loanBookCFlowCounter = 0;
3
4  pointcut flowLoanBook() : if(BookManager.testingEnabled())
5                            && cflow(execution( * BookManager.loanBook(..)))
6                            && withincode(* BookManager.loanBook(..));
7
8  before() : flowLoanBook()
9  {
10     if (thisJoinPoint.toString().equals("call(int java.sql.PreparedStatement↩
           .executeUpdate())"))
11     {
12         loanBookCFlow[loanBookCFlowCounter] = "executeUpdate()";
13         loanBookCFlowCounter++;
14     }
15     if (thisJoinPoint.toString().equals("call(int java.sql.Statement.↩
           executeUpdate(String))"))
16     {
17         loanBookCFlow[loanBookCFlowCounter] = "executeUpdate(String)";
18         loanBookCFlowCounter++;
19     }
20     if (thisJoinPoint.toString().equals("call(int loggerpackage.impl.ILogger↩
           .addLog(String))"))
21     {
22         loanBookCFlow[loanBookCFlowCounter] = "addLog(String)";
23         loanBookCFlowCounter++;
24     }
25 }
26
27 before(int memberID, int bookID) : testLoanBook(memberID, bookID)
28 {
29     lock.lock();
30
31     if (BookManager.testingEnabled())
32     {
33         loanBookCFlowCounter = 0;
34         loanBookCFlow = new String[3];
35
36         old_numberRequests = ((BookManager) thisJoinPoint.getThis()).↩
               numberRequests;
37     }
38 }
39
40 after(int memberID, int bookID) returning(int loanID) : testLoanBook(↩
       memberID, bookID)
41 {
42     if (BookManager.testingEnabled())
43     {
44         new_numberRequests = ((BookManager) thisJoinPoint.getThis()).↩
               numberRequests;
45         Loan loan = ((BookManager) thisJoinPoint.getThis()).getLoan(loanID);
46
47         try
48         {
49             Assert.assertEquals(old_numberRequests+1, new_numberRequests);
50             Assert.assertEquals(memberID, loan.memberID);
51             Assert.assertEquals(bookID, loan.bookID);
52
53             Assert.assertEquals("executeUpdate()", loanBookCFlow[0]);
54             Assert.assertEquals("executeUpdate(String)", loanBookCFlow[1]);
55             Assert.assertEquals("addLog(String)", loanBookCFlow[2]);
56         }
```

```
57          catch (Throwable e)
58          {
59              unitTestsPassed = false;
60              System.out.println("Error at: " + thisJoinPoint.↩
                    getSourceLocation());
61              System.out.println(e);
62          }
63      }
64      lock.unlock();
65  }
```

Listing 4.14: Unit test case: Control flow of method `loanBook`

On lines 4–6 the pointcut is declared to intercept the control flow of method `loanBook`. The recording of the control flow is achieved with the *before* advice defined on lines 8–25. The remainder of the advices have been updated with the means to reset the test variables on lines 33–34, and the actual test case on lines 53–55.

Another test case, which is an integration test, would be that the value returned by method `addLog` from service `Logger` has a certain value. This is achieved with the aspect in listing 4.15.

```
1  pointcut testAddLog(String s) : if(BookManager.testingEnabled())
2                                  && call(* addLog(..)) && args(s);
3
4  after(String entry) returning(int response) : testAddLog(entry)
5  {
6      try
7      {
8          Assert.assertNotSame(-1, response);
9      }
10     catch (Throwable e)
11     {
12         integrationTestsPassed = false;
13         System.out.println("Error at: "+ thisJoinPoint.getSourceLocation());
14         System.out.println(e);
15     }
16 }
```

Listing 4.15: Integration test case: Expected response from `Logger`

In this test case it is validated that the log has been correctly added to the database. If `addLog` returns -1, an error occurred at the `Logger` service and an exception is thrown.

4.3.3 Test Isolation

Test isolation is needed to ensure that other clients are not effected by tests while the service is being tested, and to not cause any undesired effects to occur. Two approaches to achieve test isolation have been implemented for method `loanBook`. Execution of this method increases variable `numberRequests` from `BookManager` by 1, update the database and invoke method `addLog` on service `Logger` as depicted in listing 4.2.

The first approach is to skip the actual incrementing of the variable, database updates and the method invocation with the around advice as can be seen in listing 4.16.

```
1  pointcut isolateNumberRequests(int i) :
2                            if(BookManager.testingEnabled())
3                            && set(int BookManager.numberRequests) && args(i);
4                            && withincode(* BookManager.loanBook(..));
5
6  pointcut isolateDatabaseUpdate(): if(BookManager.testingEnabled())
7                                    && call(* executeUpdate(..))
8                                    && withincode(* BookManager.loanBook(..));
9
10 pointcut isolateAddLog() : if(BookManager.testingEnabled())
11                            && call(* addLog(..))
12                            && withincode(* BookManager.loanBook(..));
13
14 Object around(int i) : isolateNumberRequests(i)
15 {
16     return null;
17 }
18
19 Object around() : isolateDatabaseUpdate()
20 {
21     return null;
22 }
23
24 Object around() : isolateAddLog()
25 {
26     return null;
27 }
```

Listing 4.16: Test isolation by skipping variable assignment and database update

The pointcut on lines 1–4 is triggered when `numberRequests` is set with value `i`. The around advice is used to skip assignment of `numberRequests` which results in the variable not being changed. The same mechanic is applied for the other two cases of test isolation.

Skipping, for example, the variable assignment can have an effect on other test cases. The test case shown in listing 4.13 that checks whether the variable `numberRequests` has been increased by 1, fails to perform correctly since the value is restored before a comparison could be made. Instead, the variable should then be restored after the execution of the test case. However, first we will have to solve an issue that occurs when simply adding the test isolation code to the test case. When a client wants to do an integration test on method `loanBook`, we only want to ensure test isolation and not run the actual test case. Therefore, we have to differentiate between clients who test the service itself and clients who invoke integration tests on the service to check if the service performs as they require. In other words, the clients will have to tag their messages with different testing information, depending if either test isolation should be ensured or if the test cases on the service itself should be executed while test isolation is still applied. Listing 4.17 illustrates how we have modified the test case from listing 4.14 to achieve test isolation, depending on the tagged testing information.

```
1  before() : testLoanBook()
2  {
3      lock.lock();
4
5      if (BookManager.testingEnabled() >= 1)
```

```
6     {
7         loanBookCFlowCounter = 0;
8         loanBookCFlow = new String[3];
9
10        old_numberRequests = ((BookManager) thisJoinPoint.getThis()).↩
              numberRequests;
11    }
12 }
13
14 after(int memberID, int bookID) returning(int loanID) : testLoanBook(↩
       memberID, bookID)
15 {
16    if (BookManager.testingEnabled() >= 1)
17    {
18        new_numberRequests = ((BookManager) thisJoinPoint.getThis()).↩
              numberRequests;
19        Loan loan = ((BookManager) thisJoinPoint.getThis()).getLoan(loanID);
20
21        if (BookManager.testingEnabled() == 1)
22        {
23          try
24          {
25            Assert.assertEquals(old_numberRequests+1, new_numberRequests);
26            Assert.assertEquals(memberID, loan.memberID);
27            Assert.assertEquals(bookID, loan.bookID);
28
29            Assert.assertEquals("executeUpdate()", loanBookCFlow[0]);
30            Assert.assertEquals("executeUpdate(String)", loanBookCFlow[1]);
31            Assert.assertEquals("addLog(String)", loanBookCFlow[2]);
32          }
33          catch (Throwable e)
34          {
35            unitTestsPassed = false;
36            System.out.println("Error at: "+ thisJoinPoint.getSourceLocation↩
                  ());
37            System.out.println(e);
38          }
39        }
40        ((BookManager) thisJoinPoint.getThis()).numberRequests = ↩
              old_numberRequests;
41        ((BookManager) thisJoinPoint.getThis()).returnBook(loan.bookID);
42        ((BookManager) thisJoinPoint.getThis()).deleteLoan(loanID);
43    }
44    lock.unlock();
45 }
```

Listing 4.17: Test isolation by restoring the variable and database

Lines 40–42 contain the code for restoring the value of variable `numberRequests`, the restoration of the state of the book and the deletion of the inserted loan. Note that we have not reversed the invocation of method `addLog` on service `Logger`. We could have modified the test case for `addLog`, displayed in listing 4.11, to delete the inserted log, but instead we will show later on how this can be done by the `Logger` service. Furtheremore, the test cases on lines 25–31 are not executed when a client wants to do integration tests with the service. Lastly, we have modified the `testingEnabled` method to return an integer, based on the tagged test information. When the method returns *two*, only test isolation is active, and when it returns *one*, both the test cases and test isolation are active.

When integration tests have to be run by the service it might also be needed to ensure

test isolation. Let us take, for example, the case where method `addLog` is invoked on service `Logger` in listing 4.2 on line 48. If we want to inform service `Logger` that we are testing, then we can simply adjust the HTTP request headers as we did before in listing 4.3. Listing 4.18 shows how we change these headers after we obtained the reference to `Logger` in listing 4.2 on line 47.

```
1  pointcut bindLogger() : if(BookManager.testingEnabled() >= 1
2                          && call(* getLoggerPort(..))
3                          && withincode(* BookManager.loanBook(..));
4
5  after() returning(ILogger logger) : bindLogger()
6  {
7      Map<String, Object> rctx= ((BindingProvider)logger).getRequestContext();
8      Map<String, List<String>> headers = new HashMap<String, List<String>>();
9      headers.put("Testing", Collections.singletonList("client"));
10     rctx.put(MessageContext.HTTP_REQUEST_HEADERS, headers);
11 }
```

Listing 4.18: Test isolation for service `Logger`

Since service `Logger` updates the database with the entry that has been sent with method `addLog`, we can either choose to skip that operation entirely or remove the data from the database after execution. Since the test case in listing 4.15 checks the response, we will delete the added log entry after returning the id. Listing 4.19 shows how the `Logger` service ensures test isolation.

```
1  pointcut isolateAddLog() : if(Logger.testingEnabled() >= 1
2                             && execution(* Logger.addLog(..));
3
4  after() returning (int logID) : isolateAddLog()
5  {
6      ((Logger) thisJoinPoint.getThis()).deleteLog(logID);
7  }
```

Listing 4.19: Test isolation by deleting inserted log after executing `addLog`

4.3.4 Invoking Unit and Integration Tests on the Service

Services only expose methods that clients are allowed to use. However, there might also be internal methods that have to be tested, such as the private method `openConnection` from listing 4.7. To this end, two methods, `runUnitTests` and `runIntegrationTests`, are added to access the unit and integration tests that are not visible through the service its interface. These two methods can be seen as a simple test suite, consisting of multiple test cases. Listing 4.20 depicts how the methods are exposed in the interface as web service operations by using the `@WebMethod` annotation.

```
1  import javax.jws.WebMethod;
2
3  public aspect IBookManagerTest
4  {
```

```
5       public boolean IBookManager.runUnitTests() {return false;}
6       public boolean IBookManager.runIntegrationTests() {return false;}
7       declare @method: !@WebMethod * IBookManager.*(..) : @WebMethod;
8   }
```

Listing 4.20: Adding test operations to the interface of `BookManager`

The actual implementation of these methods is shown in listing 4.21.

```
1   public boolean BookManager.runUnitTests()
2   {
3       unitTestsPassed = true;
4
5       loanBook(1, 1);
6
7       return unitTestsPassed;
8   }
9
10  public boolean BookManager.runIntegrationTests()
11  {
12      integrationTestsPassed = true;
13
14      openConnection();
15
16      LoggerService loggerService = new LoggerService();
17      ILogger logger = loggerService.getLoggerPort();
18
19      logger.addLog("Integration Test");
20
21      return integrationTestsPassed;
22  }
```

Listing 4.21: Implementation of the test operations for `BookManager`

The client first has to enable testing through the HTTP request headers as done in listing 4.3. Afterwards, the client can choose which tests to run by invoking one of these two methods. As an example we only listed the test cases that were described in this study.

Figure 4.3 depicts how every service has been aspected with BIT technology. The variables used by the test cases, the test cases themselves and the methods of the services have been left out to maintain some overview.

4.3.5 Overhead

We have measured the performance and development overhead that our approach generates.

Performance Overhead

In order to get an idea about the performance overhead, we will compare the performance of the original service without aspects, shown in figure 4.2, to the performance of the service with aspects, shown in figure 4.3, without testing enabled. In other words, both services are used normally by the clients. We used two versions of `BookManager` and for each service we executed `getBook` one thousand times in a row and measure the total time it takes to complete. By dividing that time by one thousand we will get the average execution time. Listing 4.22 lists the code for this performance test.

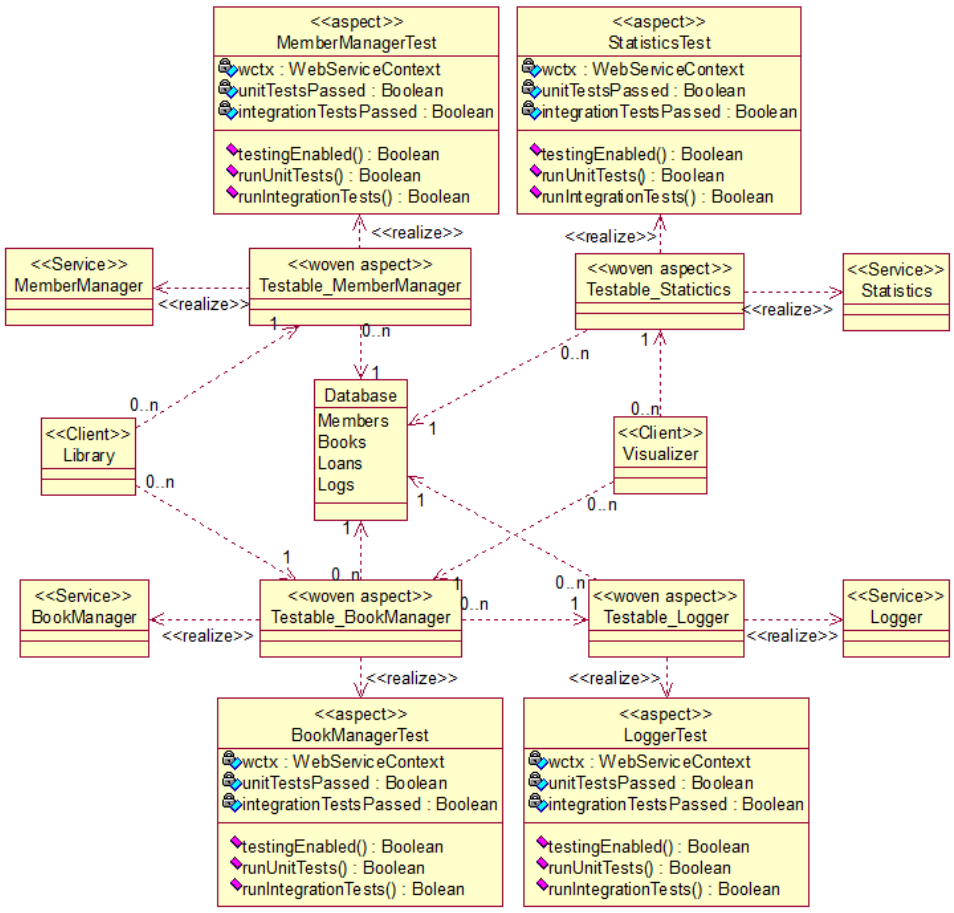


Figure 4.3: Class diagram: Test setup aspected with BIT technology

```
1  import bookmanagerpackage.impl.BookManagerService;
2  import bookmanagerpackage.impl.IBookManager;
3
4  public class BookManagerClient
5  {
6      public static void main(String[] args) throws Exception
7      {
8          BookManagerService service = new BookManagerService();
9          IBookManager bookManager = service.getBookManagerPort();
10
11         long t0, t1;
12         double t2 = 0.0;
13
14         for (int i=0; i<1000; i++)
15         {
16           t0 = System.currentTimeMillis();
17           bookManager.getBook(1);
18           t1 = System.currentTimeMillis();
19           t2 = t2 + (t1 - t0);
20         }
```

```
21          t2 = t2/1000.0;
22          System.out.println("Average time taken to execute getBook after 1000↩
                executions with aspects: " + t2 + "ms");
23      }
24  }
```

Listing 4.22: Client code for the performance test comparison

In order to make sure that our results are accurate, five runs have been done. The results of these runs can be seen in table 4.1.

	Without aspects	With aspects
Run 1	5.265	6.471
Run 2	5.273	6.482
Run 3	5.308	6.523
Run 4	5.395	6.443
Run 5	5.259	6.441
Average	5.300	6.472

Table 4.1: Average execution time in milliseconds of `getBook` after 5x1000 calls

On average the execution of `getBook` with aspects built-in is about 20 percent longer. Although testing is disabled, the pointcut still checks on every execution what the state of testing is. We can reduce the performance loss by adding a variable, `testingAllowed`, at the start of the `testingEnabled` method from listing 4.4. This variable can then be set to allow or disallow testing in general on the service. When testing is not allowed, the code for checking the HTTP request headers is skipped which should increase the performance. When executing `getBook` again for five runs with one thousand executions each run, the average time to complete each execution is 6,424 milliseconds, which is hardly an improvement. This means that the largest performance loss is not coming from the fact that we check if testing is enabled, but from the actual way the aspects are applied during runtime. In order to validate this, we change the condition `if(BookManager.testingEnabled())` to `if(false)` in listing 4.5. Although the average execution time of `getBook` is again decreased, this is only marginal and does not come close to the execution time of the test system without aspects. Furthermore, decompiling the class file of the aspected service shows how AspectJ has generated links to the advice code. Further performance tests with other methods show the same performance loss. The performance loss averages out to about 20 percent for our particular test setup. However, when putting this into perspective the performance loss is only a little over a millisecond which is nothing in a real world scenario where it may take seconds to execute a method on a service.

Development Overhead

When we look at the development overhead a lot of additional code is required, which will have to be added by the service provider. These are the required steps to achieve built-in testing with aspects for JAX-WS services:

- Add the method `testingEnabled` to the service (listing 4.4).
- Write test cases as aspects. An example is shown in listing 4.12.
- Add test operations to the interface of the service and implement them. That way clients can invoke unit and integration tests on the service (listings 4.20 and 4.21).
- As the client, enable testing by modifying the HTTP request headers with testing information before invoking a method on a service (listing 4.3).

In order to generalize the amount of extra required code, table 4.2 shows the code for each required concept for this method.

Concept	Requires	Lines of code
Toggle testing	Method	23
Test case per method	Pointcut, advice, variables	14–x
Test isolation per method	Pointcut, advice, variables	5–x
Test isolation per required service	Pointcut, advice	8
Test suite	Two variables, two methods	23–x

Table 4.2: Required code for each concept, `x` is variable

The amount of required code to toggle testing, and to ensure test isolation for others services that are used by the service, is fixed. Furthermore, these can be automatically generated with a template since they only depend on the name of the service. The skeleton code of the test suite is also the same for every service and can also be generated with a template. The amount of test cases that have to be invoked by the test suite determines the amount of extra lines of code that is required. In general, the invocation of integration tests on other services require three lines of code and the invocation of unit tests on the service require one line of code. The number of lines of code to implement the test cases and test isolation for each method of the service, largely depends on the requirements of the test case. For example, testing if the value of a variable has been increased by one takes less code than testing whether the control flow of a method is correct. For the test cases knowledge of the service itself is required, which makes it the most difficult task when applying our approach to JAX-WS services. However, this is no more complex than creating ordinary test cases with other test approaches.

4.4 Discussion of the Feasibility Study Results

In the next five sub-sections we will discuss the obtained results in regard to each sub research question defined in section 1.1.

4.4.1 Mechanism to Enable and Disable Testing

Disabling and enabling testing by using a simple variable does not suffice. Setting the service in a certain state to enable testing interferes with the other clients who use the service.

Instead, the mechanism we developed is based on the modification of messages between clients and services. We tagged the HTTP request headers with testing information on the client side and processed this tagged communication on the service side. The pointcuts then check whether testing is enabled, and advices are applied accordingly.

The mechanism itself works very well, as demonstrated in the study, and apart from the performance overhead and the additional development effort, there is no disadvantage to applying the technique. Services that are unaware of our BIT approach are not interfered.

4.4.2 Service Test Cases as Aspects

The test cases were created with the service related faults in mind, and they were all implementable as aspects. However, some of the service related faults were caught and handled by the JAX-WS runtime, such as methods that are missing on a service. Nevertheless, we have shown in listing 4.10 that aspects can also be used in the case of missing methods. In case of a detected failure, the location of the fault is printed, which helps localizing the problem.

There is, however, a problem that occurs when a variable is a shared resource. Testing such a variable, for example `numberRequests` in listing 4.12, results in a failure when another client uses it when it is tested by some other client. Although, it is solvable by locking the method that uses the variable, as shown in listing 4.13, this might affect normal usage of the service. When there are no clients testing the service, the locking mechanism is still active, which means that only one client can use the method at any given time. This could be solved by tracking the amount of clients that test the service in a local variable and by skipping the locking mechanism when there are no more clients that are testing the service. Another solution to the shared resource problem would be to duplicate the method or the entire service, but this might be too expensive when taking resources into account. The service provider will have to determine what the best solution is, depending on the required resources.

4.4.3 Test Isolation

Different approaches were used to ensure test isolation. In some cases operations were skipped and in other cases the state of the system was restored. In cases where the state of the system is altered during the test, the locking mechanism can again be used. In order to ensure test isolation when integration tests are performed, the same mechanism used to toggle testing by the client was used. The environment of the service was successfully informed that testing should be enabled. However, services that are not known to the service that is being tested, such as the `Statistics` service from figure 4.2, can be affected when test isolation is achieved by restoring the state after the test is completed since test data is still inserted into the database for a brief moment. In the worst case this test data is retrieved from the database by the `Statistics` service, producing incorrect data as a result. This can be solved by making a copy of the database for the purpose of testing.

A shortcoming of our implementation was discovered while we implemented test isolation for the test case in listing 4.17. Since test isolation and test cases are both implemented

on the same pointcut, they were both executed when testing was enabled. In certain situations, such as a client performing tests on the service to validate that it performs as the client requires, only test isolation should be ensured and the test cases should not be run. To this end, we have to tag the messages with different testing information depending on the needs of the client.

Another solution would have been to implement the test cases as normal methods without the *before*, *after* and *around* advices, as Bruel et al. did with their approach [6], and the test isolation code as aspects. The advantage of this solution is that we do not have to need to differentiate between different tagged testing information and apply the aspects accordingly. However, we still have to record certain information with aspects, such as the control flow of the method in listing 4.14. The disadvantage of the approach by Bruel et al. is that this information is always recorded if we only use one value for the tagged testing information. Furthermore, since the test cases are not added to the interface, the test cases can only be invoked by the service and not by the client using the service.

4.4.4 Invoking Unit and Integration Tests on the Service

A test suite was implemented by adding two methods to the service and by exposing them to the clients. Each test case sets a variable to `false` if the test produces a failure. When all the test cases in the test suite have been performed, the client is informed if the test suite failed or passed.

Although not part of our work, regression tests could be realized using this approach. Regression testing is needed when the service is updated and certain test cases will have to be executed again. The two methods that are used to execute the test suite could be augmented with arguments, and by using different values for these arguments, a selection of test cases can be made that have to be executed again.

4.4.5 Overhead

The performance overhead was measured by comparing the performance of the service without aspects with the performance of the service with aspects without testing enabled. A timer was started and a method was executed many times for average values. A performance loss of around 20 percent was determined. Although we managed to reduce the performance loss by adding an extra variable that allows or disallows testing in general, the effect was minimal. The largest part of the performance loss is caused by the way aspects are applied by AspectJ. By decompiling the aspected service, we can clearly see that extra code and links to code have been added to execute the aspects, resulting in the observed performance loss. The observed performance loss is, however, related to the size of our services. The services used in the study are very small. Heavier services will have bigger methods and more code, but the aspect code stays the same in size, therefore, we expect the overhead to be reduced in realistic services.

The development overhead turned out to be quite excessive when we compare the lines of code of the aspects with the lines of code of the original service. However, with other test approaches the same test cases and test facilities will also have to be written. Having

equipped multiple services with aspects, we can also conclude that once you know how they are implemented, creating test cases and test facilities with aspects is relatively easy. Furthermore, part of the creation of the aspects, such as the method to toggle testing, can be automated.

4.5 Concluding Remarks

Although we have answered the questions we set out in section 4.1 throughout this feasibility study, we will provide a recap of the questions and their answers:

- *Which part of the message can we tag with testing information to let the service know that testing should be enabled?* The HTTP request headers were modified with testing information by the client, allowing the service to process to enable testing accordingly by processing these headers. This is an elegant and cheap approach, and it does not interfere with services that are unaware of our BIT approach.
- *If there is a performance loss, how is this caused?* In our study, we observed a performance overhead of 20 percent, mainly attributable to the inner workings of AspectJ. Aspects are woven in the compiled service and when we decompiled the aspected service, we could clearly see the extra code and links to code that had been added to execute the aspects.
- *What is the complexity of the instrumentation of the service with aspects and can it be automated?* The test facilities, such as the `testingEnabled` method, are fairly easy to understand and to implement and their implementation can be done with a code generator using a template. The most complex part of the instrumentation is the generation of test cases. Good knowledge of the inner workings of the service is required for this task. However, having equipped dozens of services with aspects, we conclude that it is relatively easy to implement the test facilities and test cases.

Chapter 5

Evaluation and Threats to Validity

In this chapter, an evaluation of our approach on an abstract level, as described in chapter 3, will be given. The advantages and disadvantages of using aspects for built-in testing of services will be listed and our sub research questions, set out in section 1.1, will be answered. Lastly, the threats to the validity of our conducted feasibility study will be detailed.

5.1 Evaluation

The goal of aspect-oriented programming is to increase modularity and evolvability of software by separation of concerns. With our approach services are more modular by keeping the test facilities and test cases separated from the original source code. It might, however, lead to badly organized code when dealing with numerous test cases since they can be defined in multiple aspects while still using the same pointcut. The developer will have to make sure that a good overview is maintained by grouping these test cases together.

Compared with other built-in test approaches, such as built-in contract testing [21], aspects offer the same testability of the system under test. The advantage of using aspects is that the original source code of the service does not have to be modified, even when dealing with private members of a service. Normally, `get` and `set` methods would have to be created and inserted into the original source code to access those. In other words, aspects have full access to the class they are woven into. This is especially useful when aspects have to be applied after development, since modification of the original service is not required.

Since the user of the service does not own the service, the service provider will have to provide the test facilities and test cases. Our approach should be easy to implement, but it will still require development time to be spent on it by the service provider. The amount of development time spent by the service provider can be reduced if a tool is used to generate a part of the aspect-oriented BIT framework. In section 4.3.5 we have shown that everything but the generation of test cases can be automated by using a code generation tool with a template for these test facilities. The actual creation of the test cases is the most complex part of the process. However, we believe that this is no more complex than creating ordinary test cases without aspects.

Aspects also decrease the need to duplicate code. For example the test isolation of a certain variable assignment in multiple methods can be defined in a single aspect. However, since our approach does not make a copy of the service to conduct testing, the amount of code is not always decreased. For example, one issue arises when trying to ensure that a shared resource is not modified when a test is in progress, as was shown in section 4.4.2. Although this issue is resolvable by introducing a locking mechanism, this leads to extra code that has to be added and also to possible limitations for other clients since their execution is temporary halted.

The rest of the evaluation will be structured according to the sub research questions listed in section 1.1.

5.1.1 Mechanism to Enable and Disable Testing

Our mechanism to toggle testing relies on modifying the communication between the client and service. By letting the clients tag their messages with testing information, the services can process these tagged messages and enable testing for these clients. The service will, however, have to be able to process these tagged messages. When it is not possible for the service to process these tagged messages, test mode cannot be activated. Another way to let the service know that testing should be enabled is by adding a parameter to the method signature of that service and use that parameter to send testing information. However, this would defeat the purpose of using aspects since their goal is to leave the original service intact and not equip them directly with code that is only used for testing.

5.1.2 Service Test Cases as Aspects

The test cases, defined with the aid of the fault taxonomy from section 2.1.2, are all implementable as aspects. The issue where a shared resource was modified during a test was resolved with a locking mechanism. The disadvantage of this solution is that it might hinder other clients since their execution is halted while the method is being tested. Another solution is to make a copy of the method or entire service, but this might be too expensive when looking at the amount of extra required resources. Test cases that have their own temporary variables also cause problems when multiple clients run the same test. The solutions for the shared resources problem can also be used to solve this problem. If running certain test cases does not require a long amount of time, then the locking mechanism suffices. When these test cases take minutes to complete, it might be better to use a copy of the method or service instead to not hinder other clients. It is up to the service provider to determine the appropriate solution.

5.1.3 Test Isolation

We have shown that test isolation can be achieved by skipping certain operations. However, this might interfere with other test cases since they need the result of this specific operation. Another solution to ensure test isolation is to restore the state of the service after the test is completed. This does, however, increase the overhead since extra method invocations are required to undo the state changes and to remove test data from the system.

For integration tests, the environment can be made aware of the fact that testing should be enabled by using the same mechanism used to toggle testing. However, in the case of restoring the state after testing is completed, test data is still briefly inserted into the system. Services that are not known to the service that is being tested, and are therefore not informed that testing is enabled, might retrieve this test data and produce incorrect results. A solution to this problem would be to replicate the resources for the purpose of testing.

5.1.4 Invoking Unit and Integration Tests on the Service

In order to invoke unit and integration tests on the service, a test suite was included as an aspect. The extra methods that are required in the interface, and the implementation itself are added with aspects. Our solution for the test suite can possibly be extended to support regression testing, as was discussed in section 4.4.4.

5.1.5 Overhead

When looking at the performance of aspects, a performance overhead seems unavoidable. After all, aspects are woven into the original service and have to be dealt with during runtime. Also, on every execution the program checks if testing is enabled or not which reduces the performance even more. Furthermore, our approach allows any client to test the service whenever needed, which could strain the system. This can be solved by scheduling the tests, but this may not be desirable since it might hinder other clients if they cannot test whenever they want. The performance depends on the hardware and the implementation of the services. The actual overhead is expected to be a couple of milliseconds for each method invocation.

The development overhead depends on the number of test cases and on the amount of code that is required to achieve test isolation that have to be written. The unit and integration test methods, and the methods to check if testing is enabled take the least amount of code and are almost the same for every service which means that they can be automatically generated by using a template. Overall, the development overhead seems quite excessive, but with other test approaches the same test cases and test facilities will also have to be written. We believe that using aspects is no more complex than using other test approaches to create the test cases and test facilities.

5.2 Threats to Validity

For the validation of our feasibility study, the threats to validity described by Wohlin et al. [33] have been used.

In our study Java was used for the implementation of our service-oriented system since Java can achieve increased platform independence. Nowadays there are many more variations of aspect compilers for other programming languages, such as AspectC++ for c and c++ [1]. The terminology used to define aspects is roughly the same, but the implementation differs since programming languages work differently.

When trying to generalize the result, we have to take into account that other types of services, and other AOP extensions for different programming languages might not provide the same functionality. Our implementation of our mechanism, for example, relies on some JAX-WS features that allow us to tag messages and process tagged messages with testing information. All AOP extensions provide the *before*, *after* and *around* advices, which means that our test cases, and the way we achieved test isolation can be achieved in the same manner. Furthermore, the addition of extra methods, which are needed to implement our test suite, and to check if testing is enabled, is also something that is possible with all AOP extensions. In general, we believe that our approach can be applied to other types of services, possibly coded in other languages, but the actual implementation of the mechanism to toggle testing might be dependent on the type of service.

A threat to the internal validity of our study is that, although, we have determined the performance overhead, this might not be representative for other AOP extensions or programming languages since they might be faster or slower. However, since extra checks are made to see if testing is enabled, a minor performance loss at the very least should always be expected. The amount of required code for the test facilities and test cases might also differ depending on the AOP extensions and used programming languages.

Furthermore, our test setup consisted of a small service-oriented system and not a real world scenario since this was not available for our work. An evaluation of our approach applied to a large service-oriented system, with a high number of transactions, is required to determine the additional development work that is required, and the execution overhead that is introduced in such systems.

Finally, all test cases were defined using the fault taxonomy described in section 2.1.2 and, therefore, we might have missed test cases. Furthermore, JAX-WS intercepted a number of faults that were present in the description of the service and that we could not test with test cases. However, we do believe that we can create our own test cases for these faults, as we have shown, for example, for missing methods during a binding test in section 4.3.2.

Chapter 6

Conclusions and Future Work

This chapter starts with a summary of the work we did and the contributions we have made. We will then reflect on our results and draw some conclusions. Finally, we will discuss some possible ideas for future work.

6.1 Summary and Contributions

Services are used a lot nowadays in order to deploy large scale systems that should never be shut down such as internet shops, online banking applications, and surveillance systems. Because services can be updated at any given time, testing is required during runtime by its clients to validate that the updated service still performs as required. However, testing during runtime may interfere with the nominal operations of the service and its environment. Furthermore, only the methods exposed by the service through its interface are available to the clients, which reduces the ability to control and observe the service for testing purposes, i.e. its testability. Built-in testing can be used to increase the testability, and to ensure that other clients are not hindered while the service is being tested. However, in order to increase the testability of the service, methods will have to be added to the service, which decreases the modularity and evolvability of the service. In order to maintain the modularity of the service, aspects can be used to separate the test facilities and test cases from the service.

The goal of this thesis project was to find out how aspects can be used to achieve built-in testing for services. First of all we presented the background and related work to the topic. Testing for services was explained, several BIT approaches were listed and testing with aspects was introduced. With the obtained knowledge our approach was created following each sub research question we set out in section 1.1. We then applied our approach in a feasibility study to a service-oriented system consisting of four JAX-WS services, two clients and database, and discussed the results. Lastly, we evaluated our approach in general and discussed the threats to the validity of our feasibility study.

Although our approach is closely related to the approach by Bruel et al. [6], where aspects are used to achieve BIT for components, we expanded on it to overcome its shortcomings when applied to services, in particular the lack of a mechanism to differentiate between nominal mode and test mode, service related test cases, test isolation during run-

time testing, a test suite to invoke unit and integration tests on the service, and an analysis on the overhead of aspects.

6.2 Conclusions

Our main research question was how aspects could be used to achieve built-in testing for services. We divided this question into five sub-questions, which we have answered throughout the thesis by creating an approach and validating this approach in a feasibility study.

The conclusions we have drawn are listed per subquestion:

- *Which mechanism can be used to differentiate between nominal mode and test mode?* The modification of the communication between clients and services was used to toggle testing. The client tags the messages with testing information, and the service processes these tagged messages to check if testing should be enabled. When the service is unable to process tagged messages, test mode cannot be activated.

- *To which extent can test cases for services be implemented as aspects?* The test cases, defined with the help of the fault taxonomy from section 2.1.2, were all implementable as aspects. The largest issue when creating and executing test cases is when there is a shared resource. This was solved by locking the operation for other clients while a test is performed. The disadvantage to this solution is that it might hinder other clients. Another solution was to replicate the method or service, but this might be too expensive when the method or service is complex, since it requires extra resources. When the test case does not take up a large amount of time to execute, we advise to use the locking mechanism. If the test case requires minutes to complete, it might be better to use a copy of the method or service instead to not hinder other clients. In the end it is up to the service provider to determine the appropriate solution.

- *To which extent can test isolation be achieved during runtime testing with aspects?* In general, we have shown that test isolation can always be achieved. In certain cases operations can be skipped, while in other cases the state of the system can be restored after the test case is completed. For integration tests the environment can be made aware of the fact that testing should be enabled by using the same mechanism used to toggle testing. However, in the case of restoring the state of the system after testing is completed, test data is still briefly inserted into the system. Services that are not known to the service that is being tested, and are therefore not informed that testing is enabled, might retrieve this test data and produce incorrect results. A solution to this problem would be to replicate the resources for the purpose of testing. Again, it is up to the service provider to determine the appropriate solution. Especially the requirements, i.e. may clients be hindered, and the required resources, e.g. a copy of the database, should be taken into consideration.

- *How can unit and integration tests on a service be invoked?* A test suite was included in the service to allow clients to run unit and integration tests on the service itself.

The extra methods that are required in the interface, and the implementation itself are added with aspects. Our solution for the test suite can possibly be extended to support regression testing, by augmenting the two methods with arguments. By using different values for these arguments, a selection of test cases that have to be executed can be made.

- *How much overhead is generated when aspects are used to achieve built-in testing for services?* Although the performance is dependent on the test setup, a performance overhead can be expected since the program checks if testing is enabled on every method execution. Furthermore, our approach allows any client to test the service whenever needed, which could strain the system even more. This can be solved by scheduling the tests in some way, but this may not be desirable since it might hinder other clients who want to test the service. The performance overhead was determined to be around 20 percent for our particular test setup. This is, of course, relative to the amount of time it takes to execute a method. The actual overhead is a little over one millisecond for our test setup. The development overhead depends on the amount of test cases and on the code that is required to achieve test isolation that have to be written. The unit and integration test methods and the method to check if testing is enabled take the least amount of code. They are almost the same for every service and they can, therefore, be automatically generated by using a template. Overall, the development overhead is quite excessive, however, with other test approaches the same test cases and test facilities will also have to be written.

In general, we conclude that aspects can be used effectively to achieve built-in testing for services. However, more evaluation is required to assess how our approach will hold up when applied to a large real world service-oriented system.

6.3 Future work

Since we did not have access to a large service-oriented system, we did not have the chance to assess how our approach would hold up in a system with a high number of transactions. More evaluation is needed to determine the additional development work that is required to achieve BIT with aspects. and the execution overhead that is introduced with our approach in such systems.

The service user does not own the service which means that the service provider has to equip the services with the test facilities and test cases. In order to facilitate this task, a tool should be developed to automate the generation of aspects for the required test facilities and to aid with the creation of test cases. Many of the test facilities we used, such as enabling and disabling the aspects, can be created by using a template. By using reflection on the compiled classes, all the members and functions of a class can be obtained. This information can be used by the tool, together with input from the user, to aid with the creation of the test cases.

Another idea is to automatically test the service when it is deployed or updated. The test suite, which is included with our approach, can then automatically be executed by the application server. It also has to be extended to support regression testing.

Lastly, our approach was only applied to a system consisting of JAX-WS services. Although we believe that our approach can be applied to other implementations of services, with different AOP extensions and in different languages, an evaluation is still required to determine the suitability of our approach in that regard.

Bibliography

[1] Aspectc++. http://www.aspectc.org/.

[2] Sami Beydeda and Volker Gruhn. *Testing Commercial-off-the-shelf Components And Systems*. SpringerVerlag, 2005.

[3] Robert V. Binder. Design for testability in object-oriented systems. *Commun. ACM*, 37:87–101, September 1994.

[4] Daniel Brenner, Colin Atkinson, Rainer Malaka, Matthias Merdes, Barbara Paech, and Dima Suliman. Reducing verification effort in component-based software engineering through built-in testing. *Information Systems Frontiers*, 9(2-3):151–162, 2007.

[5] Alan Brown, Simon Johnston, and Kevin Kelly. Using service-oriented architecture and component-based development to build web service applications. 2002.

[6] J.M. Bruel, J. Araújo, A. Moreira, and A. Royer. Using aspects to develop built-in tests for components. *The 4th AOSD Modeling with UML Workshop*, 2003.

[7] Stefan Brüning, Stephan Weißleder, and Miroslaw Malek. A fault taxonomy for service-oriented architecture. In *HASE '07: Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, pages 367–368, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Christian Bucanac. The v-model. 1998.

[9] G. Canfora and M. Di Penta. Testing services and service-centric systems: challenges and opportunities. *IT Professional*, 8(2):10 –17, March-April 2006.

[10] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, 2004.

[11] Xerox Corporation. Aspectj. http://www.eclipse.org/aspectj/.

[12] Schahram Dustdar and Stephan Haslinger. Testing of service oriented architectures - a practical approach. 2004.

[13] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[14] Eclipse Foundation. Aspectj quick reference. `http://www.eclipse.org/aspectj/doc/next/quick5.pdf`.

[15] The Apache Software Foundation. Apache tomcat application server. `http://tomcat.apache.org/`.

[16] Roy S. Freedman. Testability of software components. *IEEE Trans. Softw. Eng.*, 17(6):553–564, 1991.

[17] Alberto González, Éric Piel, and Hans-Gerhard Gross. Architecture support for runtime integration and verification of component-based systems of systems. *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 41–48, 2008.

[18] Alberto González, Éric Piel, and Hans-Gerhard Gross. A model for the measurement of the runtime testability of component-based systems. In *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 19–28, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Michaela Greiler, Hans-Gerhard Gross, and Khalid Adam Nasr. Runtime integration and testing for highly dynamic service oriented ict. solutions an industry challenges report. 2009.

[20] Michaela Greiler, Hans-Gerhard Gross, and Arie van Deursen. Evaluation of online testing for services: a case study. In *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, PESOS '10, pages 36–42, New York, NY, USA, 2010. ACM.

[21] Hans-Gerhard Gross. *Component-based Software Testing With Uml*. SpringerVerlag, 2004.

[22] IEEE. Ieee standard glossary of software engineering terminology. 1990.

[23] Genivia inc. gsoap. `http://www.cs.fsu.edu/~engelen/soap.html`.

[24] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. In Jrgen Knudsen, editor, *ECOOP 2001 Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin / Heidelberg, 2001.

[25] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.

[26] Mika Koskela, Mikko Rahikainen, and Tao Wan. Software development methods: Soa vs. cbd, oo and aop. 2008.

[27] LuaForge. Aspectlua. `http://aspectlua.luaforge.net/`.

[28] Sun Microsystems. Java api for xml web services (jax-ws). `http://jax-ws.java.net/`.

[29] Ron Patton. *Software Testing, 2nd edition*. Sams Publishing, 2005.

[30] John Stamey and Bryan Saunders. Unit testing and debugging with aspects. *J. Comput. Small Coll.*, 20:47–55, May 2005.

[31] D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes, and R. Malaka. The morabit approach to runtime component testing. In *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, volume 2, pages 171 –176, sept. 2006.

[32] Yingxu Wang, Graham King, and Hakan Wickburg. A method for built-in tests in component-based software maintenance. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pages 186–189, Washington, DC, USA, 1999. IEEE Computer Society.

[33] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

Appendix A

Example of Using Aspects

This example shows how the execution of a method can be altered with aspects.

```
1  public class Calculate
2  {
3      public static void calculate(int number1, int number2)
4      {
5          System.out.println(number1+number2);
6      }
7
8      public static void main(String[] args)
9      {
10         calculate(1,2);
11     }
12 }
```

Listing A.1: Adding two numbers with method `calculate` and printing the result

```
1  public aspect CalculateAspect
2  {
3      pointcut message() : execution(void Calculate.calculate(..));
4
5      before() : message()
6      {
7          System.out.println("Entering method calculate()");
8      }
9
10     after() : message()
11     {
12         System.out.println("Exiting method calculate()");
13     }
14 }
```

Listing A.2: Printing messages before and after execution of method `calculate` with an aspect

```
1 public aspect CalculateAspect
2 {
3     pointcut changeNumbers(int i, int j) : execution(void Calculate.↩
          calculate(..)) && args(i, j);
4
5     Object around(int i, int j) : changeNumbers(i, j)
6     {
7         return proceed(i+1, j+1);
8     }
9 }
```

Listing A.3: Changing behavior of method `calculate` with an aspect

Appendix B

Schema of Test Setup Database

Figure B.1 depicts the database used by the service-oriented system described in our feasibility study in chapter 4.

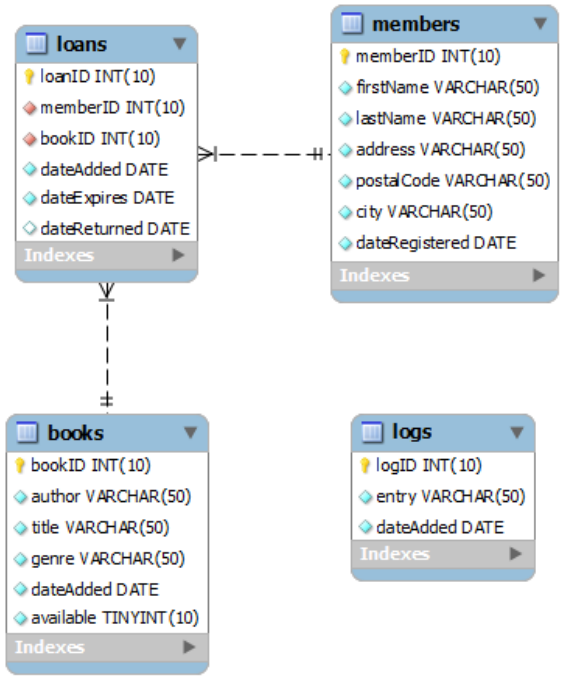


Figure B.1: Schema of the database used in the test setup

Appendix C

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

Advice: Code that is run when a pointcut has been reached. There are three different advices: *before*, *after* and *around* which determine where the advice is executed.

AOP: Aspect-Oriented Programming. A programming paradigm which aim is to increase modularity by separating concerns that cut across other concerns, e.g. test code that logs information about a method.

Aspect: Contains pointcuts, advice code and other methods and variables for the purpose of extending and changing the original software.

BIT: Built-in Testing. Equipping software with test facilities and test cases to give it the ability to test itself, its environment, and be tested by its environment.

Client: The user of a service, which can be an end-user application or another service.

Failure: Observed misbehavior.

Fault: A ”mistake” in the code.

HTTP: HyperText Transfer Protocol. A networking protocol that allows for data communication between systems.

Integration Test: Integrated units that are tested according to the system design.

Interface: Used to connect software units for the purpose of passing information from one to the other.

JAX-WS: Java API for XML Web Services. Library used to develop Java-based web services.

Join Point: A well-defined point in the execution of a program.

Pointcut: A set of join points.

Regression Test: Partial retesting of the system whenever an update to the system occurred.

Service: A course-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model.

SOAP: Simple Object Access Protocol. Used by web services to exchange information with other services and clients. It can use HTTP to send XML messages

Test Case: A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test Facility: Extra code that facilitates achieving test isolation and, for example, setting the software into a certain state.

Test Isolation: The means to prevent test operations from interfering with the production state of the system, and the state of its environment.

Test Suite: A collection of test cases and plans to use them.

Testability: (1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.

Testing: (1) The process of operating a system or component/service under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component/service. (2) The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.

Unit Test: The smallest testable part of a program. A unit is tested following its specification stated in the design models.

WSDL: Web Services Description Language. Describes the interface operations that are available to the clients of the service.

XML: Extensible Markup Language. A set of rules for encoding documents in machine-readable form.