University of Porto
Faculty of Engineering

Incremental Modular Testing in Aspect Oriented Programing

André Monteiro de Oliveira Restivo

August 2015

Scientific Supervision by

Ademar Aguiar, Assistant Professor
Departmento de Engenharia Informática
Faculdade de Engenharia da Universidade do Porto

Ana Moreira, Associate Professor
Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

In partial fulfillment of requirements for the degree of
Doctor of Philosophy in Computer Science
by the Doctoral Program in Informatics Engineering

Contact Information:

André Monteiro de Oliveira Restivo
Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Rua Dr. Roberto Frias, s/n
4200-465 Porto
Portugal

Tel.: +351 22 508 1772
Fax.: +351 22 508 1440
Email: arestivo@fe.up.pt
URL: http://www.fe.up.pt/~arestivo

This thesis was typeset on an Lenovo E530® running Ubuntu 12.04® using the free LaTeX typesetting system, originally developed by Leslie Lamport based on TeX created by Donald Knuth. The body text is set in Latin Modern, a Computer Modern derived font originally designed by Donald Knuth. Other fonts include Sans and Typewriter from the Computer Modern family, and Courier, a monospaced font originally designed by Howard Kettler at IBM and later redrawn by Adrian Frutiger. Typographical decisions were based on the recommendations given in *The Elements of Typographic Style* by Robert Bringhurst (2004), and graphical guidelines were inspired in *The Visual Display of Quantitative Information* by Edward Tufte (2001). This colophon has exactly one hundred and twenty-four (124) words, excluding all numbers and symbols.

André Monteiro de Oliveira Restivo
"Incremental Modular Testing in Aspect Oriented Programing"

to my parents, for the unconditional support

to Filipa, for understanding

Abstract

Developing good quality software is a hard task. The development cycle is composed of several tasks that must fit perfectly in order to produce software that is usable, maintainable, reliable, extensible and secure. The reason why it is so hard to attain these goals is that software is extremely complex. A large software product is composed of several parts that must be perfectly integrated like in a giant jigsaw puzzle.

If done correctly, the parts that compose a software product, let us call them modules, are going to be as loosely coupled as possible. Modules should need to know as little as possible about the inner workings of other modules. This so-called modularity is of paramount importance as it makes modules more understandable, autonomous and promotes reusability.

The ideal scenario would be one where each module is only responsible for one concern. Developing software would then be a question of composing these modules together. Unfortunately, although software development paradigms and techniques are surely focused in achieving this perfect separation of concerns, there are still some concerns that get inevitably scattered and tangled throughout the code.

Aspect Oriented Programming (AOP) is a paradigm that promises to tame these crosscutting concerns and separate them into their own units of modularity. The way it proposes to do so is to enable the developer to pinpoint specific points of interest in the code, using a domain specific language, and then inject context aware code into those points. In this way, a concern that used to be scattered, can be contained and still affect many different parts of the code.

Software testing provides stakeholders a measure of the quality of the software being produced. It is one of the activities of the development cycle and is extremely important in order to achieve good quality software. Unit testing is specially

important when we are trying to create modular software, as it allows testing each module almost independently from others.

Unfortunately, as we will show in this thesis, using AOP hinders the usage of unit testing. You can either throw away part of the modularity gained by using it or drop some unit tests. This happens due to the capability of AOP to change the behavior of existing modules, making tests that have been specifically created for those modules only work when the AOP code is not present. Changing the tests to account for the AOP code breaks the modularity, and the only other existing option is to remove those tests.

In this thesis, we propose using semi-automatic code inspection to create a tree of the dependencies between modules and use an incremental testing approach that allows each software layer to be tested in isolation. As layers are added, one after the other, if a certain part of the code breaks a previous test, it will be possible to mark the broken tests as being replaced by new tests that take into account the modifications that have been introduced by the AOP code. This allows unit tests in lower layers to still be usable without breaking the modularity promised by AOP.

A testbed was created to test the proposed approach and we were able to create and use unit tests that would otherwise be impossible to run in the complete system. We also showed how the approach is able to tackle some AOP specific software faults. To help with the tests, we developed a reference implementation of a support tool.

We believe that the approach being presented solves an important problem that prevents the adoption of AOP from being more widespread. We do not assume to have solved all AOP problems, but we think this thesis presents an important step in the right direction as it enables a more sane usage of tests in conjunction with aspects.

Resumo

O desenvolvimento de *software* de boa qualidade é uma tarefa complicada. O ciclo de desenvolvimento é composto por várias tarefas que têm de se coordenar perfeitamente, a fim de produzir *software* utilizável, mantível, confiável, extensível e seguro. A razão porque é tão difícil de atingir estes objetivos é que o *software* é extremamente complexo. Um produto de *software* de grandes dimensões é composto por várias peças complexas que têm de se encaixar perfeitamente como num *puzzle* gigante.

Se feito corretamente, as peças que compõem um projeto de *software*, vamos chamá-las de módulos, vão ser fracamente acoplados. Os módulos devem precisar saber tão pouco quanto possível sobre o funcionamento interno dos outros módulos. A modularidade é de grande importância, pois acaba por produzir módulos mais fáceis de perceber e promove a sua reutilização.

Idealmente, cada módulo deveria ser responsável por apenas um único *interesse* (em inglês *concern*). O desenvolvimento de *software* seria então uma questão de composição destes módulos. Infelizmente, apesar da existência de técnicas e paradigmas de programação focados em conseguir uma perfeita separação destes *interesses*, ainda existem alguns que ficam inevitavelmente espalhados e misturados ao longo do código.

A Programação Orientada a Aspectos (AOP) é um paradigma que promete separar esses interesses transversais nas suas próprias unidades de modularidade. A forma como se propõe fazê-lo é permitindo ao programador que identifique pontos de interesse no código, usando uma linguagem específica do domínio e, em seguida, injecte código com acesso ao contexto nesses pontos. Desta forma, um interesse que esteja espalhado, pode ser contido e ainda assim afectar diferentes partes do código.

O processo de testes fornece às partes interessadas uma medida da qualidade do *software* produzido. É uma das fases do ciclo de desenvolvimento, e é extremamente importante a fim de conseguir que o *software* seja de boa qualidade. O teste de unidade é especialmente importante quando estamos a tentar criar um *software* modular, pois permite testar cada módulo separadamente, isto é, sem a influência de outros módulos.

Infelizmente, como ficará demonstrado nesta tese, a utilização de AOP dificulta o uso de testes de unidade. A solução é descartar alguns testes ou perder parte da modularidade ganha com o uso de AOP. Isto acontece devido à capacidade que os aspectos têm de alterar o comportamento dos módulos existentes, fazendo que testes que foram criados especificamente para esses módulos não funcionem quando certos aspectos estão presentes. Alterando os testes para terem em conta o código AOP quebra a modularidade, e a única outra opção existente é remover esses testes.

Nesta tese, propomos a utilização de métodos de inspeção de código para criar uma árvore de dependências entre módulos e o uso de uma abordagem de teste incrementais que permitem testar cada camada de *software* em isolamento. À medida que as camadas são adicionadas, uma após a outra, quando uma nova camada, que contém aspectos, é adicionada, será possível marcar testes que sejam quebrados por esta nova camada como sendo substituídos por novos testes que levam em consideração as alterações que tenham sido introduzidas por código AOP. Isto permite que testes de unidade em camadas mais baixas ainda possam ser utilizados sem quebrar a modularidade prometida pelos aspectos.

Foi criado um projecto para servir de bancada de testes para a abordagem proposta e, nesse projecto, fomos capazes de criar e usar testes de unidade que de outra forma seriam impossíveis de executar. Também mostramos como a abordagem é capaz de resolver algumas falhas comuns específicas ao AOP. Para ajudar com os testes, desenvolvemos uma ferramenta de apoio de referência.

Acreditamos que a abordagem apresentada resolve um problema importante que impede uma adopção mais alargada do AOP. Não assumimos ter descoberto a solução para todos os problemas relacionados com AOP, mas achamos que esta tese apresenta uma passo importante na direcção certa, uma vez que permite um uso mais fácil de testes em conjunto com os aspectos.

Contents

List of Figures

List of Tables

List of Listings

Preface

Having read quite a few of these prefaces myself, I have to admit that I feel like a *cliché* when trying to explain where my inclination towards the computer sciences comes from. I was about 6 or 7 years old when my parents bought me a 1982 *ZX Spectrum 48K*. It was quite a remarkable piece of technology and no one I knew had one at that time.

In the beginning, I mostly used it to play games. For those who never had the pleasure to hold one of these, it is important to notice that the main I/O device was a common tape recorder. One day, something seemingly ordinary happened, that ended up having an extraordinary influence on my life. The tape recorder stopped working and I really wanted to play. So I picked up the instruction manual and started reading it, looking for a way to fix it. This very small book did not contain what I expected, it was a manual for a language that was preloaded into every one of these devices: Sinclair BASIC. I started fiddling with it and, since that day, I never stopped enjoying programming.

When I was not playing with my computer, I was either playing with my friends or doing something that, in retrospective, also shaped my future. The *LEGO* system was always one of my preferred toys. What really attracted me to it, was that after following the manual that came with each box set, one could just combine the pieces with pieces from other boxes to create bigger, and sometimes better, constructions. This modularity was something that always fascinated me and that I will always consider a major cornerstone of every simple and elegant system.

The combination between programming and the elegance of modular systems was only made apparent to me a few years later, when one of my favorite high school teachers helped me transform, what was at the time, an horrible spaghetti code

into modular procedural code. After that episode, I always strive to make my code as modular as possible.

Having explained where my inclination for the subject came from, it is still hard to pinpoint exactly where the topic of this thesis came from. I remember that the first time I read about *aspect oriented programming*, it seemed that a very well kept secret had been revealed to me. It seemed the solution for all my modularity problems – it ended up not being exactly that – and I immediately decided that I wanted to explore the matter further. After that, the topic I was pursuing seemed to gain a life of its own as it changed from detecting interactions between *aspects* to testing *aspect oriented* code.

This thesis would have been possible, but would have been harder, if not for the help of all my friends and colleagues that never gave up on asking me if it was finished. All the time, even when asked not to. I thank them for their support and I do not dare to try to enumerate them for fear of forgetting someone.

I have to thank especially my good friend Sérgio Carvalho who had to endure my ramblings about aspects, especially in the beginning of this work and ended up being an excellent *think wall*. A thank you note goes to Miguel Pessoa Monteiro for his help on constantly broadening my perspectives regarding interferences due to *aspect* composition.

And of course to my supervisors Ademar Aguiar and Ana Moreira, who never gave up on me and always offered their support and guidance whenever and wherever I needed.

A final word of appreciation goes to FCT, for the support provided through scholarship *SFRH/BD/32730/2006*.

André Restivo
Porto, Portugal, August 2015

"Begin at the beginning,", the King said, very gravely, "and go on till you come to the end: then stop"

Lewis Carrol – 1865

1

Introduction

Contents

The software development cycle is a complicated beast. It is composed of many stages, each one producing a series of artifacts that should fit together perfectly to produce a product that is as correct as possible.

Many other desirable characteristics are usually associated with the development of good software. Software must be usable, maintainable, scalable, reliable, extensible and secure just to mention a few. Most of these are strictly related to another characteristic that we normally associate with well-written software: modularity.

When we call a piece of software modular, what we are illustrating is the fact that it is composed of several smaller units that can be described, reused, replaced and tested in isolation. It is a fundamental characteristic of software and a cornerstone for all the other desirable characteristics.

It has been long known that some software concerns are hard, if not impossible, to confine to a single unit of modularity. In fact, they often encompass all the code making a perfectly modular software system very hard to achieve. Aspect Oriented Programming (AOP) appeared with the promise that it would allow these concerns to be separated in their own units of modularity.

However, Modularity is related to more than just the code itself. Other artifacts, like documentation and test cases, must also be modular if we want to be able to reuse modules easily .

In this thesis, we intend to analyze the impact of AOP in the modularity of test cases and how we can improve it.

This chapter starts by briefly presenting the context and scope of this dissertation, followed by its goals, main results, and a description of each one of the other chapters.

1.1 Aspects and Modularity

AOP is a programming paradigm that builds on the success of the classical Object Oriented Programming (OOP) paradigm. It states that crosscutting concerns, those that are impossible to encapsulate into their own units of modularity using OOP, can be separated by using special code constructs.

This is possible because AOP languages contain two important mechanisms. The first one allows developers to identify certain points in the execution flow and the second one allows the introduction of code at those points. In this way, code that used to be spread throughout several units can be kept together in a single unit of modularity.

1.2 Motivation

The advantages of a modular approach to software design are all well known to developers. However, when a new software development approach appears, that promises to improve the overall modularity of software systems, it often finds resistance amongst developers as some of their old methods and practices are incompatible with it.

A number of software developing techniques that improve modularity are already well established in the development community. These techniques allow developers to easily reuse great portions of their code and design. We have reached a point where small improvements to current techniques are only accepted by developers if they do not affect their current workflow or there is a dramatic benefit that is perfectly clear.

To improve the acceptance of AOP amongst developers, we have to realize that software modules are no longer just another piece of code. They encompass many other artifacts like documentation and tests. To achieve true modularity, all these components must be addressed. We believe that by contributing to improve testing techniques for AOP, without changing their core aspects, we will help mitigate AOP acceptance issues by the developer community.

1.3 Research Goals

The goal of this thesis is to make testing compatible with the ideals proposed in the AOP paradigm. We will argue that, at this moment, mainstream testing procedures break the modularity that AOP tries to implement.

This happens because AOP allows units to modify the behavior of other units without their knowledge. This is often called *obliviousness* and is a fundamental property of the paradigm.

To solve this problem, we propose a new technique that we called *Incremental modular testing in Aspect Oriented Programing*. This technique, detailed in Chapter 6, evolved from the thesis that:

An incremental testing solution allows developers to keep the promise of modularity achieved by using AOP, without compromising the outcome of the testing process.

1.4 Research Strategy

Software engineering is still a relatively recent research field and is an area that is still maturing. There are several characteristics of software development that suggest its own research paradigm, combining aspects from other disciplines: it is a human creative phenomenon; software products are costly and usually have long cycle times; it is difficult to control all relevant parameters; technology changes very frequently, so old knowledge becomes obsolete fast; it is difficult to replicate studies; and there are few common ground theories.

A categorization proposed at Dagstuhl workshop [THP93], groups research methods in four general categories, quoted from Zelkowitz and Wallace [ZW98]:

- **Scientific method.** *"Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis."*
- **Engineering method.** *"Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement."*
- **Empirical method.** *"A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis."*
- **Analytical method.** *"A formal theory is developed, and results derived from that theory can be compared with empirical observations."*

These categories apply to science in general. Effective experimentation in software engineering requires more specific approaches. Software engineering research comprises computer science issues, human issues and organizational issues. It is thus often convenient to use combinations of research approaches both from computer

science and social sciences. The taxonomy described by Zelkowitz and Wallace [ZW98] identifies twelve different types of experimental approaches for software engineering, grouped into three broad categories:

- **Observational methods.** *"An observational method collects relevant data as a project develops. There is relatively little control over the development process other than through using the new technology that is being studied".* There are four types: project monitoring, case study, assertion, and field study.
- **Historical methods.** *"A historical method collects data from projects that have already been completed. The data already exist; it is only necessary to analyze what has already been collected".* There are four methods: literature search, legacy data, lessons learned, and static analysis.
- **Controlled methods.** *"A controlled method provides multiple instances of an observation for statistical validity of the results. This method is the classical method of experimental design in other scientific disciplines".* There are four types of controlled methods: replicated experiment, synthetic environment experiment, dynamic analysis, and simulation.

The best combination of methods to use in a concrete research approach is strongly dependent on the specific characteristics of the research study to perform, viz. its purpose, environment and resources. Hereafter, the research methods referred will use this terminology. Further description of each method can be found in [ZW98].

Based on the expected results and contributions of the work presented in this dissertation, the research strategy was based on the usage of observational methods. In particular, a series of case studies, the most important of which is documented in Chapter 8.

1.5 Main Results

The primary outcomes of this thesis encompass the following aimed contributions to the body of knowledge in software engineering:

- A detailed explanation of the problems introduced by testing in aspect oriented programming.

- A technique that allows the usage of testing procedures with aspect oriented programming without breaking the modularity that it strives to achieve.
- A testbed that can be used by the community to study the interactions between aspects and unit testing.
- A reference implementation of a support tool for the developed approach.
- The evidence of benefits of using the developed technique.

1.6 How to Read this Dissertation

The remaining of this dissertation is organized into three parts, with the following overall structure:

- **Part 1: State of the Art**. Where the author introduces the context of the thesis and presents the current state of the art in the research areas most relevant for the understanding of this document.
  - Chapter 2, "Modularity" (p. 11), explains the importance of modularity in the context of software development as well as the difficulties in achieving it and some of the techniques developed to tackle the subject.
  - Chapter 3, "Aspects" (p. 23), introduces the notion of AOP, its objectives and shortcomings, and a particular implementation of the paradigm in the form of the language AspectJ.
  - Chapter 4, "Testing" (p. 47), lays down the different types of testing involved in the software development cycle and the importance of each one in obtaining quality software.
- **Part 2: Problem and Solution**. Where the author first describes the problem to be solved and proposes a solution.
  - Chapter 5, "Unit Testing Aspects" (p. 55), explains how introducing testing to AOP code breaks the modularity gained or compromises the quality of the tests.
  - Chapter 6, "Modular Testing in AOP" (p. 77), presents a technique that aims at fixing the problem identified in the previous chapter.
  - Chapter 7, "Implementation" (p. 97), presents a reference implementation for the technique presented in the previous chapter based on annotations and incremental compilation.

- **Part 3: Validation and Future Work**. Where the author explains how the work done was validated and points to future developments.
  - Chapter 8, "Results and Validation" (p. 107), shows how the technique behaves by applying it to a testbed developed for this same effect.
  - Chapter 9, "Conclusions and Future Work" (p. 137), outlines the main conclusions of this dissertation, and points to further work.

Part I

State of the Art

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

Alan J. Perlis – 1982

2

Modularity

Contents

In this chapter, we will discuss how we can use modularity to tackle the inherent complexity of software. We will also discuss several paradigms and techniques used to improve modularity.

2.1 On the Complexity of Software

Software development is difficult; doing it well is very difficult. In fact, doing anything well is always difficult. Creating a building, deriving a complex mathematical formula or writing a novel, wherever there is complexity, we have a hard time coping with it.

Modern software systems, at least those that are large enough to be worth considering, are inherently complex. Some of this complexity is accidental and is introduced by the languages, tools and processes used to build software systems. But most of it is essential to the software itself and trying to abstract it often abstracts away its essence [Bro87]. Software systems will always be complex and hard to reason about, but slowly we are discovering new ways to make developing them easier. Still, a single *silver bullet* does not exist.

The first computer programs ever written were so small that there was no reason for thinking about common software engineering problems. Those first programs were written in low-level code and even the first high-level languages did not have structured programming constructs or even block structures. It was only in the late 1960s that these started to become mainstream [Eds68].

As systems grew larger, the need to decompose them into smaller and more manageable pieces became stronger. This led to a new family of programming languages and a new programming style usually referred to as Procedural Oriented Programming (POP). The ability to decompose a system into procedures and functions made developing larger systems considerably easier.

In the late 1960s a new generation of languages appeared. Objects were now the most important artifacts and new features such as encapsulation, modularity, polymorphism, and inheritance were considered a solution for most programming problems. Although it was in the early 1990s that OOP became mainstream, it is still the most widely used programming paradigm.

High-level languages, POP, and OOP are all examples of advances that allowed the removal of some of the non-essential complexities of software systems without removing their fundamental characteristics. The common trait of these new advances is that they allow the decomposition of a larger problem into a set of smaller and more manageable ones.

2.2 Principles of Software Design

A clear Separation of Concerns (SoC) [Eds82] is of paramount importance in achieving a good software design. It is also considered one of the most important principles of software engineering [Rob00] and yet it is one of the most difficult to achieve. The next list highlights some of those principles:

- **Separation of concerns**. Every important concern should be considered in isolation [Eds76, Eds82]. This allows developers to concentrate on one concern at a time, abstracting the complexity of the complete system and making development easier and errors less prone to occur.
- **Low coupling**. Every module should communicate with as few others as possible [EL79]. This will prevent large ripple effects when changing the behavior of a module, easier assembly of modules and higher reusability.
- **Weak coupling**. If two modules communicate at all, they should exchange as little information as possible [EL79].
- **Information hiding**. All information about a component should be private to a component unless it is specifically declared public [Dav72]. Preventing others from accessing the internal implementation and the data representation of a module, will ensure that changes to these will not affect dependent modules (as long as the public interface is kept the same).
- **Logical cohesion**. Related components should be grouped together [EL79].
- **Open closed principle**. A module should be open for extension but closed for modification [Ber88]. This allows developers to change the behavior of a module without having to change its source code.
- **Liskov substitution principle**. A derived class may substitute a base class [BJ94]. For this to be true, preconditions of the supertype cannot be strengthened, postconditions cannot be weakened and invariants must be preserved.
- **Dependency inversion principle**. High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions [Rob96a]. This allows for the creation of modules that are more flexible, durable and reusable.
- **Stable dependencies principle**. A module should only depend on modules

that are more stable than itself [Rob97]. In this case, stable means less prone to changes.

- **Interface segregation principle**. Many client specific interfaces are better than one general purpose interface [Rob96b]. Clients of a module should only know about the specific details they care about. No module should be forced to depend on methods it does not use.
- **Law of Demeter**. Each unit should have knowledge only about closely related units [KIA88, KI89]. A module should avoid calling a method returned by another method. In other words: do not talk to strangers.

By using these principles, and in particular SoC, we will certainly be able to create better software. Software created in this way will feature the following advantages [SPB06] [1]:

- **Explicitness** The structure of each concern is fully and clearly expressed in a single module.
- **Reusability** As concerns are contained in a single module, they can be more easily reused in other programs.
- **Modularity** Since all concerns are inside modular units that encapsulate structure and behavior, the overall modularity of an application is improved.
- **Evolution** Evolution becomes easier since the implementation of changes to concerns always occurs locally within a module and saves the need to adapt existing classes.
- **Pluggability** Since all concerns are described inside a single module, they can be easily plugged in and out of an application.

In the next section, we will analyze some techniques and paradigms that are used to create modular programs.

2.3 Modular Programming

Modular programming is a programming style where different concerns of a single program are separated into different units of modularity making them easier to reason about, reuse, maintain and debug.

[1] In the original paper, this list referred to the advantages of aspect oriented programming. The list has been adapted to better follow the flow of the document.

OOP was the first real take on creating a modular programming paradigm. It is based on the idea that functions and properties should be aggregated inside programming units called classes. It provides some key concepts that help in achieving modularity like encapsulation, inheritance and polymorphism.

2.3.1 Inheritance

There are two key concepts that are often confused when talking about inheritance. Interface inheritance, also know as behavioral *subtyping*, is related to the Liskov substitution principle – *if S is a subtype of T, then objects of type T may be replaced with objects of type S* [BJ94]. It establishes an *is-a* relationship between types. Implementation inheritance, on the other hand, allows implementation reuse. Subclasses inherit their parents external interface, data members and code. Many languages embrace the two concepts making *subtyping* and inheritance a single aspect.

2.3.2 Multiple Inheritance

As the name indicates, languages that support multiple inheritance allow a class to inherit from more than one parent class. This inheritance is normally done both at the interface and implementation levels.

Multiple inheritance promotes reusability but can cause ambiguity problems when classes inherit from more than one class. This is sometimes called the *deadly diamond of death* [Mar98] and is tackled in different ways by different languages.

2.3.3 Interfaces

Some programming languages prohibit multiple inheritance altogether and use interfaces instead. Single inheritance is still allowed and classes can implement more than one interface at a time.

Interfaces are just like classes but do not provide a default implementation. This way, they remove the ambiguity problem that comes with multiple inheritance but can force developers to use the same code in different implementing classes.

2.3.4 Mixins

In classical OOP, classes must declare from which parent classes they inherit from. Mixins are a mechanism to define non-instantiable classes that will eventually extend a superclass. Using mixins, we are able to specialize the behavior of several parent classes. For this reason, mixins are also referred to as abstract subclasses [BC90].

At first look, mixins might just look like multiple inheritance but as they do not force a static inheritance chain, they can be used in different ways. For example, in re-mix, a framework that provides mixins for *C#* and *Visual Basic .NET*, mixins can be applied in the following places [Sch09]:

- A class can declaratively choose to be extended by one or more mixins.
- A mixin can declaratively choose to extend one or more classes.
- An application can declaratively choose to combine mixins and classes.
- A programmer can – at run-time – choose to combine mixins and classes.
- Any person can – at deployment time – choose to combine mixins and classes.

There is a subtle, but important, difference between the capabilities of mixins and those of multiple inheritance. By allowing the extension of classes to be declared outside the scope of the child class we make them oblivious to the final composition of the system, promoting a better SoC.

2.3.5 Traits

Traits are another inheritance mechanism very similar to mixins. A trait is also a non-instantiable class that can be used to extend the functionality of other classes. They have, however, a few key characteristics that separate them [FR04]:

- Symmetric sum: an operation that merges two disjoint traits to create a new trait.
- Override: forms a new trait by layering additional methods over an existing trait.
- Alias: creates a new trait by adding a new name for an existing method.
- Exclusion: forms a new trait by removing a method from an existing trait.

In this way, traits are also a step forward in promoting a good SoC.

2.3.6 Composition

Using composition instead of inheritance, is a technique where classes are composed together in order to achieve the same benefits of inheritance without establishing a fixed hierarchy. Some languages even go as far as not allowing inheritance altogether making composition the only way to achieve code reuse in a modular way.

This approach promotes flexibility in the face of changing requirements as the code is not fixed to a rigid, inheritance-based, structure.

2.3.7 Summary

All the techniques and language constructs that we described in the previous sections try to solve a single problem: *How can we extend and compose modular units in order to prevent code replication and have a better separation of concerns?*

For this reason, some of them seem pretty similar just in a different package. In fact, most programming languages do not use just one of these approaches but they borrow ideas from a few of them in order to create languages that tackle the concern composition problem from a different angle.

In the next section, we will discuss the main reason why the classical OOP model has failed to achieve a clear SoC.

2.4 Crosscutting concerns

The concepts behind OOP are vital for modular programming, but as they force a single decomposition criteria on the programmer they also force some concerns to get scattered and tangled throughout the code [MT02].

We can think of a software concern as any interest, which pertains to the development of the system, its operation, or any other matters that are important to one or more stakeholders [vdBCC05].

When using a modern programming paradigm, one of the problems that need to be tackled is which decomposition to choose from all the possible ones [Dav72]. The optimum solution would be to have each concern in its own unit of modularity.

When a program is modularized, following any given decomposition criteria, the concerns that do not align with that criteria end up tangled and scattered throughout several modules of the system.

These concerns are usually referred to as crosscutting concerns. Scattering occurs when a concern is spread through several modules while tangling occurs when one module contains the code for several concerns (see Figure 2.1).

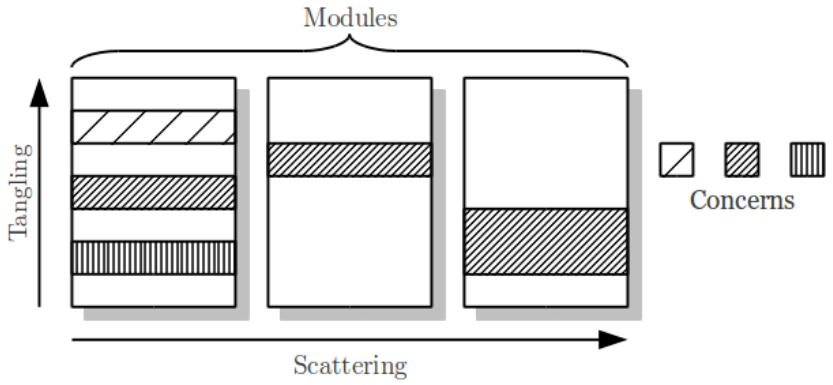

Figure 2.1: Scattering and Tangling of Concerns

Some common examples of crosscutting concerns are logging, security, instrumentation, error handling and caching, but there are many others.

It is believed that for any large enough program, having crosscutting concerns is inevitable as there will never be a decomposition criteria able to separate every concern into a different module. This problem is often referred to as the *tyranny of the dominant decomposition* [MT02]. The core concerns of the application force other concerns, often non-functional, to get scattered throughout the application code. This tangling of concerns is one of the major contributors for the increased complexity of large software applications.

The techniques we will analyze in the next sections try to improve on this paradigm by adding different composition mechanisms that promote modularity.

2.5 Other Approaches

There are many other approaches to improving the separation of concerns in software besides using objects. Several other techniques and paradigms have been

proposed throughout the time with different degrees of success and acceptance. Next we present the ones that we felt were most relevant for this work.

2.5.1 Role Oriented Programming

In Role Oriented Programming (ROP), we consider that objects can play different roles at different times and sometimes play more than one role at a time. Roles can be seen as interfaces with rights and responsibilities [ZA06].

Roles allow objects to evolve over time, taking different characteristics depending on the current context and time. Each time two classes collaborate, they are each playing a different role. These roles often fall outside the main concern of the class and should be also modeled and coded separately.

Advocates of ROP believe this is how us, as humans, perceive the world. For example, a person has a set of intrinsic characteristics. But when it plays another role, for example the role of a teacher, a new set of behaviors and characteristics emerges without replacing the characteristics that make a person a person but building upon them.

As roles act as representatives of classes, this enables them to participate in several places of the program and still maintain a separation of concerns at the code level [Gra06].

A ROP programming language must allow the definition of such roles and the assignment of roles to objects dynamically. There must also exist a mechanism that allows querying an object for a particular role. Concerns can be separated in each role and joined together as roles are assigned to objects.

2.5.2 Feature Oriented Programming

Feature Oriented Programming (FOP) is generally used in Software Product Lines (SPLs) where each feature is seen as an increment in program functionality. This concept leads to conceptually layered software designs. In FOP [Chr97], *features* are raised to first-class entities and are seen as small increments in program development or functionality.

Features are composed in a particular order and the interaction between them, for each specific composition, is defined in separate units. Instead of having method overwriting, as in OOP, FOP uses *lifters*. *Lifters* provide a mechanism to adapt the features to each possible composition. Objects can then be created simply by composing a set of features in a specific order.

2.5.3 Subject Oriented Programming

The essential characteristic of Subject Oriented Programming (SOP) is that different subjects can define and operate upon shared objects separately, without each of the subjects needing to know any details except the identity of the object. Subjects can be combined to form cooperating groups called compositions. The composition also defines a rule, the composition rule that specifies in detail how the components are to be combined [WH93].

In OOP, each class only has one single point of view in regard to the object it describes. In SOP [WH93], the state and behavior of each class are not intrinsic to itself but is the result of the composition of various subjective perceptions.

The interaction between subjects, composed together to form a *subject activation*, is determined by *composition rules*.

2.5.4 Publish and Subscribe

The Publish and Subscribe pattern, and also the Observer pattern, are well-established design techniques that allow programmers to develop objects that can send messages to other objects without knowing before hand which are those objects.

Using this pattern, developers don't need to know which objects are interested in their messages but have to decide which messages are important. For each one of these possible messages, some code has to be written inside the publisher class making it less flexible than AOP. In fact, the Observer pattern can be refactored to AOP in a very elegant way making it unnecessary to repeat its code throughout the program and keeping its logic completely separated from the main concerns [MF05b].

2.5.5 Generative Programming

Generative programming is another approach to consider. By stepping up to the meta level, a program that generates source code for another program is able to inject code in specific points of execution. We can even look into AOP as a specific case of generative programming. For example, most AspectJ compilers do a first pass where AspectJ code is transformed into pure Java code. This first step can bee seen as going from a higher level language, where concerns can be easily untangled, to a lower lever generated one.

2.5.6 Aspect Oriented Programming

Aspects encapsulate crosscutting concerns in separate units of modularity as an implementation pattern that can then be applied in several different places throughout the code.

A comprehensive review of aspect oriented programming will be done in Chapter 3.

2.6 Summary

The complexity of software development has been tackled in several different ways throughout the years. The general consensus seems to be that modularity and a clear SoC are key ingredients to make programming easier and more productive.

When some of the alternatives to objects – aspects, subjects, roles and features – are compared, it is sometimes hard to distinguish between them as they seem to revolve around the same idea of composition with just slight variations in semantic.

In the next chapter, we will explore in more detail one of these alternatives, aspect oriented programming.

Every program is a part of some other program and rarely fits.

Alan J. Perlis – 1982

3

Aspects

Contents

3.1 Key Concepts

AOP is a programming paradigm having the objective of encapsulating crosscutting concerns into separate units of modularity [GJA+97].

Just as OOP did not replace POP, but instead was built on top of it, AOP does not intend to replace OOP but rather to be an extension to it. In fact, the AOP paradigm is so generic and decoupled from OOP that it can be applied to other paradigms, including directly to POP [RD00].

Most AOP languages make use of two fundamental constructions: advices and join points. Join points are well-defined points in the control flow of a program. Each AOP language has a specific join point model that defines the different type of join points that can be specified. On the other hand, advices are pieces of code that should be executed when a certain join point is activated.

There are many programming techniques with the objective of reaching a better separation of concerns. AOP has two fundamental properties that make it different [RD00, Rob01]:

- **Quantification**. The idea that one can write unitary and separate statements that have effect in many, non-local places in a programming system.
- **Obliviousness**. The places these quantifications apply to do not have to be specifically prepared to receive these enhancements.

These two properties are fundamental to achieve a clear separation of concerns.

In a non-oblivious language, the developer would have to add some code or annotation to the base system in order for the desired concern to be invoked. Classical programming languages already have constructs for this (e.g. method calls) but using them to invoke an external concern would violate the principle of separation of concerns.

Quantification allows us to define sets of join points without knowing exactly to which code they will apply to. A good join point model will be one that will allow a great amount of expressiveness making it easier to describe these set of join points. It is also important for join points to be resilient (i.e. not easily breakable when code changes) and have a clear intention (making it easier for the reader to understand their purpose) [BR08].

As an example, let us imagine an application that logs all variable changes into a single file. With a classical programming language, this could be done by using one of many available logging libraries. As hard as we try we will find it impossible to implement this application without polluting the source code with calls to this

library in all methods that change values of variables (optimally setter methods). This means the logging concern of the application will be inextricably tangled and scattered throughout the source code.

With AOP, we could create a separate aspect with a single pointcut that captured all the join points where setter methods are called. We could then apply an advice to this pointcut that would capture the old and new value of the variables and write them to a log file. In this way, the concern is completely separated from the rest of the program and can be easily removed and even reused in other applications.

In the next section, we will give a brief introduction to AspectJ, one of the most used AOP languages, and the one we will be using throughout this work.

3.2 AspectJ

AspectJ was one of the first AOP languages to be developed. As most AOP languages, it was built as an extension to an already existent OOP language, in this case, the Java language.

AspectJ showcases a powerful join point model. It also has constructs to define sets of join points and to expose data from their execution context (pointcuts), places holding code to be run at those join points (advices), the AOP equivalent to OOP classes (aspects) and a mechanism for static crosscutting.

AspectJ is one of the most used and known AOP languages, mainly because it was one of the first and also because it is based on a very popular OOP language. There are also a lot of support tools for the language, making it a perfect candidate for AOP research.

3.2.1 Join Point Model

AspectJ sports a large number of different join points that can be targets for advices. These join points range from method and constructor calls, field reference, and assignment to exception handling and even execution of advices.

Method call join points can refer to method names, method parameters, return types, modifiers like static or private, and throws clauses, while constructor declarations omit the return type and replace the method name with the class name

(just like Java method calls and constructors). Wildcards can be used to replace any of these elements or a part of them. Listing 3.1 shows some examples of AspectJ join points and showcases the flexibility of the join point model.

Listing 3.1: Join Point Examples

```
1  // any public method call having a single parameter of type int.
2  call(public * *(int))
3
4  // execution of a private method starting with set from class
5  // Foo having a number of parameters with the last one being of type int
6  execution(private Foo.set*(.., int)).
7
8  // call to the constructor of any class whose name ends with Bar
9  // or any subclass of a class with that name pattern.
10 call(*Bar+.new())
```

3.2.2 Pointcuts

Pointcuts have two purposes in AspectJ. The first is to combine any number of join points under one single predicate (using boolean expressions). The second is to allow the exposure of the join point context.

To expose context from the join points, AspectJ uses three different predicates: *this*, *target* and *args*. The first one allows one to expose the current object, the second one to expose the target object and the last one to expose the arguments. The meaning of these keywords changes slightly depending on the type of pointcut they are used with. Table 3.1 shows the different meanings in detail.

Pointcuts can be named and have a list of arguments that can be used to expose the context of the included join points to advices. A pointcut can be either a member of a class or an aspect and can have an access modifier such as *public* or *private*. It is also possible to declare a pointcut *abstract* or to override it in subclasses or subaspects. Listing 3.2 has some examples of pointcuts and context exposure.

There are also some special pointcut predicates. The predicates *within* and *withincode* allows one to match only join points where the executing code is inside either a specific type or a specific method. The predicates *cflow* and *cflowbelow* pick join points in the control flow of any join point picked out by a specific pointcut (including or excluding that same pointcut).

Table 3.1: AspectJ Join Point Model [Asp10b]

Join Point	*this*	*target*	*args*
Method Call	executing object	target object	method args
Method Execution	executing object	executing object	method args
Constructor Call	executing object	None	constructor args
Constructor Execution	executing object	executing object	constructor args
Static Init Execution	None	None	None
Object Pre-initialization	None	None	constructor args
Object initialization	executing object	executing object	constructor args
Field reference	executing object	target object	None
Field assignment	executing object	target object	assigned value
Handler execution	executing object	executing object	caught exception
Advice execution	executing aspect	executing aspect	advice args

3.2.3 Advices

Advices contain the code to be executed when a pointcut picks up a join point. There are three types of advices: *before*, *after* and *around*.

As the name implies, *before* advices execute before the matched join point executes. This is the simplest of the three types of advices.

Unlike the *before* advice, the *after* advice can have three different forms. One can choose to execute the advice always, only if the method returns or only if the method throws an exception. The last two forms allow the capture of the returning value or the thrown exception.

Around advices run instead of the matched join point. Because of this, they must declare a return type. *Around* advices can still choose to execute the original matched join point, with the same argument values or with different ones, using the special clause *proceed*. Listing 3.3 has a small example of an *around* advice.

Listing 3.2: Pointcut Examples

```
1  // Exposing the int argument of any call to any public
2  // methods of any class having a single int argument
3  pointcut publicCall(int x): call(public *.*(int)) && args(x);
4
5  // Exposing the int argument of any call to any public
6  // methods starting with set in class Foo and also the called object.
7  pointcut setCall(Foo f, int x):
8    execution(private Foo.set*(.., int)) && this(f) && args(x);
```

3.2.4 The Aspect Construct

Aspects are a first-class artifact of AspectJ. They play the same role as classes in OOP but are different in a number of ways. Aspects can have methods and fields, just like classes, but they also have pointcuts and advices. Aspects cannot be instantiated directly, but they can have a constructor. They can also implement interfaces, extend classes and even extend other aspects.

By default, aspects are singletons. Using special clauses one can change this behavior. By defining an aspect as being *perthis* or *pertarget*, a different aspect is instantiated for each current or target object of the matched join point. An aspect defined as *percflow* or *percflowbelow* is instantiated for each flow of control of the join points picked out by the pointcut.

Listing 3.3 has an example of what a complete aspect looks like and Figure 3.1 depicts the various elements of AspectJ.

Listing 3.3: A Simple Security Example Aspect

```
1  public aspect Security {
2    // Every method starting with set and having a single int argument
3    // of class Account. Capturing the account and value.
4    pointcut setValue(Account a, int v):
5      call(public Account.set*(int)) && this(a) && args(v);
6
7    // If user has access to the account run the original join point.
8    void around(Account a, int v) : setValue(a, v) {
9          if (SecurityManager.userHasAccess(a) proceed(v);
10   }
11 }
```

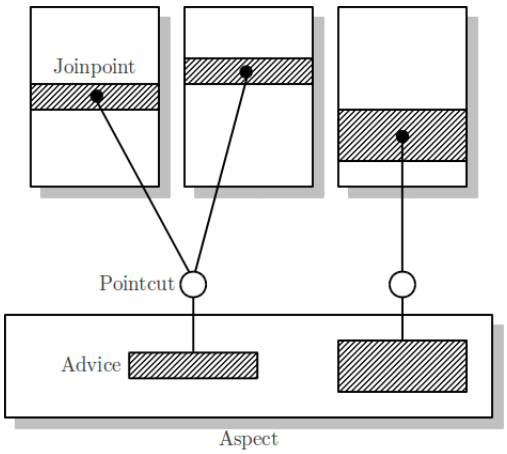

Figure 3.1: AspectJ Primary Elements

3.2.5 Precedence

In a single join point, there can be multiple advices being applied simultaneously. There are several rules that determine the precedence between these. If the two pieces of advice are defined in different aspects, then there are three possible cases [Asp10a]:

- If aspect A is matched earlier than aspect B in some declare precedence form, then all advices in concrete aspect A have precedence over all advice in concrete aspect B when they are on the same join point.
- Otherwise, if aspect A is a sub-aspect of aspect B, then all advices defined in A have precedence over all of those defined in B. So, unless otherwise specified with a *declare precedence*, advices declared in a sub-aspect have precedence over those declared in a super-aspect.
- Otherwise, if two pieces of advice are defined in two different aspects, it is undefined which one has precedence.

If the two pieces of advice are defined in the same aspect, then there are two possible cases:

- If either are after advices, then the one that appears later in the aspect has precedence over the one that appears earlier.
- Otherwise, the one that appears earlier in the aspect has precedence over the one that appears later.

These rules can lead to circularity, which will result in errors signalized by the compiler.

The mechanism behind advice precedence resolution is the following:

- At a particular join point, advices are ordered by precedence.
- Around advices controls whether advices of lower precedence will run by calling proceed. The call to proceed will run the advice with next precedence, or the computation under the join point if there is no further advices.
- Before advices can prevent advices of lower precedence from running by throwing an exception. If it returns normally, however, then the advices of the next precedence, or the computation under the join point if there are no further advice, will run.
- After returning advices will run the advice of next precedence, or the computation under the join point if there is no further advice. Then, if that computation returns normally, the body of the advice will run.
- After throwing advices will run the advice of next precedence, or the computation under the join point if there is no further advice. Then, if that computation throws an exception of an appropriate type, the body of the advice will run.
- After advices will run the advice of next precedence, or the computation under the join point if there is no further advice. Then the body of the advice will run.

3.2.6 Inter-type Declarations

Aspects can declare members (fields, methods, and constructors) that are owned by other types. These are called inter-type members. Aspects can also declare that other types implement new interfaces or extend a new class. See Listing 3.4 for some examples.

3.2.7 Summary

These last sections are just a very brief summary of everything that can be accomplished using AspectJ [Asp10a].

Listing 3.4: Inter-type Declarations

```
1  public aspect Security {
2    // Declare that the class User has a new field called username.
3    private String User.username;
4
5    // Declare that the class User has a new method called setPassword.
6    private boolean setPassword(String old, String new) {...}
7
8    // Declare that the Account class implements the Private interface.
9    declare parents: Account implements Private;
10
11   // Declare that the Account class extends the SecuredObject class.
12   declare parents: Account extends SecuredObject;
13 }
```

By introducing the notion of join points, pointcuts and advices into a widely popular programming language (Java), AspectJ has managed to make AOP more popular. These same introduced constructs transformed the language from a classical object oriented programming language to a language where obliviousness and quantification are possible.

In the next section, we show how some of these concepts can be put together to encapsulate a crosscutting concern in a separate module.

3.3 AspectJ Example

Let us start by imagining a class that represents a *person*. Let us also assume that same class has a static method that retrieves a *person* from a database and returns it. Listing 3.5 shows that method without the actual database retrieval implementation as it is not important for this particular example.

Listing 3.5: Person Class

```
1  public class Person {
2          public static Person getPerson(int id) {
3                  Person person = getPersonFromDatabase(id);
4                  return person;
5          }
6  }
```

A common addition to such a method would be to add a *caching* mechanism. By doing so, we are preventing multiple calls to this method with the same id from

having to execute a possibly expensive operation, as is querying a database, by sacrificing some memory. The problem with the implementation shown in Listing 3.6 is that two concerns, the retrieval of a *person* from the database and *caching*, are tangled inside the same method.

Listing 3.6: Person Class with Caching

```
1  public class Person {
2          private static HashMap<Integer, Person> peopleCache;
3          public static Person getPerson(int id) {
4                  if (people.containsKey(new Integer(id)))
5                          return peopleCache.get(new Integer(id));
6                  Person person = getPersonFromDatabase(id);
7                  peopleCache.put(new Integer(id), person);
8                  return person;
9          }
10 }
```

The AOP way of preventing this tangling is to define a separate aspect designed exclusively for the caching concern. Listing 3.7 shows how this could be achieved. By creating a pointcut that captures all join points where the *getPerson* method is called, we can then apply an *around* advice to this pointcut. This advice can then verify if the person with the desired id has already been retrieved from the database and if not proceed with the method call.

Listing 3.7: Caching Aspect

```
1  public aspect PersonCaching {
2          private static HashMap<Integer, Person> peopleCache;
3
4          pointcut getPerson(int id) : call(public Person getPerson(int)) && args(id);
5
6          Person around(int id) : getPerson(id) {
7                  if (peopleCache.containsKey(new Integer(id)))
8                          return people.get(new Integer(id));
9                  else return proceed(id);
10         }
11 }
```

In this small example, we have seen how AspectJ can help to separate concerns into their own units of modularity. In the next couple of sections, other approaches to AOP will be shown as well as alternative techniques to AOP as far as separation of concerns goes.

3.4 Alternative Approaches to AOP

In the previous sections, we have seen the most common approach used in AOP languages. However, there are several other techniques that can be used with similar or even better results. In this section, we will analyze some of these alternative approaches.

3.4.1 Composition Filters

Composition Filters (CF) [BA92, AT98] use two first class elements to define crosscutting concerns: *Concerns*, which are used to define the primary behavior, and *Filters* that are used to extend that behavior.

Filters can be composed around objects in a modular way. Their primary objective is to intercept messages passed to the object and either accept them or reject them. Depending on the filter type these two actions can have different results. For example, when an *Error Filter* accepts a message it will pass it to the next filter unchanged, and if it rejects the message it will raise an error. A *Dispatch Filter* will delegate the message to an internal message if it is accepted or pass it to the next filter if it is rejected.

An example of an *Error Filter* written in *Sina*, a CF language, can be seen in Listing 3.8 [Koo95]. This example is just a snippet of class defined in this language that implements a simple *stack*.

Listing 3.8: Error Filter Example

```
1 stackEmpty : Error = { stackNotEmpty => {pop, top}, true ~> {pop, top} };
```

In this example, *stackNotEmpty* is a condition defined previously in the class. The initializer of the first *input filter*, the error filter named *stackEmpty*, specifies that the filter accepts messages with the selector *pop* or *top* when the stack is not empty. It could be read as “if stackNotEmpty then accept a message with (pop or top)”.

The second filter element could be interpreted as “if true then accept a message not with (pop or top)”, which is the same as “accept a message without pop and without top”. As a result, the error filter rejects messages with selector pop or top when the stack is empty, causing it to generate an error, and accept messages otherwise.

By creating a language-independent specification, based on filters, on top of the classical OOP model, CFs provide a mechanism to separate crosscutting concerns from the primary concerns.

3.4.2 Hyperslices

Hyperslices [PHWSMS99] are a set of conventional modules containing units that pertain to only one single concern. A system is written as a collection of hyperslices that can be composed using hypermodules.

A hypermodule is a set of hyperslices composed using a *composition rule* to create a new hyperslice. As hypermodules are also hyperslices, these can be nested at will to create the composed system.

A simple example written in *Hyper/J*, a *Java* based hyperslices language, of how hyperslices can be used can be seen in Listing 3.9 [Fai04]. In this example, we must considerer that two classes called *Foo* and *Bar* have been defined in their own separate files. Each one of these classes has one method only. The method *hi* has been declared in the *Foo* class and the method *bye* has been declared in the *Bar* class. Concerns are then declared for each one of these methods and a hypermodule is created that composes these concerns together.

Listing 3.9: Hyperslices Example

```
1  -hyperspace
2    hyperspace demo
3    composable class Foo;
4    composable class Bar;
5
6  -concerns
7    class Foo : Feature.hi
8    class Bar : Feature.bye
9
10 -hypermodules
11   hypermodule DemoHM
12     hyperslices: Feature.hi, Feature.bye;
13     relationships:
14       mergeByName;
15       equate operation Feature.hi.sayHi,
16                        Feature.bye.sayBye
17         into greet;
18     merge class Feature.hi.Foo, Feature.bye.Bar;
19   end hypermodule;
```

Contrary to most AOP approaches, there is no separation between base code and crosscutting concerns when using hyperslices.

3.4.3 Event Based AOP

While a classic AOP language like AspectJ is based on the capture of join points by pointcuts, Event Based AOP (EAOP) is based on the monitoring of execution events [ROM01].

Events are very similar to AspectJ join points as they also represent the execution points of a program but instead of being captured by pointcuts, sequences of events are analyzed by a monitor and matched against selected event sequences. The use of a monitor instead of pointcuts being matched in a pre-processor, allows greater flexibility, a better composition of aspects and dynamic instantiation of aspects. Besides this, it makes it easier to define aspects from a formal point of view [ROM01].

3.5 Aspect Oriented Software Development

A large number of software development approaches exist. Nevertheless, most of them follow the same development lifecycle. Aspect Oriented Software Development (AOSD), although not a software development methodology *per se*, can be used in each of the various activities of the development process. AOSD adds to the process with the inclusion of new tools and languages, but, at the same time, some activities must be changed to accommodate this new approach. For example, documentation and testing techniques must evolve to cope with the new concepts and language constructs brought by AOP. The following sections summarize some of the existing approaches, each focusing a particular phase of the lifecycle.

3.5.1 Requirements Analysis

The requirements phase of the software development cycle is one that can benefit with the introduction of the AOP approach. Grundy [Gru99] states that traditional requirements capturing techniques are not powerful enough to describe

component requirements, leading to less reusable components. This study lead to a new requirements specification methodology named Aspect Oriented Component Requirements Engineering (AOCRE).

Rashid [RMA03] rationalized that SoC issues were a real problem even in the requirement analysis phase of the software development cycle, leading to another new requirement specification paradigm: Aspect Oriented Requirements Engineering (AORE). Using aspects in this early phase of the process, Early Aspects [RMA+10], allows the identification of conflicts between crosscutting concerns earlier and at the same time it helps achieve a better traceability of system-wide requirements throughout the development process. Using aspects in this phase also ensures better homogeneity in an aspect oriented software development process. A new extension to the Unified Modeling Language (UML) notation has also been developed to support these new ideas [AMBR02].

Sutton [SMSR02] presented Cosmos, a software concern-space modeling schema. In this proposed schema, the author separates the notions of concerns, relationships and predicates. Concerns are classified as *logical* and *physical*. Logical concerns are further typed as *classifications*, *classes*, *instances*, *properties*, and *topics*; physical concerns as *collections*, *instances*, and *attributes*. Relationships are classified as *categorical*, *interpretive*, *physical*, and *mapping*. Predicates apply to concerns and relationships and reflect consistency considerations.

A promising approach has been introduced by Baniassad [BC04] and Clarke [CB05] with their *Theme Process*. This process is composed by two separate but related methodologies: Theme/Doc (that allows users to identify aspects in a set of requirements) and Theme/UML (how to model them in UML style designs).

Araújo [AWK04] proposed an approach to model scenario-based requirements using aspect oriented principles. His approach used Interaction Pattern Specifications (IPSs) to model aspectual scenarios. He also showed how these aspectual scenarios could be later composed with non-aspectual scenarios and transformed into executable state machines.

3.5.2 Design

The second phase of most software development methodologies is the high-level design phase. The requirements captured in the Requirement Analysis phase are

transformed into modules, classes, and their interactions. Once again in this phase, the shift to AOSD brings some new challenges.

Specification

The most obvious problem found in this phase is how to adapt current architecture specification methodologies to cope with the new elements and ideas introduced by AOP. Several approaches to this problem exist, and not surprisingly most of them are based in the existing UML extension mechanisms, like the use of *stereotypes*.

Suzuki [SY99] states that crosscutting problems are very often only found in the construction phase of the development cycle. Developers usually deal with the problems found by adding aspects manually at that stage. This happens mainly due to the lack of aspect oriented tools focused in this phase of the development cycle. The same author listed some of the advantages of incorporating aspects earlier, in the design phase:

- **Documentation and Learning.** By visualizing aspects early in the design phase, developers can better understand how they interact in a more intuitive way. Also, this results in the early documentation of aspect usage.
- **Reuse of Aspects.** The documentation of aspects in the design phase will allow the reuse of the aspectized components in different projects making it possible to create aspect libraries.
- **Round-trip Development.** Incremental development is a common development strategy, where the various phases of the development process are repeated in order to fine tune any design flaws encountered during any of the phases. By adding aspects in the construction phase of the development process, developers are compromising the chance of going back to the design phase and change the system architecture.

In order to allow early aspect oriented system designs, Suzuki [SY99] proposed extensions to the current UML diagrams supporting the design phase of the development process. In these new extensions, aspects would be represented as classes with an *aspect* stereotype. This is an obvious solution as aspects are much like classes, as they also have methods and attributes. The operation list compartment of the aspect would then show each weaving of the aspect as operations with the

weave stereotype attached and a classifier to show which classes, methods and variables are affected by it.

There are three types of relationships possible between classes in the UML notation: Association, Generalization, and Dependency. The same author states that, the kind of relationship between classes and aspects is better suited for a Dependency relationship, or, more precisely an Abstract Dependency relationship with a *realize* stereotype attached to it. Woven classes, the virtual classes that are the result of the weaving process, could then be represented simply as classes with the *woven class* stereotype.

Aldawud [AEB01] has a similar, but more simplistic, approach using stereotypes to mark classes as being aspects, and associations between aspects and classes as having the *control* stereotype meaning that an aspect controls in some way that particular class.

Kandé [KKS02] proposed an alternative to this notation, with pointcuts being represented as new separate elements and adding an advice block to the aspect elements. New notations to represent AOP features, like the multiplicity of aspects, have also been proposed. The same author also presented some modifications to the collaborative diagrams of the UML in order to represent when join points are reached.

Ho [HPJP00] has a different approach to the problem by using annotations and stereotypes as guides to the weaving process. For example, a class that should be made persistent could be marked as *persistent* and the weaving process would know which aspects had to be weaved in order to accomplish this.

Another different approach, using stereotypes to represent crosscutting concerns, advices and introductions (inter-type declarations), has been presented by Stein [SHU02]. This work also proposes the use of UML interaction diagrams to represent join points and collaboration diagrams to show how aspects interact with other units.

The Theme approach [CB05], already referred in the previous section, also allows the modeling of aspects in UML style diagrams.

A complete semantics for specifying pointcuts in UML diagrams has also been detailed in [PDF+02].

Design Patterns

Design patterns are well-known generic solutions to commonly recurrent problems. Patterns gained popularity in the software engineering community after the publication of the famous Design Patterns book by the The Gang of Four (GoF) [GHJV94]. The advent of AOP is a great opportunity to redefine patterns in a more modular form.

Hannemann [HK02] has shown how Design Patterns could be mapped as aspects. The same author has described some of the advantages of using AOP to implement Design Patterns:

- **Locality.** All code implementing the patterns is local to the pattern itself. None of the related classes is changed in the process.
- **Reusability.** The pattern code can be reused throughout an application only by implementing a single concrete aspect (see Listing 3.10).
- **CompositionTransparency.** If a class becomes involved in more than one Design Pattern (even in patterns of the same kind), each pattern can be reasoned about independently.
- **(Un)pluggability.** Adding and removing patterns becomes as simple as removing the implementing aspect from the system.

Clarke and Kande [CW01, KC03] have also written about how patterns could be implemented using AOP. Interestingly, both used the UML annotation extensions for AOP they proposed (see Section 3.5.2).

Garcia [GSF+05] has made an interesting study, comparing the implementations of all 23 Patterns proposed by the GoF in both OOP and AOP. This assessment study has shown that in many cases the AOP version of the patterns provided a better SoC, better reusability and needed lesser number of lines of code.

3.5.3 Construction

The construction phase of the development cycle is when the actual code is written. As AOP has several implications in this phase, several issues have been raised, namely: reusability, conflicts between aspects and accidental pointcuts.

Reusability

Achieving better reusability has always been one of the major goals of software engineering. The use of reusable modules reduces the implementation time and ensures that the used code has already been thoroughly tested and documented. As has been shown by Garcia [GSF+05], AOP allows developers to achieve better SoC in software applications, therefore better modularity and easier reusability is attained.

Listing 3.10 shows how the *Observer pattern* can be implemented using AOP aiming for reusability. In this example, a generic *Observer pattern* aspect was created. This aspect, created as an abstract aspect, specifies two new interfaces: *Observer* and *Subject*. At this point, these interfaces are not implemented by any class. Notice that this aspect is declared as being *perthis(subjectConstructed(Subject))*, meaning that one instance of this aspect exists for each *Subject* object. With this, one instance of the *observers* vector is created for each *Subject*. Then, the *addObserver* and *removeObserver* methods were created, as well as the abstract pointcut *subjectChanged*. An advice connected to this pointcut was also created. This advice will call the abstract method *updateObserver* for each *Observer* in the *observers* vector. Notice that the *subjectChanged* method captures the context in order to the advice to access the *observers* of the correct *Subject*. As this aspect never refers to the *Screen* or *Point* classes, it can be reused in the same software application or in other systems.

Binding this aspect to the correct classes is done by extending this aspect into a concrete aspect called *PointObserverPattern*. This aspect uses the declare clause to specify that the *Point* class implements the *Subject* interface and that the *Screen* class implements the *Observer* interface. This aspect defines also the concrete implementations of the *subjectChanged* pointcut and the *updateObserver* method making it possible to describe what events should be observed and what to do when these events occur. This example shows how suited AOP is to implement modular and reusable crosscutting concerns.

Listing 3.10: Reusable Observer Pattern (AOP)

```
1 public abstract aspect ObserverPattern : pertarget(Subject s)
2 {
3         public interface Observer { }
4         public interface Subject { }
5
6         Vector observers = new Vector();
```

```
7
8           public void addObserver(Observer o)
9           {
10                  observers.add(o);
11          }
12
13          abstract protected pointcut subjectChanged(Subject s);
14
15          after(Subject p) : subjectChanged(p)
16          {
17                  Iterator itr = observers.iterator();
18                  while (itr.hasNext())
19                          ((Observer)itr.next).updateObserver();
20          }
21
22          public abstract void updateObserver(Observer o, Subject s);
23  }
24
25  public aspect PointObserverPattern extends ObserverPattern
26  {
27          declare parents: Screen implements Observer;
28          declare parents: Point implements Subject;
29
30          pointcut subjectChanged(Subject s)
31          {
32                  call(void Point.set*(..) && target(s));
33          }
34
35          public void updateObserver(Observer o, Subject s)
36          {
37                  (Screen(o)).updateDisplay());
38          }
39  }
```

Hanenberg [HU01] introduced some interesting rules on how to use aspects to achieve reusability:

- **Separated pointcut declarations**. Whenever a new aspect is created, a corresponding abstract super-aspect has to be implemented that contains all pointcuts declarations and definitions needed by the aspect. Advices in the sub-aspect refer to the pointcuts defined in the super-aspect.
- **No pointcut for more than one advice**. If one pointcut is used for more than one advice, there is no possibility to adapt the behavior of a single advice.
- **Concrete aspects are always empty**. Once an advice is within a concrete aspect, it becomes lost for any further reuse. In this way, a concrete aspect

should not contain any pointcut definition (besides abstract ones). This guarantees the possibility to redefine the pointcuts in a concrete aspect.

Refactoring

Refactoring is the process of rewriting a computer program, or other material, to improve its structure, or readability, while explicitly preserving its meaning and behavior. Several common methods for refactoring have been detailed in [FBB+99].

With AOP, new methods allowing the refactoring of existing code into this new paradigm have been described in [Mon05, MF05a, MF05b]. Using these methods, it is possible to take an original OOP code and untangle it safely, turning it into AOP code. Unit tests can then be used to allow developers to refactor code making sure modules still behave correctly [Bin99].

Debugging

Debugging has been an issue with AOP. When debugging AOP code, most frameworks use the result of the weaving process instead of the original aspect and class implementations. This may make debugging harder for AOP developers. Tools should improve by showing crosscutting structures, like thread trees that hide generated calls, and giving the ability to set breakpoints on pointcuts. [MR02] has an interesting approach both to the problem of debugging as well as to the related problem of profiling.

On the other hand, AOP can be used as a powerful debugging tool. By instrumenting an application using AOP techniques, we can achieve extremely powerful debugging tools as is the case of TOD, an omniscient debugging tool based on weaving [PTP07].

Development Environments

Before a new paradigm becomes ready for wide use and is accepted by industry, good development environments have to emerge. Several of these environments are already available for AOP. As AspectJ is currently the AOP leader language, it was expected that the first environments to appear would support this particular

language. AspectJ is an extension of the Java language while Eclipse [Ecl10] is a successful open-source development environment, for that same language, that already had features like refactoring and unit testing. So, it was no surprise that the first good AOP development environments, AspectJ Development Tools (AJDT), appeared as extensions to the Eclipse IDE [AJD10].

3.5.4 Testing and Validation

Testing and validation routines are important aspects of the software development cycle because no development paradigm ensures code correctness *per se*.

In AOP, the main element is the aspect. Aspects differ significantly from classes and procedures. Testing AOP should not only test if the aspect performs as expected, but also if the classes modified by them continue to work correctly [Zha02].

Zhao [Zha02, Zha03] proposed three levels of testing for aspect-based code:

- **Intra-module testing.** Testing each individual element, such as advices methods and introductions.
- **Inter-module testing.** Testing a public module along other modules it calls without considering invocations from other modules outside the aspect or class.
- **Intra-aspect testing.** Testing the interaction between the aspect and multiple modules when they are called in a random sequence from outside the aspect.

3.5.5 Code Documentation

Code documentation is another important step of the development cycle. With the introduction of the new AOP elements, code documentation must be rethought. AspectJ already offers AOP oriented documentation features such as the *ajdoc* documentation tool [Asp15].

Besides AOP code documentation, another interesting possibility is to incorporate the AOP paradigm ideas in the code documentation process. In this way, docu-

mentation snippets could be thought as being documentation aspects that could be weaved together and, in this way, be composed into a complete document.

3.6 Key Research Issues

In this section, we will discuss some key research issues that are relevant to the work being presented.

3.6.1 Fragile Point Cuts

Designing pointcuts that truly capture the intention of the developer is difficult. Instead of that, pointcuts often rely on the code structure of the base program. This makes pointcut expressions not capable of remaining valid as the base code evolves.

> "The fragile pointcut problem occurs in aspect oriented systems when pointcuts unintentionally capture or miss particular join points as a consequence of their fragility with respect to seemingly safe modifications to the base program" [AKJ06].

For example, stating that we want to capture join points in all methods that modify the internal state of an object, would be better than saying that we want to capture all methods starting with *set*. The latter alternative would be prone to errors if, for example, we added a new method to the class named *setup*.

Enumerating all the desired join points is a bad idea because we would have to continuously add new join points as new methods are added to the base code. Using wildcards to circumvent this problem can cause problems if the pointcut language is not as intentional as it should.

3.6.2 Interferences

One of the most criticized aspects of glsaop is that when multiple concerns are woven at the same join point they can easily interfere with each other. We can

think of an interference as an unexpected interaction that causes one, or more, of the woven concerns to have an undesirable behavior.

In Section 5.3 we will analyze this issue more thoroughly and discuss some of the work being done to mitigate it.

3.7 Summary

Achieving a perfect separation of concerns in software development is something that has been eluding developers for many years. The general consensus is that there will never be a perfect solution. All we have are a diversity of tools that tackle specific difficulties.

AOP is another great tool that allows developers to write reusable and modular code. There are however several issues that still have to be polished, either by changing the way in which we use the idea of AOP, or by creating new tools that help developers better control the extreme power behind it.

In the next chapter, we will talk about testing, in general, and in Chapter 5 we will discuss the particular problems inherent to the testing of aspects.

Testing leads to failure, and failure leads to understanding.

Burt Rutan

4

Testing

Contents

Software testing is one of the core activities of software development. It is the process in which stakeholders assess the quality of the software product being developed.

Testing can be seen in two different perspectives. As a way to improve the confidence of the stakeholders in the code or as a way to find errors in order to correct them [Mye79]. This might seem a subtle difference, but if your goal when creating a test is to try to prove that the code has errors, you will use inputs that you know have a higher probability of breaking the code. A successful test is one that reveals a failure.

In this chapter, we will take a quick glance into the world of software testing starting by understanding the importance of testing followed by an explanation of the testing process itself.

4.1 The Importance of Software Testing

For the most part of it, software is written by humans, and humans are known to make mistakes; sometimes lots of them.

Most of the times, when a program fails to perform its function, there are no catastrophic consequences to that event. At most you will lose a day of hard work if you forgot to save your documents. However, it is estimated that software bugs have an annual impact close to $60 billion in the United States economy and a third of this value could be avoided with better testing [Tas02].

But sometimes, the aftermath of a software bug is indeed catastrophic. Besides the software that we normally use in our homes and work, some software is used in hospitals, banks, rocket launching and many other activities where a single software bug can cost lives and lots of funds and resources.

4.2 Types of Tests

The simplest way to categorize tests is to divide them into the black box and white box approaches [Whi87]:

- **White Box.** In white box tests, the inner workings of the code are taken into consideration when selection the test data to be used. This approach is useful when trying to determine if there are any significant errors in the code not directly caused by the misinterpretation of the requirements or specification. White box tests are also known as *structural* testing.
- **Black Box.** In black box tests, only the external interfaces are exercised in order to find if they comply with the requirements and specification. No knowledge of the inner working of the code is used. Black box tests are also known as *functional* testing.

4.3 Testing Levels

Software testing can be done in several different levels according to the software development activity they are used in.

4.3.1 Unit Testing

Unit testing is a testing process where small units of code are tested in isolation [Bec94]. Since attention is focused on smaller units of the program, unit testing makes testing complex software systems easier. It is also easier to pinpoint the cause of an error, since, when an error is found, it is known to exist in a particular unit [Mye79].

Unit testing is used during the implementation activity of the software cycle, the phase where the code is actually produced. It is the lowest level of testing and the one that developers are more familiar with.

Unit tests are very often written by the same developers writing the code, usually at the function or class level. This is done to assure that each piece of code is working as expected. However, these tests cannot make any assurances about the behavior of the complete system.

A perfect unit test would be one that tests all possible inputs to a function and correctly tests any side effects produced while being shielded from code from other units.

Of course, testing all possible inputs is usually impossible, so developers must carefully choose which inputs they want to test the unit with. The chosen inputs should cover all possible code branches [Whi87] and all edge and corner cases. This is not an easy task and several techniques, manual and automated, have been developed to aid in selecting test cases.

It is also important for the tests to be correct. For this to happen, tests should be written by someone that did not write the code or at least before writing the code. This will prevent mistakes induced by a form of confirmation bias. It is also common for test data to be automatically generated using, for example, a slower but less error prone version of the code.

Unit testing is usually done using a white box approach.

Stubs, Mocks, Dummies and Fakes

As most units depend on other units, to assure that tests are run in isolation, test doubles have to be used. Test doubles are pretend-objects that are used in place of objects that the tested unit depends upon but are not part of the test.

There are several types of these objects [Fow07]:

- **Dummy** objects are objects that are passed around but are never used. These are usually just objects that fill parameter lists as the code would not compile without them.
- **Fake** objects are objects that work as intended but take shortcuts in the implementation making them more suitable for testing. Either because they are less prone to errors, faster or they have an easier setup.
- **Stubs** are only able to provide canned answers to the calls that are made during the test. They do not have an actual working implementation and sometimes can record information about calls.
- **Mocks** are pre-programmed objects with expectations. When using mocks, the tester starts by defining which calls are expected to be made by the tested unit. Unlike the other types of doubles, mocks can do behavior verifications while the other are only good to do state verifications.

4.3.2 Integration Testing

Integration testing is the phase in which previously tested units are aggregated into larger groups that are tested together. Unit testing is useful to find local faults in programs but does not exercise the relations between units.

Several different strategies can be used to choose the order in which units are grouped together:

- In the big bang strategy, a large group of modules is grouped together into the complete software system or into a major part of it. Real usage scenarios are then used to flush any problems created by the interaction of the previously tested units.

- The top-down starts by testing modules that have no other modules depending on them. This forces the creation of a lot of code doubles in order to test modules that depend on modules that have not been integrated yet. The testing continues by replacing those doubles with the actual implementation.
- On the other hand, the bottom-up strategy starts by testing modules that do not depend upon any other ones. The testing continues by climbing up the dependency graph until the complete system is tested.

Integration testing is normally done using a mix of black box and white box testing.

4.3.3 System Testing

In the system testing phase, the complete system is tested all at once. There is no more need for code doubles. The behavior of the system is compared to the one present in the specification in order to uncover faults. The software is run on top of the final hardware and focus on final details like usability, load, security, installation and compatibility.

4.3.4 Acceptance Testing

Acceptance tests are very similar to system tests. The main difference is that they are normally performed by the client or by the final users of the product. In this way, technical aspects are less important and feature completion and correctness is more valued.

4.3.5 Regression Testing

The objective behind regression testing is to uncover faults that might be introduced by changes to the system. These faults can be new ones, or faults already fixed. In this way, it is considered important that tests exist for all faults discovered and fixed previously.

4.4 Summary

In this chapter, we have talked about the extreme importance of testing in software development. We also described several strategies and the usage of tests in the different phases of the development cycle.

When tests are done to the complete system, like in system and acceptance tests, there is no real big difference between testing OOP or AOP code. But when it comes to testing smaller parts of the system, the nature of AOP gets in the way making the promise of a good separation of concerns harder to achieve; not at the code level but at the also very important testing level.

To truly have reusable software components, we have to consider tests as an integral part of those same components. In the next chapter, we will see how AOP is not suitable for that goal. At least not without some help.

Part II

Problem and Solution

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald E. Knuth

5

Unit Testing Aspects

Contents

Most current software developers regard testing as an important part of their development process. In many development technologies, it is the first stepping stone to developing top quality software [Mye79].

In this chapter, we will discuss some issues that arise when we try to apply testing techniques, and in particular unit testing, to an AOP language like AspectJ.

With AOP, we are aiming at having each concern in its own module and with unit testing one tries to create tests that are particular to one unit at a time.

Ideally, one should be able to create unit tests for each concern that would also be self-contained and have no traces of other crosscutting concerns.

5.1 Motivational Example

In this section, we will describe a simple example, written in AspectJ, that we believe illustrates the problem we are trying to solve.

For simplicity sake, we will use a very simple and stripped down example of a banking system. This system will have only two classes: A *Person* class and an *Account* class. We will start by explaining how these two classes work and interact.

5.1.1 Base System

In this simple banking system, a person will only have two attributes: a *name* and an *id*. Listing 5.1 shows the Java code used to implement this class.

Listing 5.1: Banking System – Person Class

```
1  public class Person {
2    private int id;
3    private String name;
4
5    public Person(String name, int id) {
6      this.name = name;
7      this.id = id;
8    }
9
10   public String getName() {
11     return name;
12   }
13
14   public int getId() {
15     return id;
16   }
17 }
```

Accounts were also kept to a minimum specification. Besides having an *id* and its current balance, each account has only one owner. The most complicated feature of this class is its capability of transferring money to other accounts. Listing 5.2 shows the Java code used to implement this class.

Listing 5.2: Banking System – Account Class

```
1  public class Account {
2    private Person owner;
3    private int id;
4    private double balance;
5
6    public Account(Person owner, int id, double balance) {
7      this.owner = owner;
8      this.id = id;
9      this.balance = balance;
10   }
11
12   public Person getOwner() {
13     return owner;
14   }
15
16   public int getId() {
17     return id;
18   }
19
20   public void transfer(double ammount, Account otherAccount)
21       throws Exception {
22     if (ammount <= 0)
23       throw new Exception("Can't transfer a negative value");
24     otherAccount.setBalance(otherAccount.getBalance() + ammount);
25     setBalance(getBalance() - ammount);
26   }
27
28   public void setBalance(double balance) {
29     this.balance = balance;
30   }
31
32   public double getBalance() {
33     return balance;
34   }
35 }
```

5.1.2 Unit Tests

We also created some very simple unit tests for both classes. Listing 5.3 shows the Java code used to test the *Person* class. These unit tests only verify if the *id* and *name* of the person are stored correctly after a *Person* object is created and if they can be retrieved correctly.

A different set of unit tests have been created to test the *Account* class. These test if the *owner* of the account and balance are set correctly after an *Account* object is created, if they can be retrieved correctly, and if transfers work as they are supposed to. For simplicity sake we did not test any error conditions. Listing 5.4

Listing 5.3: Banking System – Person Class Tests

```
1  import junit.framework.TestCase;
2
3  public class TestPerson extends TestCase {
4    public void testName() {
5      Person p = new Person("John", 1);
6      assertEquals("John", p.getName());
7    }
8
9    public void testId() {
10     Person p = new Person("John", 1);
11     assertEquals(1, p.getId());
12   }
13 }
```

shows the Java code used to test the *Account* class.

5.1.3 Separating Concerns

Despite the obvious shortcomings of the example, a classical object oriented programming developer would be quite happy with the achieved design. He would say that the *account* and *person* concerns were clearly separated into their own modules. Even the unit tests would be to his satisfaction as each unit test only tests the behavior of one class.

A more thorough inspection of the code will show that one concern, which we can call *each account has one owner*, is tangled with the other account concerns. This will make it harder to evolve the system if later on we decide that each account can have several owners or if we want to reuse the *Account* class in a different setup where accounts have several owners.

Separating this concern into a different module can be easily achieved with some AspectJ code. Listing 5.5 shows the code of the *AccountOwner* aspect where inter-type declarations are used to implement the concern we want to remove from the *Account* class.

To improve the modularity of the system, the tests should also be separated. This can be achieved by removing all references to the *Person* class from the account tests and moving the account owner test to a separate file containing tests for the *AccountOwner* aspect. Listing 5.6 shows the *AccountOwner* tests.

Listing 5.4: Banking System – Account Class Tests

```
1  import junit.framework.TestCase;
2
3  public class TestAccount extends TestCase {
4
5    public void testOwner() {
6      Person p = new Person("Jonh", 1);
7      Account a = new Account(p, 1, 0);
8      assertEquals(p, a.getOwner());
9    }
10
11   public void testBalance() {
12     Person p = new Person("Jonh", 1);
13     Account a = new Account(p, 1, 1000);
14     assertEquals(1000.0, a.getBalance());
15   }
16
17   public void testTransfer() throws Exception {
18     Person p1 = new Person("Jonh", 1);
19     Account a1 = new Account(p1, 1, 1000);
20
21     Person p2 = new Person("Mary", 2);
22     Account a2 = new Account(p2, 2, 500);
23
24     assertEquals(1000.0, a1.getBalance());
25     assertEquals(500.0, a2.getBalance());
26
27     a1.transfer(200, a2);
28
29     assertEquals(800.0, a1.getBalance());
30     assertEquals(700.0, a2.getBalance());
31   }
32 }
```

5.1.4 Adding Authentication and Security

To prove our point, we are going to add two more modules into the system. The first one will allow persons to authenticate by adding a *login* and a *password* to each person. We will also provide an extended constructor to the *Person* class and implement methods for logging in into the system and verifying who is the current user. Listing 5.7 shows the *Authentication* aspect.

The *Security* aspect will use the data provided by the *Authentication* to infer if the current user can execute operations like transferring money from one account to another. This will be accomplished by capturing the *transfer* method join point of the *Account* class and, using an *around* advice, verify if the user is owner of that account.

Listing 5.5: Banking System – AccountOwner Aspect

```
1  public aspect AccountOwner {
2    private Person Account.owner;
3
4    public Person Account.getOwner() {
5      return owner;
6    }
7
8    public Account.new(Person owner, int id, double balance) {
9      this(id, balance);
10     this.owner = owner;
11   }
12 }
```

Listing 5.6: Banking System – AccountOwner Tests

```
1  import junit.framework.TestCase;
2
3  public class AccountOwnerTest extends TestCase {
4    public void testOwner() {
5      Person p = new Person("Jonh", 1);
6      Account a = new Account(p, 1, 0);
7      assertEquals(p, a.getOwner());
8    }
9  }
```

In order not to overcomplicate this example, we are not going to provide the code used to test these two new modules. The complete code, with separate modules for each concern and separate test cases for each module, would have the set of dependencies that can be observed in Figure 5.1.

5.2 Research Problem

In the previous section, we showed how we can use unit tests in AspectJ while keeping concerns separated even at the testing level. The problem with this approach is not obvious at first sight but can be easily spotted if we actually run the tests. The *Security* aspect changes the *Account* class behavior in such a way that it breaks its tests. The *testTransfer* test will start throwing an *exception* because the owner of the account is not logged into the system. This means that the system can no longer be easily tested. There are several naive approaches we can use to solve this problem but, as we will see in the next sections, none of them

Listing 5.7: Banking System – Authentication Aspect

```
1  public aspect Authentication {
2    private String Person.login;
3    private String Person.password;
4
5    private static HashMap<String, Person> people
6      = new HashMap<String, Person>();
7    private static Person currentUser = null;
8
9    private Person.new(String name, int id,
10       String login, String password) {
11     this(name, id);
12     this.login = login;
13     this.password = password;
14     people.put(login, this);
15   }
16
17   public String Person.getPassword() {
18     return password;
19   }
20
21   public Person getCurrentUser() {
22     return currentUser;
23   }
24
25   public boolean login(String login, String password) {
26     if (!people.containsValue(login)) return false;
27     if (people.get(login).getPassword().equals(password)) {
28       currentUser = people.get(login);
29       return true;
30     }
31     return false;
32   }
33 }
```

is perfect.

5.2.1 Moving the Test

The most simple solution would be to move the offending test from the *account* module to the *security* module and changing it to accommodate the *security* concern. This could be easily achieved by adding some code to the test in order to login the correct user before performing the transfers.

The problem with this approach is that the *account* module would lose the *testTransfer* test. This would make it harder to reuse this module in other systems.

Listing 5.8: Banking System – Security Aspect

```
1  public aspect Security {
2    pointcut transfer(Account a) :
3      execution(public void Account.transfer(..)) && target(a);
4
5    void around(Account a) throws Exception : transfer(a) {
6      Person owner = a.getOwner();
7      Person currentUser =
8        Authentication.aspectOf().getCurrentUser();
9      if (owner.equals(currentUser))
10       proceed(a);
11     else throw new Exception("Operation not allowed");
12   }
13 }
```

At the same time, the basic functionality of transferring money between accounts would no longer be tested individually.

5.2.2 Changing the Test

By changing the *accountTransfer* test to accommodate the changes introduced by the security aspect, we could easily make it work again. This would also have the effect of making the *security* and *account* concerns tangled with each other.

This would happen not at the working code level but at the testing level. Although it might seem to be just a small problem, this would prevent the *account* module from being easily reused alone in other systems. It would also mean that the basic functionality of transferring money between accounts would no longer be tested individually.

5.2.3 Using AOP to Change the Test

We could keep the account module code as it is and use the AspectJ capabilities to change the test by only changing some code in the *security* module.

This could be achieved, for example, by adding an aspect to the *security* module, that captured the *testTransfer* method join point and, with an around advice, recreated it having the *security* module concerns into account.

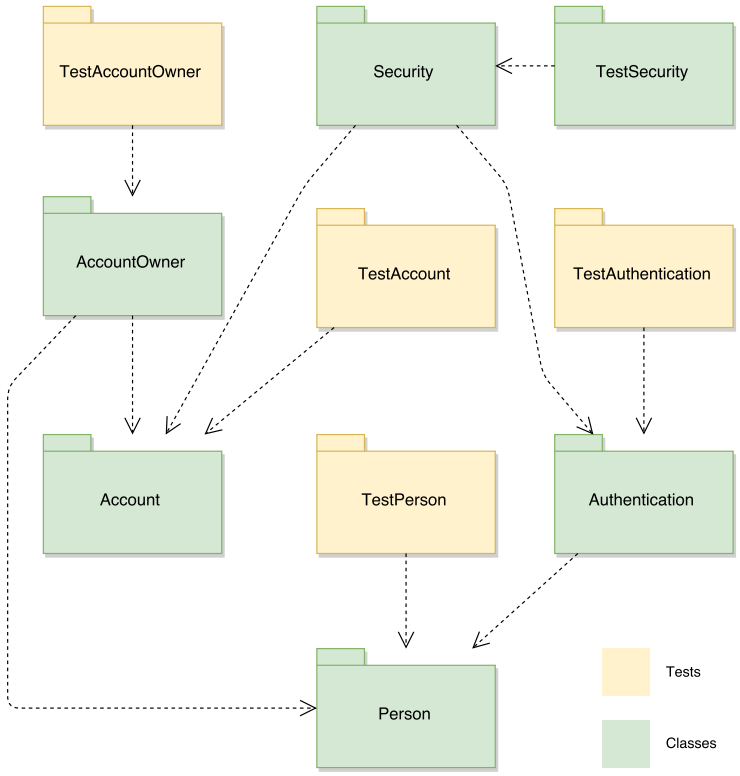

Figure 5.1: Banking System Dependencies

The problem with this approach is the same of the previous one. The difference is that the tangling now happens in the *security* module. The advantage is that the *Account* class, if used in isolation, now has working tests for its *transfer* method.

5.2.4 Reasoning

The reason why these approaches do not work is that there are several properties of the system that we want to keep but are unable to:

1. **Invasive aspects from higher order modules should not influence the outcome of a unit test.** The *testTransfer* test should run without the influence of the *security* aspect. Each test should run at least once without the influence of higher level modules and concerns. This assures that tests run in isolation as they are supposed to.

2. **Tests should be oblivious in relation to other modules.** The *account* module should have no knowledge of the *security* module. Tests inside a module should not have to care about other modules as we want them to run in isolation.
3. **Unit tests should only test the concerns of the module they belong to.** The *testTransfer* test should be coded into the correct module. Each test should be coded inside the module it is testing in order to make reusability easier.

All the proposed approaches violate at least one of these properties. Figure 5.2 shows what happens in each one of these proposed solutions. The diagram is horizontally divided into two different scenarios: the top scenario shows what happens when the *account* and *security* modules are used together; the bottom scenario shows what happens if we reuse the *account* module separately from the *security* module. In each scenario, we can see two images: the top one, represents the code that is actually compiled and that will be tested; the bottom one represents the code that the *testTransfer* test is really testing. The image is then divided into four parts representing the original code and the three proposed solutions:

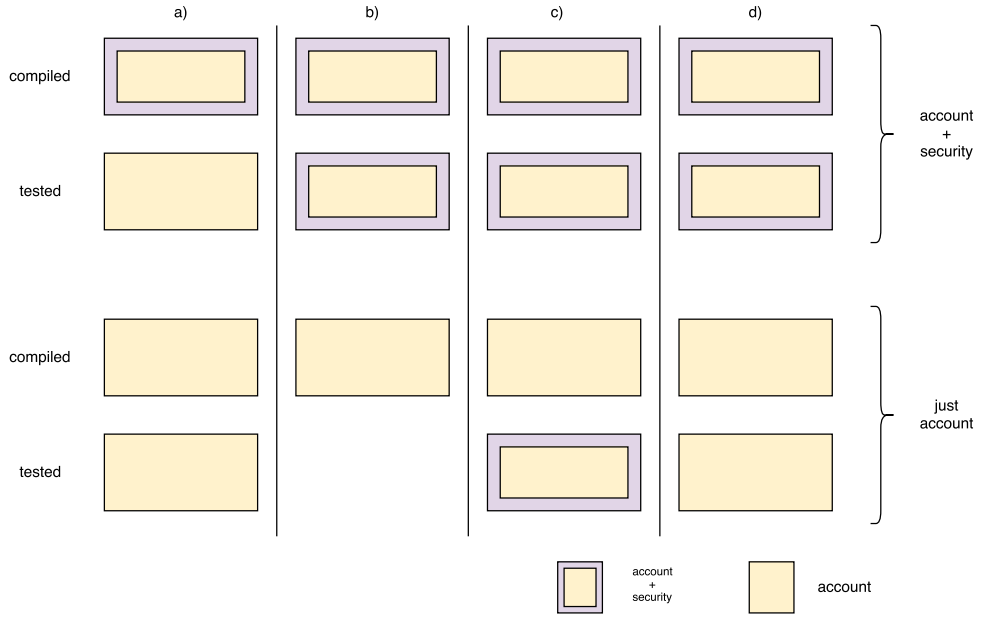

Figure 5.2: Naive Solutions to the Testing Concern Problem

- Figure 5.2a) shows the original scenario. The test code tests the *account* module without any regard for the *security* module. When run together, the code being tested is not the same as the compiled one (and, therefore, is prone to break inadvertently). However, when run in isolation, the test is accurate and tests only what it should.
- In Figure 5.2b) we moved the test. When both modules are compiled together, we are testing the same behavior that has been compiled. However, we are never testing the *account* module separately. When the module is used without the *security* module, it is not even tested.
- In Figure 5.2c) we changed the test. When both modules are compiled together, we have the same result as in the previous point. When the *account* module is used without the *security* module, the test that is run is the changed test. This will result in a compilation error as the *security* features that are used by the test are absent from the compiled code.
- In Figure 5.2d) we used AOP to change the test. When both modules are compiled together, we have the same result as in the previous points. When the *account* module is used without the *security* module, the correct code is compiled and tested. Besides being harder to code, the only problem with this approach is that when the complete system is tested, the *account* module is never tested in isolation.

As we have just shown, none of these solutions is perfect. In the next section we will discuss one the of the research issues that would greatly benefit from a more tamable approach to AOP testing. In Section 5.4 we will see how other authors have managed to use testing techniques together with AOP and in Chapter 6 we will propose our own solution for this problem.

5.3 Interferences

Besides providing higher software modularity, AOP also aims for a characteristic called *obliviousness* that states that developers should be able to implement application modules without any knowledge of previously implemented aspects or any future implementations [RD00, Rob01].

Current AOP languages, like AspectJ, are so powerful that *obliviousness* sometimes appears to be more of a problem than a solution. This is especially true when a

large number of aspect modules are added into an application, often changing its behavior, and thus making some aspect modules incompatible with each other. This happens because most aspects expect to be weaved into an application with a certain behavior and, if that behavior has been changed by a previously weaved aspect, then their own behavior could prove erroneous.

Having a good testing strategy is of paramount importance to detect these interferences [RA09a].

The next section presents a brief roundup of several classification terminologies found in the literature that try to describe the different types of conflicts and interferences between aspects. Following this section, we describe prior work regarding how conflicts can be detected, solved and prevented.

5.3.1 The Anatomy of Aspect Interferences

Understanding a problem is always the first step to solving it. In this particular case, it is important to identify the various kinds of interferences between aspects and how they emerge. Several researchers have tried to categorize aspect interaction according to different perspectives.

In the next sections, three different approaches to aspect interference terminology and cataloging are presented. These approaches look at the same problem from different angles and each one of them has an important perspective into the problem.

The Interference Point of View

The most important work done in this area is probably the classification of aspect interferences by Tessier [TBB04]. In his work the author points out several different ways in which aspects can interfere with each other:

- **Crosscutting Specifications.** The use of join points, and specially with '*' wildcards, can lead to accidental join points or infinite recursions.
- **Aspect-Aspect Conflicts.** When multiple aspects exist in the same system, problems like mutual exclusions between aspects, the importance of

aspect ordering, or conditional execution of an aspect by another aspect can occur.

- **Base-Aspect Conflicts.** Circular dependencies between aspects and basic classes.
- **Concern-Concern Conflicts.** Aspects changing a functionality needed by other aspect and composition anomalies normally happening due to subtype substitutability.

The Aspect Point of View

According to Katz [Kat04], three types of aspects can be described in respect to *how they affect an application*. This classification is important as some interferences only happen with some types of aspects. The three different aspect types are the following:

- **Spectative aspects** only gather information about the system to which they are woven, usually by adding fields and methods, but do not influence the possible underlying computations.
- **Regulatory aspects** change the flow of control (e.g., which methods are activated in which conditions) but do not change the computation done to existing fields.
- **Invasive aspects** change values of existing fields (but still should not invalidate desirable properties).

The Dependency Point of View

Kienzle [KYX03] approached the problem from a different point of view by considering only the *relationships of dependency between aspects and the original code*. Three different kinds of aspect dependencies have been identified:

- **Orthogonal aspects** provide functionality to an application that is completely independent of the other functionalities of the application. No data structures are shared between these aspects and the rest of the application. This kind of aspects is very uncommon.

- **Uni-directional aspects** depend from some functionality of the application. These can be further divided as *preserving* – if the application functionality is maintained or enhanced without any current functionalities being altered or hidden; or *modifying* – if the application functionality is altered or hidden.
- **Circular aspects** are mutually dependent of each other. This kind of aspects is so tightly coupled that one can argue if they should really be considered as separate aspects or as one unique aspect.

Axis of Invasion

Munoz [MBB08] proposed to classify invasive aspects under three different axes: control flow, data access and structural.

- **Control flow** invasive advices allow developers to change the control flow of a program. These have been further categorized as augmentation, replacement, conditional replacement, multiple and crossing.
- **Data access** invasive advices allow developers to change the flow of information in a program. These have been categorized as read, write and argument passing.
- **Structural** invasive advices allow developers to change the structure of a class. They can change the class hierarchy and add operations and fields.

5.3.2 Detecting Aspect Interferences

In order to solve the problem posed by the interference of aspects, a second problem must be solved first: how to detect that an aspect interferes with another aspect or module? Literature has many different ideas about how to solve this problem. These ideas are explained next.

Program Slicing

Balzarotti [BM04] claims that this problem can be solved by using a technique proposed in the early 80's called program slicing. A slice of a program is the set

of statements which affect a given point in an executable program. According to the author the following holds:

> Let A1 and A2 be two aspects and S1 and S2 the corresponding backward slices obtained by using all the statements defined in A1 and A2 as slicing criteria. A1 does not interfere with A2 if $A1 \cap S2 = \emptyset$.

According to the author, this technique is accurate enough to identify all interferences introduced by an aspect but some of those are later considered to be false-positives (i.e., intentional interferences). Furthermore, the existence of pointcuts that are defined based on dynamic contexts forces the analysis of every execution trace increasing the number of these false-positives. However, the approach has the advantage of removing the burden of having to declare formally the expected behavior of each aspect.

Aspect Integration Contracts

Contracts have been introduced by Meyer [Mey92] as a defensive solution against dependency problems in OOP. Some authors claim that contracts can be imported into the AOP world in order to assist programmers on avoiding interference problems.

Lagaisse [LJDW04] proposed an extension to the Design by Contract (DbC) paradigm by allowing aspects to define what they expect of the system and how they will change it. This will allow the detection of interferences by other aspects that were weaved before, as well as the detection of interferences by aspects that are bounded to be weaved later in the process. According to the author, for an Aspect A bound to a component C the following should be defined:

1. The aspect should specify what it requires from component C and possibly from other software components.
2. The aspect also needs to specify in which way it affects the component C and the functionality it provides (if applicable).
3. The specification of component C must express which interference is permitted from certain (types of) aspects.

This approach has the disadvantage of forcing the programmer to verbosely specify all requirements and modifications for each aspect as well as permitted interferences. On the other hand, the formal specification of behaviors has proven to be a valuable tool in Software Engineering.

Regression Testing

Katz [Kat04] proposed the use of *regression testing* and *regression verification* as tools that could help identifying harmful aspects. The idea behind this technique is to use regression testing as normally and then weave each aspect into the system and rerun all regression tests to see if they still pass. If an error is found, either the error is corrected or the failing tests have to be replaced by new ones specific for that particular aspect.

Service-Based Approach

It has been noticed by Kienzle [KYX03] that aspects can be defined as entities that require services from a system, provide new services to that same system and removes others. If there is some way of explicitly describing what services are required by each aspect it would be possible to detect interferences (for example, an aspect that removes a service needed by another aspect) and to choose better weaving orders.

Introduction and Hierarchical Changes Interferences

Störzer [SK03] developed a technique to detect interferences caused by two different, but related, properties of AOP languages. He claims that the possibility of aspects introducing members in other classes can lead to undesired behaviors as it can result in changes of dynamic lookup if the introduced method redefines a method of a superclass. He calls this type of interference *binding interference*.

The other problem Störzer refers to is the possibility of aspects changing the inheritance hierarchy of a set of classes. He claims that this type of changes can also give place to *binding interferences* as well as some unexpected behavior caused by the fact that *instanceof* predicates will no longer give the same results as before.

To detect this kind of conflicts, Störzer proposes an analysis based on the lookup changes introduced by aspects.

Kessler [KT06] also studied how structural interferences could be detected. However, his approach is based in a logic engine where programmers can specify rules (ordering, visibility, dependencies, ...). He also described the different types of interferences that are possible with introductions and hierarchical changes and proposes solutions for each one of them.

Graph-Based Approach

Havinga [HNBA07] proposed a method based on modeling programs as graphs and aspect introductions as graph transformation rules. Using these two models it is then possible to detect conflicts caused by aspect introductions. Both graphs, representing programs, and transformation rules, representing introductions, can be automatically generated from source code.

Although interesting, this approach suffers the same problem of other automatic approaches to this problem as intentional interferences cannot be differentiated from unintentional ones.

5.3.3 Aspect Interference Resolution

Douence [DFS02, DFS04] proposed a framework that allowed programmers to solve aspect interferences by using a dedicated composition language. The idea behind this language is to allow an explicit composition of aspects at the same execution point. The interferences solved by this approach are those that occur when the same crosscut is used by two different aspects.

5.3.4 Avoiding Aspect Interferences

The powerfulness of current AOP languages has been the target of several researchers that claim that without any control mechanisms, interferences will always be a big problem in the AOP world. In the next few sections, some of the approaches that follow this path are described.

Robust Pointcuts

Braem [BGKV06] proposed a method based on Inductive Logic Programming in order to automatically discover intentional pattern-based pointcuts. This method aims at solving the *fragile pointcut problem*, that states that pointcuts defined by enumeration do not cope well with program evolution and that the use of wildcards to solve this problem can cause interferences by means of accidental join points.

Crosscutting Interfaces

Crosscutting Programming Interfaces, or XPIs, have been introduced by Griswold [GSS+06] as a form of making AOP programming easier. By using abstract interfaces to expose pointcut designators, this approach decouples aspect code from the unstable details of advised code without compromising the expressiveness of existing AO languages or requiring new ones. The author expects that integrated-development-environment support could aid programmers by showing the scope of an XPI applicability.

Join Point Encapsulation

The reason why AOP is so powerful and at the same time so easily misused is that join points are available for weaving without any knowledge of the programmer that originally developed the code. On one hand, this allows the developers to be oblivious about what code is going to be weaved in their code, but, on the other hand, is the source of interferences and conflicts. Larochelle [LSS03] proposed the idea of adding join point encapsulation by introducing a new kind of advice: join point encapsulation advice, or restriction advice. Restriction advice serves to encapsulate the join points selected by a pointcut against modification by other aspects thus enabling the modular representation of the encapsulation of crosscutting sets of join points.

5.4 Related Issues

In this section, we will highlight some of the previous work done on the field of testing aspect oriented programming software.

5.4.1 Using Unit Tests with AOP

Zhou [ZZR04] acknowledged the difficulties posed by aspect oriented programming when testing is involved. He proposed a four step approach to the problem.

In the first step, only classes are tested. This would enable the elimination of non-aspect related errors.

The second step is to weave each aspect with the base classes separately and test each one of the woven systems. Zhou claims that this is similar to unit testing each concern and that aspects seldom interact with each other.

In the third step, several aspects would be tested simultaneously in order to do integration testing. This step would enable the interaction between aspects to be tested.

The final step would perform system-wide testing as all aspects are tested at the same time.

Ceccato [CTR05] explores the idea that aspect oriented programming code is easier to test than object oriented programming code because each concern can be tested separately due to the nature of aspects themselves and at the same time harder due to the new fault types introduced by the paradigm (see Section 8.2). To solve the identified problems, he proposes an incremental test based approach.

Wanga [WZ12] addresses some of the problems posed by testing aspect oriented programming by testing base classes first using unit testing, then testing each aspect using integration testing and finally testing the complete woven system using system testing.

5.4.2 Using Different Approaches to Testing AOP

Mortensen [Mor05] described an approach to solving errors both in the quantification of pointcuts and in the code of advices. He proposes the combination of two traditional techniques – coverage and mutation testing. By using static analysis of an aspect to guide in the selection of appropriate coverage criteria for aspect code fragments, he provides a set of mutation operators to evaluate if a test suite is sufficiently sensitive to find errors.

Xu [XX06] proposed a state-based approach to testing AOP programs. He argues that it can reduce the testing cost by reusing test cases and identifying aspect specific faults. They achieve that by thinking of aspects as incremental modifications of their base classes and identifying how to reuse the concrete base class tests for testing. Xu [XEAXW12] also presented a framework for testing whether or not aspect oriented programs conform to their state models. This framework supports two families of strategies for the automated generation of aspect tests from aspect oriented programming state models. The first strategy derives tests and test code from an aspect-oriented state model while the second one generates test code from the counterexamples of model checking. The author used mutation analysis to evaluate the effectiveness of these testing strategies.

Parizi [PGAA09] [PGAA09] proposes using automated random testing (ART) with AspectJ. He argues that it is possible to do so but that current distance measures would not be all applicable or sufficient to address the notion of evenly spread of test cases suggested by ART. Later [PGL15] he proposed an automated test generation technique with tool support based on guided random testing for AspectJ programs.

Wedyan [WG10] applied data flow testing to AOP. He created a tool that identifies the *DefUse Associations* in AspectJ programs and computes the coverage obtained from a test suite.

Ferrari [FNRM10] created a mutation-based testing tool called *Proteum/AJ* that, according to the author, overcomes some limitations identified in previous tools for aspect oriented programming programs. In a previous study, together with Levin [LF14], they studied the effort required to migrate test code from object oriented programming to aspect oriented programming programs.

Delamare [DBG+11] developed the *AdviceTracker* tool that monitors the execution of advices in an aspect-oriented program and uses this information to build test cases that target faults in pointcut designators.

Alexander et al. [ABAA04] presented the differences between OOP and AOP when it comes to testing. In particular, they address the fact that aspects have to be tested in the context into which they are woven. They propose a fault model and testing criteria for AOP.

Ceccato et al. [CTR05], explored the differences between testing AOP and OOP. They also considered an incremental testing approach for AOP.

Lopes et al. [CVL05], proposed using mock objects that emulate execution context information in order to use unit tests on aspectual behavior.

Yamazaki et al. [YSM+05] proposed a method of unit testing without weaving an aspect by describing test cases from the same viewpoint as describing the aspect for the program.

5.5 Summary

Testing software is an important part of most software development processes. It is the phase where faults are eventually discovered before they become part of the finished product.

In this chapter, we have shown that unit testing, as is, is not compatible with the use of AOP, as either the modularity of those tests is broken or not all tests are run. In the next chapter, we will present a possible approach to solving this problem that enables unit tests to still be free of crosscutting concerns without losing the ability to test all methods in isolation.

Depend upon it there comes a time when for every addition of knowledge you forget something that you knew before. It is of the highest importance, therefore, not to have useless facts elbowing out the useful ones.

Arthur Conan Doyle – 1887

6

Modular Testing in AOP

Contents

In this chapter, we will discuss a methodology that allows unit testing of AOP programs without breaking the modularity introduced by the use of aspects.

In the previous chapter, we have shown that while aspects allow a better separation of concerns, when we introduce unit tests, that separation is broken. This happens because tests are created without having in mind the addition of invasive aspects.

What we propose is a methodology where invasive aspects are allowed to declare tests from other modules as being deprecated and then adding its own tests. In the next sections, this methodology will be presented and explained in detail.

6.1 Dependency Graph

Software systems are made up of modules. Modules are independent units that can be used to construct complex systems. They should also be easily reused in a different project, and if they have unit tests, those tests should also be easily reused.

When a module interacts with another module, through a well-defined interface, we can say that that module depends on the functionality of the other module to perform its tasks. The set of dependencies between modules can be seen as a directed graph where nodes represent the modules and edges represent the dependencies.

A well-designed software system should be built in such a way that low-level modules do not depend on higher level modules. Software systems should be built layer by layer, with each layer adding more functionalities. if this is accomplished, then the module graph becomes a Directed Acyclic Graph (DAG) (see Figure 6.1).

Unfortunately, not all software systems follow this recommendation and it is common to find circular dependencies even in the most well-designed software systems. In graph theory, these collection of nodes that form circular dependencies are called *strongly connected components*, and although they cannot be easily removed, they can be isolated. We do this by considering each strongly connected component in the graph as a super module. In this way, we can consider that all software systems can be thought as being composed as a DAG of module dependencies (see Figure 6.2).

The nature of these dependencies is different when we are discussing the classic OOP model or an AOP based program. While in OOP, dependencies are created when a class references a class or an interface defined in another module, in AOP

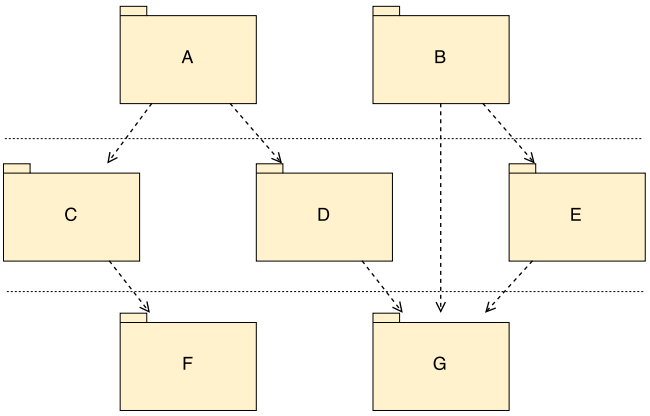

Figure 6.1: Dependencies Between Modules

references can also be created when join points are declared. These dependencies can be more complicated to detect as in some languages the join point model can be very expressive.

But the real difference between OOP and AOP dependencies is that when module A depends on module B, in a pure OOP program, we can be sure that module B behavior is not changed directly by module A. The same cannot be said if module A contains invasive aspects.

6.2 Testing Modules

When testing modules in an AOP program we have to take into consideration that some modules might change the behavior of modules they depend on. But if we are able to extract the dependency graph, that we just described in the previous section, from the source code of an AOP program, we can be sure that we will have at least one low-level module (or super module) that does not depend upon any other modules.

This module, or modules, can be easily tested in isolation as they can be compiled separately. Their unit tests can be run without any danger of them failing due to behavior modifications introduced by aspects belonging to higher level modules.

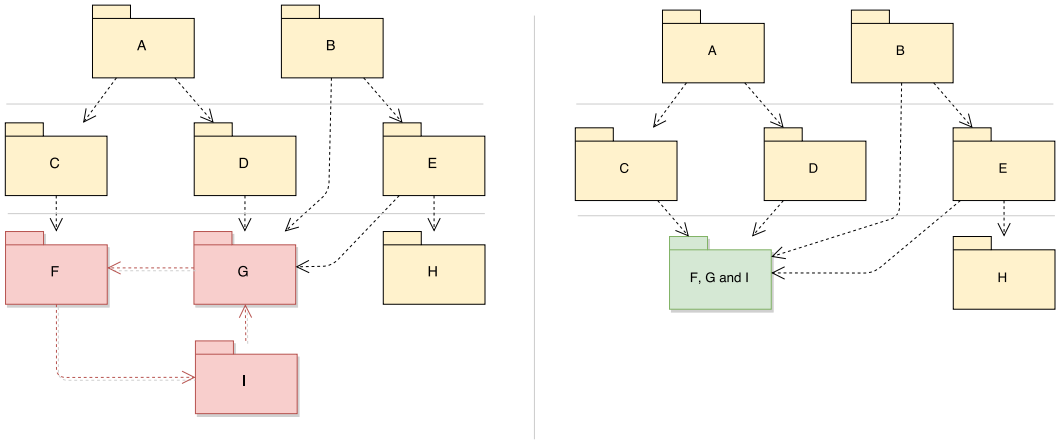

Figure 6.2: Identifying and Removing Circular Dependencies

After this initial testing step, some of the remaining modules will depend solely on the modules that have already been compiled and tested. We can take one of these modules and compile it together with the previously tested modules. Then, we can run the tests not only for the newly selected module but also the tests that have already been run for previous. This process can continue until there are no more modules to test.

When a new module is selected for the testing process, its tests can either all pass or some of them might fail. If a test fails on the newly selected module, we can be sure that the error has not been introduced by a higher level aspect as these have not been compiled into the system.

If a test on a previously tested module fails when the new module is introduced into the testing process, then it means that the new module altered the behavior of one of the previous modules. This can mean one of two things, either the newly introduced module is defective, or it is changing the behavior of the previous module in such a way that its tests have become deprecated and must be replaced with new tests.

Unfortunately, a mechanism that allows developers to deprecate and replace tests in modules whose behavior is changed by invasive aspects from other modules does not exist. This forces developers to cheat by using one of the approaches previously described in Section 5.2. In the next section, we will propose a different approach to this problem.

6.3 Annotating Tests

To tackle this problem, we propose an annotation based approach where developers can mark tests as being deprecated. The whole testing process becomes as follows:

1. Extract a dependency graph between the modules that compose the program.
2. Identify circular dependencies and treat those modules as being a single larger module. This effectively transforms the dependency graph into a DAG.
3. Pick a module that does not depend on any untested modules and test it together with the previously tested modules. Both tests from previous modules and the module being tested should be executed. Several different things can happen:
   (a) Some tests from the current module fail. This means that there is a bug either in the code of the new module or in its tests. The testing procedure stops and the problems are reported. The developer fixes the module and repeats the testing procedure from step 1.
   (b) Some tests from previous modules fail. The testing procedure stops and the problems are reported. After analyzing the results, the developer can interpret them in two different ways:
      i. The code of the new module has a bug and it is changing the behavior of another module in an unexpected way. The developer fixes the module and repeats the testing procedure from step 1.
      ii. The code of the new module is changing the behavior of a previously tested module in an expected way. If it does not already exist, the developer adds a new test for the new behavior and adds an annotation stating that the old test is deprecated by this new test. This effectively means that the deprecated test should not be run if the new test is part of the system being tested.
   (c) All tests pass. Nothing needs to be done.
4. If there are still untested modules, go back to step 3.

Figure 6.3 contains a simple flow chart depicting the whole process. In this figure, the green colored boxes denote actions and decisions that are part of the automatic testing system. The yellow colored boxes denote actions and decisions done by the developer.

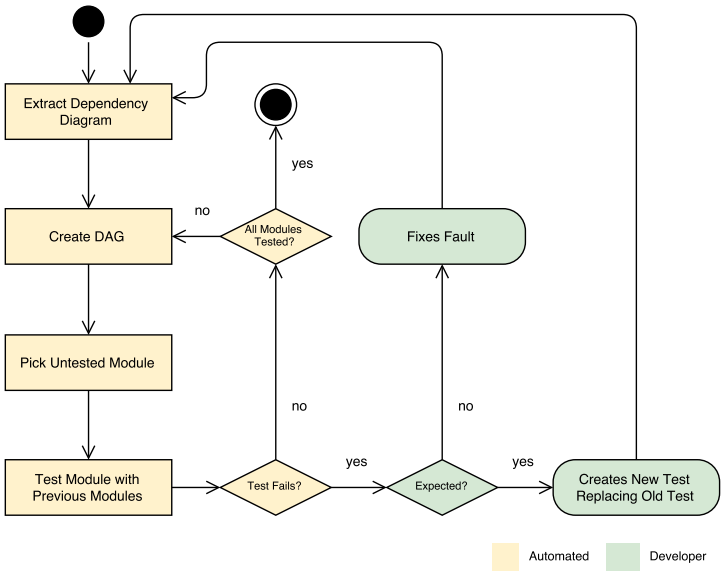

Figure 6.3: Modular Testing Process

6.4 Example Scenario

In the motivational example presented in Section 5.1, we described a very simple stripped down system for a bank. In this example, the code has been separated into five modules: *Person*, *Account*, *Authentication*, *Security*, and *AccountOwner*. Figure 5.1 shows the dependency graph for this system.

As was shown in Section 5.2, the problem with testing invasive aspects is that it is in their nature to disrupt tests from lower level modules and there is no solution to prevent that from happening except by breaking some other desirable properties.

In this example, the *Security* module breaks the tests developed for the *Account* module as these do not expect having to login into the system. By using our approach this is what would happen:

1. The *Person* module would be tested in isolation.
2. The *Person* and *Account* modules would be tested together. As they don't depend on each other their tests would not interfere.
3. The *Person*, *Account* and *Authentication* modules would be tested together. As they don't depend on each other their tests would not interfere.

4. The *Person*, *Account*, *Authentication* and *AccountOwner* modules would be tested together. The *AccountOwner* module depends on the *Person* and *Account* modules, but it is not an invasive aspect so the tests belonging to these two modules should not be affected. It is up to the test developer to make sure that the *Person* and *Account* modules do not interfere on the tests of the *AccountOwner* module. This can be accomplished by using mocks or stubs (see Section 4.3.1).
5. The *Person*, *Account*, *Authentication*, *AccountOwner* and *Security* modules would be tested together. As the *Security* module has invasive aspects in relation to the *Account* module, tests in that module do not expect users to authenticate before performing transactions. This means that tests in the *Account* module, namely the *testTransfer* test, would now fail.

By analyzing the causes of this failure, the developer could easily understand that the test failed due to an interaction between these two modules. As this interaction was expected, the developer just has to add a new test to the *Security* module that tests this interaction. This test should have an annotation stating that it replaces the *testTransfer* test in the *Account* module (see Listing 6.1).

Listing 6.1: Secure Transfer Test

```
1  @ReplaceTest("Account.testTransfer")
2  public void testSecureTransfer () throws Exception {
3    Person p1 = new Person ("Jonh", 1, "john", "password");
4    Account a1 = new Account (p1, 1, 1000);
5
6    Person p2 = new Person ("Mary", 2);
7    Account a2 = new Account (p2, 2, 500);
8
9    Authentication.login("john", "password");
10
11   assertEquals (1000.0, a1.getBalance ());
12   assertEquals (500.0, a2.getBalance ());
13   a1.transfer (200, a2);
14
15   assertEquals (800.0, a1.getBalance ());
16   assertEquals (700.0, a2.getBalance ());
17 }
```

In this way, when the code is tested again, as the *Security* module is added, the system will know to use the *testSecureTransfer* instead of the *testTransfer* test.

In the following section, we will analyze our approach in a more formal manner. In Chapter 7 we will present an implementation of this approach for the AspectJ language.

6.5 Formal Analysis

6.5.1 Domain of Discourse

We will start by defining our domain of discourse. In this analysis, our variables and terms will be modules, compositions of modules and unit tests of the same program. To simplify our analysis, we will assume that all terms of the form m or m_i are modules, all terms of the form c or c_j are compositions and all of the form t or t_i are tests.

6.5.2 Operators

We will start by defining a couple of operators that will make it easier to follow the analysis. The composition operator ($\circ$) represents the composition of two modules compiled together (see expression(6.1)). A module can also be composed with a composition (see expression (6.2)) and two compositions can also be composed with each other (see expression (6.3)).

$$c = m_i \circ m_j \tag{6.1}$$

$$c_i = c_j \circ m \tag{6.2}$$

$$c_i = c_j \circ c_k \tag{6.3}$$

We will also use the belongs operator ($\in$) to specify that a certain module or composition is part of a certain composition (see expression (6.4)).

$$m \in c \tag{6.4}$$

And we will use the belongs operator ($\in$) to specify that a certain test is defined inside a certain module (see expression (6.5)).

$$t \in m \tag{6.5}$$

And, of course, if a test is defined in a module that is part of a composition, the test is also part of the composition (see expression (6.6)).

$$t \in m \wedge m \in c \rightarrow t \in c \tag{6.6}$$

6.5.3 Predicates

We will also define certain predicates that pertain to the domain of discourse.

Modules and compositions may depend on other modules (predicate D). A module or composition that depends on another module cannot be compiled separately from the module it depends on (see expression (6.7) – module depends on module – and expression (6.8) – compostion depends on module).

$$D(m_a, m_b) \tag{6.7}$$
$$D(c, m) \tag{6.8}$$

Compositions and modules may also depend on compositions. This means that they depend on, at least one, of the modules that are part of the composition (see expression (6.9) and expression (6.10)).

$$D(m, c) \leftrightarrow \exists m_a \in c \; D(m, m_a) \tag{6.9}$$
$$D(c_a, c_b) \leftrightarrow \exists m_a \in c_a \exists m_b \in c_b \; D(m_a, m_b) \tag{6.10}$$

If all modules that a certain module or composition depends on are part of one composition, we can declare that that module or composition depends only (predicate Do) on that one composition (see expression (6.11)).

$$Do(m, c) \leftrightarrow \neg \exists m_a \notin c \; D(m, m_a) \tag{6.11}$$

If a module m has all its dependencies satisfied in composition c, then the composition $m \circ c$ is valid. However, this does not prove that the composition is correct.

Tests can be applied to modules or compositions. If a test run succeeds in the scope of a certain module or composition we can say that the module or composition passed the test (see expression (6.12) – module passes test – and expression (6.13) – composition passes test).

$$P(m, t) \tag{6.12}$$

$$P(c, t) \tag{6.13}$$

Modules can declare that they are replacing tests. We can also declare if a test was replaced by a module using the predicate R (see (6.14)).

$$R(t, m) \tag{6.14}$$

A module or composition can be declared correct if its behavior is consistent with its requirements. Tests can be used to assert if this is the case. A correct module or composition can also be declared by using the predicate C (see expression (6.15) and expression (6.16)).

$$C(m) \tag{6.15}$$

$$C(c) \tag{6.16}$$

6.5.4 Assumptions

Assumption 6.1 *All tests are correct. If a composition is correct, all tests defined inside its module must pass except those that have been replaced by modules inside that same composition (see* (6.17)*).*

$$\forall c\ C(c) \rightarrow \forall m \in c\ \forall t \in m\ (R(t, m) \vee P(c, t)) \tag{6.17}$$

Assumption 6.2 *All tests are complete. If all tests defined inside a composition, except those that have been replaced by modules inside it, pass then the composition is correct (see* (6.18)*).*

$$\forall c\ \forall m \in c\ \forall t \in m\ (R(t,m) \vee P(c,t)) \rightarrow C(c) \tag{6.18}$$

6.5.5 Theorems

Based on these assumptions, we will now try to prove some theorems that describe the whole base of the approach.

We will start by analyzing the incremental nature of the approach. Namely, if adding a new module to the test process and retesting previous and new tests is a valid option to test if a program is correct.

Theorem 6.3 *Let c be a composition of modules tested as being correct. Let m be a module that depends only on the modules contained in c. If all tests defined in $m \circ c$, that were not replaced by m or by c, pass for composition $m \circ c$, then composition $m \circ c$ is also correct (see* (6.19)*).*

$$C(c) \wedge Do(m,c) \wedge \forall m_t \in (m \circ c)\ \forall t \in m_t\ (R(t, m \circ c) \vee P(m \circ c, t)) \rightarrow C(m \circ c) \tag{6.19}$$

Proof If module m depends only on modules defined inside composition c, then composition $m \circ c$ is valid. If we substitute composition c in the assumption 6.2 by $m \circ c$ we get the same expression we are trying to prove. Hence, the theorem is correct under the assumption that tests are complete.

We will also need to show that when a module is not correct, the approach correctly predicts the existence of a fault.

Theorem 6.4 *Let c be a composition of modules tested as being correct. Let m be a module that depends only on the modules contained in c. If there exists a test defined in m that fails for composition $m \circ c$, then composition $m \circ c$ is not correct (see* (6.20)*).*

$$C(c) \wedge Do(m, c) \wedge \exists t \in m \ \neg P(t, m \circ c)) \rightarrow \neg C(m \circ c) \tag{6.20}$$

Proof As we have seen in assumption 6.2, for a composition to be correct all tests defined inside it must pass or be replaced by a module. If there is a test t belonging to module m, then this test could not be replaced by composition c as this would create a dependency between c and m and invalidate composition c, and it also could not be replaced by module m, as it is defined inside it. As the test fails to pass for composition $m \circ c$ then this composition must be incorrect.

6.6 Testing for Interactions

Although AOP improves modularity, sometimes reasoning about interactions between modules is not as easy as in OOP. This happens as reasoning about the base code can become invalid once aspects from other modules are taken into account.

When unit tests uncover faults, it is also important to be able to pinpoint easily where the fault originated. In this section, we will explore how modular testing can be used to reason about unexpected interactions that are the source of faults.

By transforming our dependency graph into a directed acyclic graph and then incrementally compiling and testing its modules using a topological sort order, we might get not one compilation order but several ones (see Figure 6.4).

Using any one of those orders is enough to use our testing approach but we can take advantage of this fact to extract even more information from the tests.

For example, in Figure 6.5, we can compile incrementally the four modules in three different ways: {C, B, A, D}, {C, B, D, A} or {C, D, B, A}. Let us imagine the following scenario:

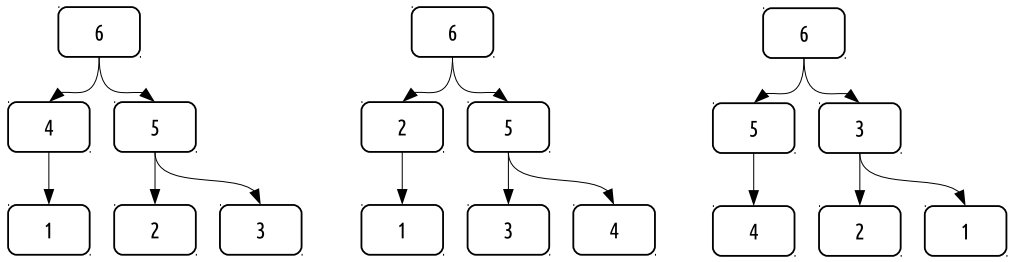

Figure 6.4: Some Possible Compilation Orders

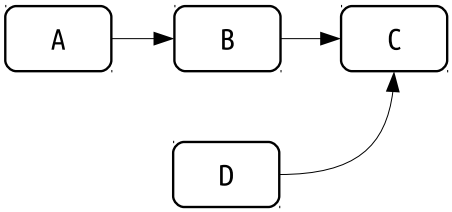

Figure 6.5: Dependency Example

1. When testing using the order {C, B, A, D}, tests for modules C, B and A pass and when module D is added a test from itself fails. The only information we can extract from this test is that something is wrong with the composition of module D with the remaining modules.
2. When testing using the order {C, D, B, A}, tests for modules C, D and B pass and when module A is added a test from module D fails. Using this order there is some more information we can extract from the test. Not only is there something wrong with module D but that something seems to be originating from module A. As module A does not depend or interact with module D directly, the developer could extrapolate that module A is changing something in module B that is changing something in module C which is breaking module D. The developer could then add some annotation to module A stating that it broke module D.
3. We could easily confirm this problem by running the tests using the order {C, B, D, A} and noticing that only when module A was added into the system did module D break.

In Figures 6.6 and 6.7 we can observe how we can extract two different informations depending on the compilation order. In Figure 6.6, when *module* D is added, we cannot pinpoint if there is an interaction with another *module*. In Figure 6.7, when *module* A is added, *module* D fails its tests revealing an interaction between the

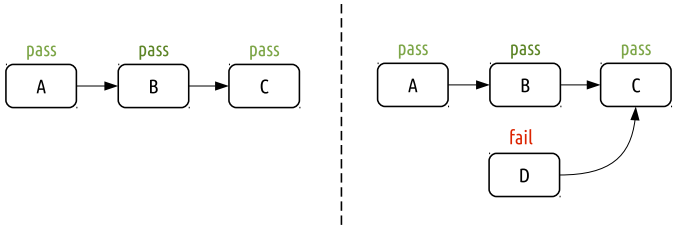

Figure 6.6: Module D is Added After Module A

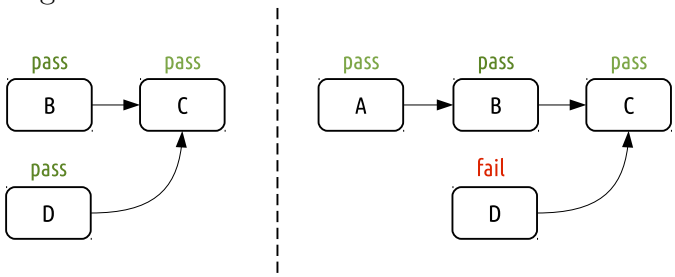

Figure 6.7: Module A is Added After Module D

two modules.

To ensure the maximum amount of information is extracted from these tests, the module where the test failed should be added as soon as possible to the system. This allows us to deduce that if, at that time, a unit test fails, then the problem lies in the added module as the only other modules added all passed their tests.

On the other hand, if all tests still pass when the module is added, then the fault is in a higher level module. One that is changing the behavior of the module or the behavior of a module this module depends on.

Taking this into consideration, after discovering that *module* D, from the previous example, has a test that fails. We should compile the modules incrementally in the following order: C, D, B and A. The test that failed previously will either fail when module D is added, uncovering a fault in that same module, or when modules A and B are added, pointing to an interaction between one of those modules and module D.

In the next section, we will summarize the proposed testing strategy that has been described in this chapter.

6.7 Proposed Testing Strategy

The strategy that has been delineated throughout this chapter can be summarized in four clear steps:

- **Develop modules with few or no circular dependencies.** The main objective of using AOP is to prevent crosscutting concerns from polluting the code. This will help make the code more modular which will improve several other aspects like reusability and maintainability. A code base where modules have circular dependencies removes a lot of these advantages so it should be already a top priority to have as few of these as possible. When using this strategy, it becomes even more important to observe this principle as circular dependencies prevent an incremental compilation strategy.
- **Use classic unit testing techniques.** Testing techniques for OOP code have already been thoroughly discussed in the literature. We argued that they are not enough for the aspect oriented programming case, but they should be used as a starting point. By using stubs and mocks, the developer can isolate each module from the behavior of lower-level modules.
- **Use incremental testing.** In order to isolate tests from the behavior of possible higher level modules containing invasive aspects, we should use an incremental testing strategy. This strategy complements the use of classical unit testing techniques by providing protection from higher level modules without breaking the encapsulation within each module.
- **Try different composition orders.** If a fault is detected, try different composition orders in order to better pinpoint the origin of the fault. This should be done only after a fault has been detected. The faulty module should be added as soon as possible so that all possible interactions can be analyzed.

6.8 Strategy Evolution

During the course of this work, the approach that we propose has taken different forms. In this section, we try to explain the evolution behind it and what are the advantages as disandvantages of each new step.

6.8.1 Method-Test Approach

Our initial idea was to consider tests as being the proof that a certain concern was implemented correctly. Ideally, for every concern in the system, the developer should be able to create a test for it. Class methods are the artefacts that end up implementing those concerns. So we could have annotations in each method with a reference to the test for the concern that the method was implementing.

In order to create the DAG of dependencies needed for our incremental testing process, we proposed another annotation where each method could declare the tests it depends on. Notice that we do not specify which methods the method depends on, but the tests that were created to test that method. This means that every time an aspect is added to the system, in our incremental testing process, and a previously tested test fails, we can pinpoint which methods are affected by that interaction.

Finally, we proposed another annotation that allows developers to declare expected interactions. This annotation would be used by developers on advices to pinpoint which tests they expect to break.

Having these annotations in place, our testing process would be able to create the directed acyclic graph of dependencies, using the *requires* and *adds* annotations, and run only the tests that have not been removed by subsequent advices by means of the *removes* annotation.

Besides the extra effort put in by the developer, the problem with this approach is that the relation between methods/advices and tests is artificial.

6.8.2 Concern-Test Approach

To mitigate the artificiality of our first approach, we decided to add a new annotation that would depict a *concern*. In this approach, each method has an annotation stating which concern it implements. These concerns can even be derived from the requirements phase.

In this way, methods and advices no longer add, remove or depend on tests but on concerns. To know which concern is tested by each test we need an annotation that will be applied to each test with a reference to the concern.

To apply our proposed testing process using this approach, we start by selecting a module whose methods do not depend on any concern from another module. Tests for the concerns defined in the module are run. In each step we add another module that only has *requires* annotations referencing concerns added by modules that already have been tested. If a test that passed in a previous step fails after a new module is added we can infer that there is an interaction between a concern implemented in that module and the concern that the failing test was testing.

In comparison with the first approach, this one has a richer set of metadata on the implemented concerns and their tests. This extra knowledge allows us to better understand which concerns are interacting with each other. Developers can therefore reason more easily if the interaction is expected or if it is an unexpected interference.

6.8.3 Module-Test Approach

The previous two approaches imposed an heavy burden on the developers as they had to add a lot of annotations to the code. In this iteration we tried to reduce the amount of extra work needed by removing most of them.

We started by considering modules as being defined by the way the used language, in this case *AspectJ*, defined its own units of modularity – *Java* packages. To prevent cases where the relation between the language defined units and the intended modules is not a direct one, we added an optional annotation so that each class/aspect could define to which module it belongs.

Tests defined inside a module are considered as being used to test some concern of the module. This removed the burden to add annotations for each test.

The only annotations really needed, are between tests. The *replaces* annotations identify cases where a test represents a concern, developed as an invasive aspect, that changes the behavior of another concern that is tested by the other test.

This approach drastically reduced the amount of extra work by the developer. However, the information gathered is much less. But still, when interactions are detected we can get information about which test failed and which modules caused the interaction. This information should be enough for the developer to identify the origin of the problem and act accordingly.

6.8.4 Advice-Test Approach

The last approach considered was an easy evolution from the previous one. The only mandatory annotation in our previous approach was used to remove a test from the system when a module containing invasive aspects was added to the system changing the behavior the test was testing. An alternative would be to use an advice to disable the test.

Although this approach does not use any annotations, besides the optional one that changes the way modules are defined as language constructs, the incremental compilation process is still needed to ensure that disabled tests are run at least once during testing.

6.9 Limitations

In the previous sections, we have shown how tests will crosscut different modules, even when using aspect oriented programming, making them, the modules, hard to reuse. We then described a testing strategy that promises to prevent the tangling and scattering of these unit tests. However, there are some limitations to this strategy that we will summarize in the next few paragraphs.

The strategy is extremely sensitive to bad code. Especially code that has lots of circular dependencies. This makes the strategy harder to use in poorly written legacy code. Fortunately, a tool based on this strategy will be able to analyze dependencies and warn the developer when circular dependencies are being created.

For the strategy to work perfectly, tests must be complete. When an invasive aspect changes the behavior of a module that module must have a test that detects that the new behavior is faulty. This is not always feasible as the tests should not have been written having in mind the aspect. It will be the responsibility of the developer of the invasive aspect to expect a test to fail in the modified module and add one if no test fails.

To implement the strategy, there must be a way to inspect the code and extract the dependencies and invasive behavior between modules. In some programming languages, this might be hard to achieve.

Although we tried to minimize the extra work needed, this strategy will mean that developers will have to step up their unit tests and add some extra annotations.

In large projects, the time needed to run all unit tests is usually quite high. By applying this strategy, the time needed to test the complete system will be squared. This might be a possible show stopper.

The last identified problem is that the strategy does not, in fact, remove all the modularity problems. When an annotation is declared in a module, stating that a test from another module is being deprecated in favor of another one, we are adding information to the module that is not pertinent to that module. Ways to solve these and other issues will be discussed in Section 9.2.

6.10 Summary

In this chapter, we delineated a possible solution to the problem identified in Chapter 5.

We have shown that programs that are well structured – with few circular dependencies – can be annotated in such a way that allows unit testing without losing any modularity.

We also identified the importance of creating tools to aid the developer into implementing the ideas that we delineated in this chapter.

In the end, we feel that an incremental testing strategy is an important step forward in order to make aspect oriented programming more usable.

In programming, the hard part isn't solving problems, but deciding what problems to solve.

Paul Graham – 2004

7

Implementation

Contents

Two tools that target AspectJ have been developed to experiment with the strategy proposed in the last chapter. These tools were implemented as Eclipse [Ecl10] plugins.

AspectJ was chosen as the target language for several different reasons. First, it is one of the most used AOP languages and a target for much of the current research. Secondly, as it is Java based, it can be used with Eclipse, an IDE where plugin development is straightforward. With Eclipse we also get two other important benefits, Java Development Tools (JDT) [JDT10] and AspectJ Development Tools (AJDT) [AJD10], tools for, respectively, the Java and AspectJ languages that allow access to the source code abstract syntax tree.

The original idea behind this thesis was to better understand how aspects interfered with one another and, especially, how unexpected interferences could be detected automatically. As time progressed, the original idea was expanded as it soon became clearer that the proposed approach could be used not only for interference detection but for testing in general.

During the course of the work, two different plugins were developed: *DrUID* and *Aida*. Both are based on the usage of annotations throughout the code that contain information about which interferences are expected.

A first approach to the problem suggested that the following annotations were necessary [RA07][RA08] (see Figure 7.1 and Appendix A):

- **@AddsTest** Indicates that a certain test has been added in order to test the code where the annotation has been attached to.
- **@RequiresTest** Indicates that the code where this annotation is used, requires the feature implemented by the code that added the test referred by the annotation.
- **@SupressesTest** Indicates that the code where this annotation is used changed the feature that was tested by the test referred by this annotation.

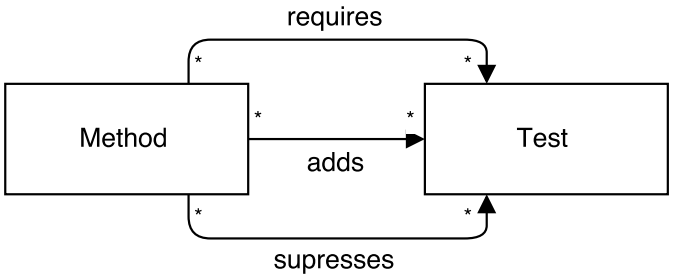

Figure 7.1: First Approach to the Problem

The idea behind this approach is that tests are indicative of features. Whenever a feature is implemented, a test can be created to test it and can be used as the indicator that the feature is still present and working. Whenever there is an interaction, a test will be broken. If the interaction is expected, the culprit code can announce that intention using an annotation.

As work progressed, other approaches where developed and, based on these new approaches, two different tools were created – *DrUID* and *Aida*.

7.1 DrUID: Unexpected Interference Detection

DrUID (UID as in Unexpected Interference Detection) [Res09] was the first attempt at creating a plugin to help developers follow the methodology being explored throughout this document. In order to accomplish this, the plugin allows developers to define several characteristics about system artifacts using Java annotations (see Figure 7.2):

- **@Feature** Used to annotate code as to being part of the implementation of a certain feature.
- **@Depends** Used to mark code as being dependent on the implementation of a certain feature. This defines the order in which the system is composed and tested.
- **@Deprecates** Informs the plugin that a certain piece of code breaks a certain feature. To be used when a expected interaction is detected.
- **@Tests** Used on tests to inform the plugin that they are used to test a certain feature. When the test fails then the test is defective or feature is either defective or has been changed by another feature. In the last case this makes it an interaction.
- **@Unit** Used to group different classes into units.

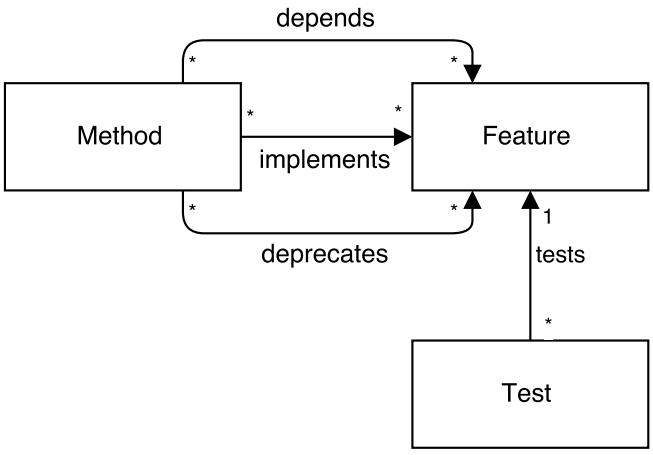

Figure 7.2: The DrUID Approach

Several aids have been implemented to guide the developer in this process in the form of Eclipse *quick fixes* and *quick assists*. Each time a file is saved in

Eclipse, the annotations are inspected and any errors are reported. Besides that, a dependency graph is created and shown in a graphical form (see Figure 7.3) that allows the developer to navigate through the code following the dependencies between artifacts.

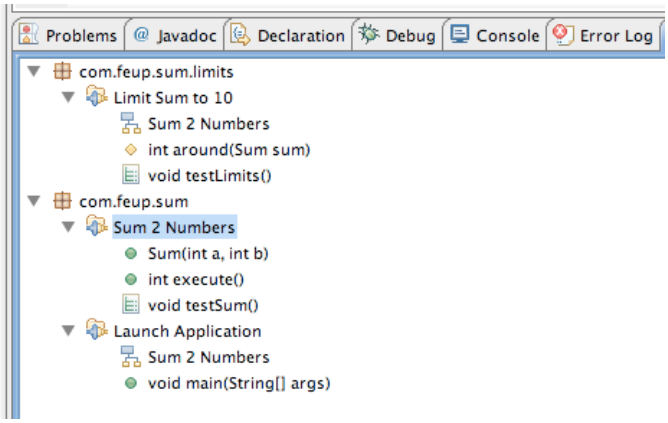

Figure 7.3: DrUID: Dependency Interface

With the dependency graph created, the developer can order DrUID to execute a conflict test analysis. This analysis executes the following steps:

1. Execute a *strongly connected* analysis of the dependency graph transforming it into a Directed Acyclic Graph (DAG) of components (sets of artifacts).
2. Execute a *topological sort* in order to determine in which order the components must be compiled.
3. For each component:
   (a) Compile it.
   (b) Execute the tests defined for the features provided by this component.
   (c) Execute the tests defined by previous components.

Step 3b) verifies if the component is working as expected and step 3c) verifies if previous features have been broken by the newly introduced component.

When the later happens, an interaction has been detected and the analysis stops. This interaction is reported to the developer with an explanation of which feature has been broken by which component. This will allow the developer to understand what went wrong and act accordingly. Two different conclusions can be extrapolated by the output generated by the tool: (i) the interaction is expected (ii) the interaction reveals a conflict between aspects.

In the first case, the developer simply has to declare the interaction by using, once again, annotations. By using Eclipse quick fixes added by the plugin this can be done easily by the developer.

In the latter case, we are confronted with a conflict that has to be dealt with in another way. If possible, the developer can change the implementation of one of the aspects in order to remove the conflict. But in some cases, the conflict might have been caused by two incompatible requirements and, in that case, it can only be corrected by changing the software requirements themselves.

After being presented to the community [RA09b], several issues with the implementation have been pointed out and addressed in the following iteration of the plugin. The most debated problem was that the plugin asked too much extra work from the developer.

7.2 Aida: Automatic Interference Detection for AOP

Aida [Res10] is an evolution of the DrUID tool, built from scratch, having the main objective of removing most of the burden put on the developer to annotate his code. It also has a bigger focus on the testing process. In this tool, we started by removing the notion of annotating features manually. We did this by considering each test as a feature. This means that the developer only needs to create test cases for each individual behavior. Obviously, this also removed the need to specify which test case tests what feature.

Using code inspection, we were also able to remove the need of specifying the dependencies between features. At the cost of losing some of the details of the dependency graph used in DrUID, with Aida we rely only on the dependencies between units. In the end, we were down to only 2 types of annotations (see Figure 7.4):

- **@TestFor** Used to indicate which unit each test is testing.
- **@ReplacesTest** Used to indicate that a test replaces another test. It also indicates that if the unit the test is related to is present in the system, then the replaced test does not have to be run.

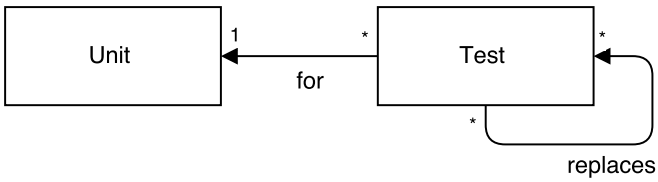

Figure 7.4: The Aida Approach

Units are defined as being contained inside Java packages by default. A third optional annotation (**@Unit**) can be used to alter this behavior.

The dependencies between units are automatically calculated by using the information provided by the JDT and AJDT Eclipse plugins. Aida is then able to show a graph of these dependencies (see Figure 7.5).

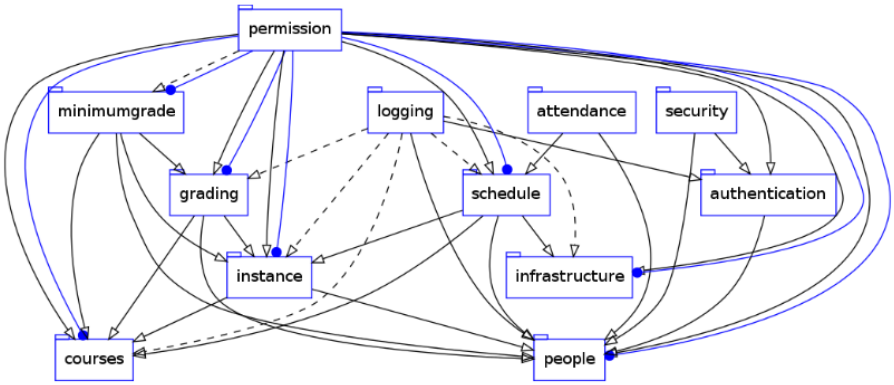

Figure 7.5: Aida: Dependency Graph

With the dependency graph calculated, the test process is very similar to that of DrUID. We start by extracting the dependency graph from the source code, then we order the units by sorting them topologically and test them adding each unit incrementally to the system. Figure 7.6 shows the interface for running all tests in a project.

After running the complete set of tests, *Aida* is capable of reporting, both graphically and in text, eventual errors and interferences detected. This allows the developer to add **@ReplaceTest** annotations, when an interaction is expected, or correct his code if the interaction was unexpected.

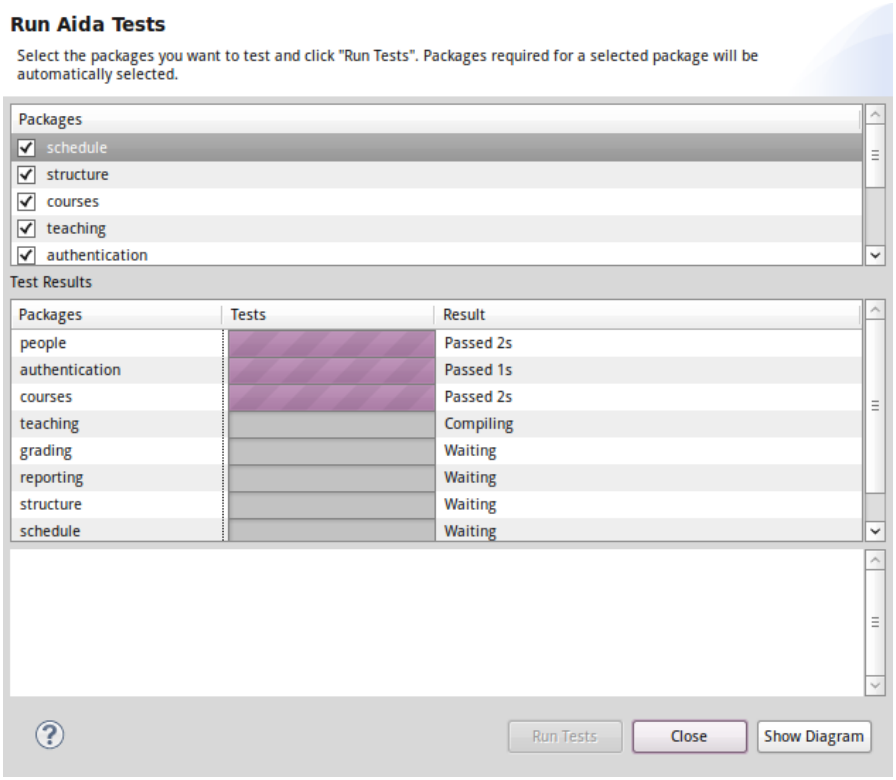

Figure 7.6: Aida: Interface for Running Tests

7.3 Current Issues

There are still some issues with the implementation of these tools. *Aida* has been a major step forward as it removes most of the burden of declaring the dependencies from the developer, but there are still a couple of issues.

The first problem is that not all dependencies can be detected. At the moment, Aida is able to detect dependencies caused by: import declarations, method and constructor calls, type declarations and advices. These encompass most of the cases, but soft dependencies, like the ones created using reflection are not detected.

The second problem is that every time the project is tested, all the tests have to be run again. This problem is augmented by the fact that most tests are being run several times.

This second problem could be mitigated by doing some code analysis to figure which tests might have their results altered by the introduction of a new unit in the incremental compilation process.

7.4 Summary

In this chapter, we described the implementation of two different plugins that are able to guide and assist developers into using the technique described throughout this document.

These two tools are capable of, not only providing assistance to the developers in the definition of the meta-data needed for the technique, but also run the entire incremental compilation and testing process. At the same time, they are able to provide a graphical representation of the dependencies between units that can prove helpful for developers as they allow the identification of unwanted circular ones.

The fact that these tools were developed and work shows that the technique is usable in the real world. However, as seen in the previous section, there are still a couple of issues that might be show stoppers in some particular situations.

Part III

Validation and Future Work

You may never know what results come of your action, but if you do nothing there will be no result.

Mahatma Gandhi

8

Validation

Contents

The best way to validate the proposed testing process would be to test it in real world applications. There are several open source AspectJ projects, some of them created exclusively for testing and validating new AOP methodologies.

The characteristics that we were looking for in such a project were that it had to be developed in AspectJ, it had to have few circular dependencies between modules and it had to have a test framework.

Unfortunately, all the existing projects fail in one of these three aspects. For example, the two most used testbed projects for AspectJ are *AJHotDraw* [MMvD07] and *Health Watcher* [GGB+07]. The first of these has an architecture with a dependency graph so complicated that most of the code is part of a mass of 14 different packages that depend on each other forming a strongly connected

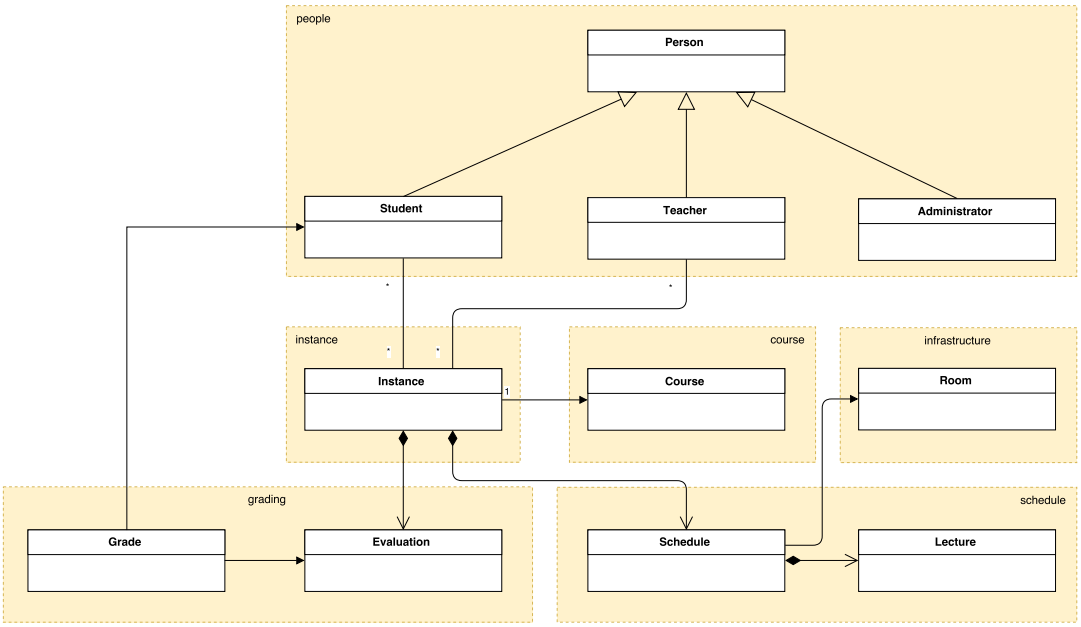

Figure 8.1: School Testbed Core Classes

component. The second one is a much cleaner project, but, unfortunately, there are no tests developed for it.

Having failed to elect a good and popular testbed where to run our testing process, we ended up developing our own testbed.

8.1 School Testbed

A simple school information system [Res14] was implemented featuring personal information for students, teacher and administrators, course information, class schedules, infrastructure information and grading. Figure 7.5 shows the dependency diagram as extracted by *Aida* while Figure 8.1 shows the base classes of the system.

Some packages containing aspects have been added to the system (see Figure 8.2). We can divide these in two groups. The first one, in blue, is composed of packages

Table 8.1: School Testbed Aspect Packages

Package	Description
Authentication	Adds a login and password attributes to the Person class. Offers methods to login and logoff as well as a way to verify who is logged in.
Attendance	Adds a list of students that attended a certain lecture and methods to manage that list.
Security	Assures that the passwords are hashed using a secure hashing algorithm. For this, it advises the methods that set and verify passwords of the Authentication module.
Permission	Verifies that the logged in user has permissions to execute the command being executed. Advises almost every method in the code in order to do this verification.
Logging	Logs to a file important information. At the moment only the creation of new objects and login attempts. To do this, it advises the object creation methods but does not change their behavior.
Minimum Grade	Adds the possibility of a course evaluation having a minimum grade that the student must attain to pass the course. Adds methods to define this minimum grade and advises the methods that calculate the student final grade.

that only declare new attributes and methods. While the second one, in purple, contains aspects that modify the behavior of the base classes by using advices. A description of these packages can be seen in Table 8.1.

8.1.1 Testing

Each one of the packages in the system was thoroughly tested. Table 8.2 lists all the test cases defined in each package.

The total number of tests amounts to 55 with most of them belonging to the *Permission* package. This happens as this package crosscuts the entire application

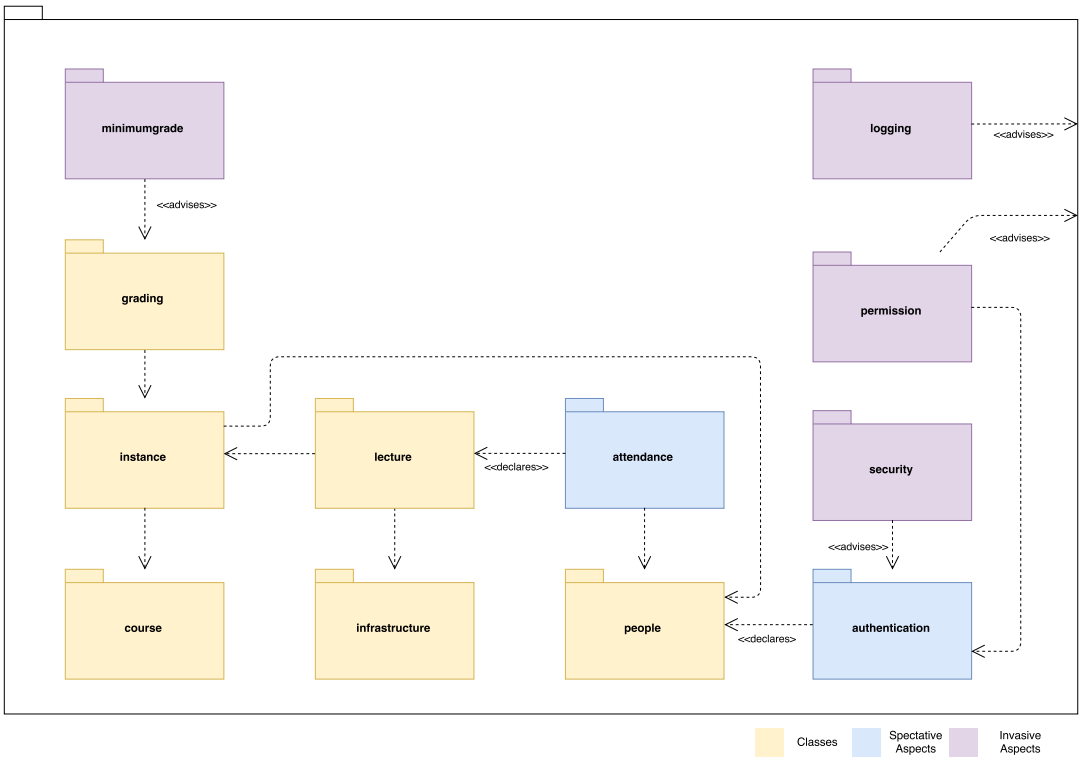

Figure 8.2: School Testbed Package Dependencies

and modifies the behavior of almost all methods by adding a permission system. This makes it important to test if those methods are still working when the user has permission to use them, and also if access is denied when the user has no permission to use them.

8.1.2 Interference Resolution

As was explained in previous chapters, when using *Aida* packages are incrementally compiled. Each time a new package is introduced into the system the complete set of tests is run.

Packages containing invasive aspects will modify the behavior of packages previously added to the system, breaking their tests. This allows us to identify potential interferences between packages. Whenever a test breaks after a new package is added, it is either a fault with the new package or an interference.

Sometimes these interferences are intentional. On those cases, we still want the complete set of tests to pass and we accomplish this by adding *@ReplaceTest* annotations on the tests that replace those that are broken, with those annotations referencing the latter. Listing 8.1 shows an example where the security *Package* has a test that verifies if passwords are not stored in plain text replacing a previous test from the *Authentication* package that tested if passwords were stored correctly.

Listing 8.1: Replacing Test Example

```
1 @ReplaceTest("authentication.tests.AuthenticationTest.testSetCredentials")
2 public void testArePasswordsHashed() {
3     Administrator admin = PersonFactory.createAdministrator("John", "Somewhere");
4     admin.setLogin("john");
5     admin.setPassword("1234");
6     assertFalse("1234".equals(admin.getPassword()));
7 }
```

By using *Aida*, interferences were easily spotted. Each time an invasive aspect was added, a test broke somewhere. In the rare event where that did not happen, it was due to an error in the implementation of the new aspect or a poorly written test. Table 8.3 shows a list of all tests for invasive aspects and the tests they replaced.

By using the technique described in this document, we were able to test all the packages of the system, in isolation, without compromising modularity.

Table 8.2: School Testbed Implemented Tests

Package	Test Cases	Total
Courses	testCreateCourse, testRemoveCourse	2
Infrastructure	testCreateRoom	1
People	testCreatePeople, testCreateStudent, testCreateTeacher, testCreateAdmin	4
Instance	testCreateInstance	1
Schedule	testCreateSchedule, testCreateLecture	2
Grading	testEvaluation, testGrading, testFinalResult	3
Attendance	testAddAttendance, testRemoveAttendance	2
Authentication	testSetCredentials, testAuthenticate, testLogoff, testWrongCredentials	4
MinimumGrade	testMinimumGrade	1
Security	testArePasswordsHashed, testAuthenticate	2
Permission	testAuthenticationPermissions, testAttendanceNoLogin, testCreateCourseNoLogin, testRemoveCourseNoLogin, testCreateCourseStudent, testCreateCourseTeacher, testCreateCourseAdmin, testRemoveCourseAdmin, testCreateEvaluationNoLogin, testCreateEvaluationNotTeacher, testCreateEvaluationNotInstanceTeacher, testCreateEvaluationInstanceTeacher, testCreateGradeNoLogin, testCreateGradeNotTeacher, testCreateGradeNotInstanceTeacher, testCreateGradeInstanceTeacher, testMinimumGrade, testCreateInstanceNoLogin, testCreateInstanceAdmin, testCreateLectureNoLogin testChangeLectureState, testCreatePeopleNoLogin, testCreatePeopleStudent, testCreatePeopleAdmin, testCreateRoomNoLogin, testCreateRoomStudent, testCreateRoomTeacher, testCreateRoomAdmin, testCreateScheduleNoLogin, testCreateScheduleStudent, testCreateScheduleTeacher, testCreateScheduleAdmin	32
All Packages		**55**

Table 8.3: School Testbed Replaced Tests

New Test	**Replaced Test**
Security.testArePasswordsHashed	Authentication.testSetCredentials
Permission.testAttendaceNoLogin	Attendance.testAddAttendance Attendance.testRemoveAttendance
Permission.testAuthenticationPermissions	Authentication.testAuthentication
Permission.testCreateCourseNoLogin Permission.testCreateCourseAsStudent/Teacher/Admin	Course.testCreateCourse
Permission.testRemoveCourseNoLogin Permission.testRemoveCourseAsAdmin	Course.testRemoveCourse
Permission.testCreateEvaluationNoLogin Permission.testCreateEvaluationNotTeacher Permission.testCreateEvaluationNotInstanceTeacher Permission.testCreateEvaluationAsInstanceTeacher	Evaluation.testEvaluation
Permission.testCreateGradeNoLogin Permission.testCreateGradeNotTeacher Permission.testCreateGradeNotInstanceTeacher Permission.testCreateGradeAsInstanceTeacher Permission.testCreateGradeNoLogin	Grade.testGrade
Permission.testMinimumGrade	MinimumGrade.testMinimumGrade
Permission.testCreateInstanceNoLogin Permission.testCreateInstanceAsAdmin	Lecture.testCreateLecture
Permission.testCreateLectureNoLogin Permission.testChangeLectureState	Lecture.testCreateLecture
Permission.testCreatePeopleNoLogin Permission.testCreatePeopleAsStudent/Teacher/Admin	Person.testCreatePeople Person.testCreateStudent/Teacher/Admin
Permission.testCreateRoomNoLogin Permission.testCreateRoomAsStudent/Teacher/Admin	Room.testCreateRoom
Permission.testCreateScheduleNoLogin Permission.testCreateScheduleAsStudent/Teacher/Admin	Schedule.testSchedule

8.1.3 Multiple Configurations

With modularity assured, we can combine the code in several different configurations. For example, we might only want the *People* and *Authentication* packages to create a simple authentication system.

The *advise* based dependencies do not have to be met in order for a combination to be valid. For example, we can combine the packages *Security* and *People* and have a functioning system even if the *Security* module advises the *Authentication* module, which would not be present in the system. Of course, without the *Authentication* package, the *Security* packaged would not be of much use, but it would still be a valid configuration.

Considering only the classical and declare dependencies, the ones that must be met, there are 77 different possible valid configurations. If we add the other 4 packages, in any possible combination, we get 8 times more possibilities. Or a grand total of 616 configurations. We were able to test all of these, successfully, using *Aida* without having to add or remove any of the tests.

8.1.4 Incompatible Modules

Sometimes modules are incompatible by design. For example, we can have two modules, with invasive aspects, advising the same module in incompatible ways. Maybe these two modules are not supposed to be used together but are alternatives to each other.

Our initial approach, or even our initial prototype *DrUID*, would be able to cope with this scenario. But *Aida*, would not. *Aida* misses the metadata necessary to understand when a feature, or a test, that is required by a module are being deprecated by another module.

We changed the metadata introduced by the developer, as we progressed our work, as a way to make the process much simpler to use. Unfortunately, at the same time, we lost some of its capabilities.

8.1.5 Performance

As expected, the amount of time needed to test the complete system was much higher due to the constant repetition of tests. Compiling the whole system took about **15 seconds** in our testing environment [1]. By using an incremental testing approach, the system had to be compiled 12 times (one for each module). Total compilation time was about **70 seconds**.

This shows a significant increase in testing time that could jeopardize the use of this approach within an agile like methodology where tests are run frequently. There are several different ways in which this problem could be mitigated:

- The developer can select small subsets of the modules appropriate to the changes he is conducting. When satisfied with these tests he can test the whole system. This would make a complete system compilation happen less frequently.
- *Aida* could be able to allow the developer to select the module that is being tested. When a certain module is selected, modules that this module depends on could be automatically considered correct. In this way, there would only be two steps involved in the compilation. First, compile the module and all modules that the module depends on. Second, compile all the modules. A complete system test would only be necessary in case of an error or, periodically, to check if everything was still correct.
- By using some kind of program slicing technique, *Aida* could be capable of selecting the tests that need to be run in each compilation step.

8.2 Incremental Testing and Common AOP Faults

A fault model identifies the relationships and components of a system under test that are most likely to have faults [Bin00]. They are important for any testing strategy as they narrow the number of places one has to look when searching for faults.

[1] Octa-core Intel i7-3632QM CPU @ 2.20GHz / 8GB

Alexander [ABAA04] has proposed a fault model for AOP that was later on extended by Ceccato [CTR05]. Deursen has also proposed his own AOP fault model [vDMM05]. In the following sections, we will describe the various fault types identified by these authors. We will also show how our approach can tackle each one of these faults using small examples.

It is not the intention of this section to point out that using our approach we can detect some faults that could not be detected using a classical approach. The intention is to show that it can still detect the faults and, at the same time, make sure the tests are in the correct modules in order to maintain modularity.

8.2.1 Incorrect Strength in Pointcut Patterns

One of the biggest selling points for an AOP language is the expressiveness level of its pointcut patterns. A good AOP language should enable developers to indicate exactly which join points they want to capture when declaring a pointcut. In reality, this is very hard to achieve.

Ideally, a developer should be able to explain the exact meaning behind each pointcut expression. However, developing a pointcut language that allows this is as difficult as developing a natural language analyzer. Instead, most AOP languages, take advantage of de-facto code standards and resort to the use of wildcards as a way of allowing developers to capture several join points in a single pointcut expression.

In this way, a developer, instead of declaring that he wishes to capture the execution of all methods of a particular class that changes its relevant internal structure, can declare that he wishes to capture the execution of all methods that start with the word *set*. This strategy may reveal itself problematic when a new method is added to the class that starts with *set* but does not belong to the specific set of methods that the developer intended to capture in the first place (e.g., a method called *setupConnections*).

The alternative would be for the developer to extensively declare exactly which methods he intended to capture. This has the obvious problem of new methods being added into the class, that should also be captured, making it hard for the developer to maintain its code. This problem is closely related to the so-called fragile pointcut problem [MC04].

Consider for example the code in Listing 8.2 describing a simple *Value* class that has 3 simple setter methods. Imagine that for this example, we have a *TestValue* class containing a *testValues* test that tests these simple methods.

Listing 8.2: Fault 1: Value Class

```
1  package value;
2
3  public class Value {
4    private int minimum, maximum, nominal;
5
6    public void setMinimum(int minimum) { this.minimum = minimum; }
7    public void setMaximum(int maximum) { this.maximum = maximum; }
8    public void setNominal(int nominal) { this.nominal = nominal; }
9  }
```

We then add a new aspect to the application that controls the maximum value of each one of the three variables of the *Value* class. Listing 8.3 contains the code to do that and Listing 8.4 contains a simplified version of the appropriate tests for this aspect. As you can see, this aspect is invasive and a *ReplaceTest* annotation is required.

Listing 8.3: Fault 1: Limits Aspect

```
1  package limits;
2
3  import value.Value;
4
5  public aspect Limits {
6    pointcut setterCalled(int value) : call(public void Value.set*(..)) && args(value);
7
8    void around(int value) : setterCalled(value) {
9      if (value > 100) throw new InvalidValueException();
10     if (value < 0) throw new InvalidValueException();
11     proceed(value);
12   }
13 }
```

Listing 8.4: Fault 1: Limits Tests

```
1  package limits;
2
3  import com.feup.contribution.aida.annotations.ReplaceTest;
4
5  import junit.framework.TestCase;
6  import value.Value;
7
8  public class TestLimits extends TestCase{
9    @ReplaceTest("value.TestValue.testValues")
10   public void testLimits() {
11     try {
```

```
12        Value value = new Value();
13        value.setMaximum(200);
14        fail("Should have failed");
15      } catch (Exception e) { }
16    }
17  }
```

If we later add a new method called *setId* to the *Value* class, that we do not which to impose any limits to, the pointcut will inaccurately select the join point that refers to the call of this new method. This will make any tests done to this method break as intended. At the same time, we ensured that all tests are in their own units of modularity.

8.2.2 Incorrect Aspect Precedence

In Section 3.2.5 we have seen how the order in which aspects are woven into the system can be changed using the *declare precedence* clause. Incorrect ordering of aspects, especially when they interact through state variables, can affect the overall system behavior.

For example, an aspect that ciphers a password for security reasons and an aspect that tests if passwords are sufficiently strong have to be run in the correct order. If the cipher aspect runs first, the other aspect will be unable to inspect the password.

Lets consider a simple example describing a telecommunication system. In this example, the *Call* class represents a simple phone call. The class can be in three different states: waiting (the caller is waiting for the callee to answer), connected (the connection is established) and finished (one of the two parties hanged up). The class goes through these states as the constructor, *connect* and *hangup* methods are called (see Listing 8.5).

Listing 8.5: Fault 2: Call Class

```
1  package call;
2
3  import com.feup.contribution.aida.annotations.PackageName;
4
5  @PackageName("Call")
6  public class Call {
7    public enum STATE {WAITING, CONNECTED, FINISHED};
8
9    private STATE state;
10
```

```
11    public Call() {
12      this.state = STATE.WAITING;
13    }
14
15    public void connect() {
16      if (this.state == STATE.WAITING) this.state = STATE.CONNECTED;
17    }
18
19    public void hangup() {
20      if (this.state == STATE.CONNECTED) this.state = STATE.FINISHED;
21    }
22
23    public STATE getState() {
24      return state;
25    }
26  }
```

To this simple system (having only one class), we added two aspects. The first one controls the amount of time spent in a single call (see Listing 8.6); the second one, calculates how much should be payed for that same call (see Listing 8.7). Both these aspects are triggered at the same join point (when the call is finished) and this leads to a precedence problem as the *billing* aspect needs information calculated by the *timing* aspect to work properly.

Listing 8.6: Fault 2: Timing Aspect

```
1   package timing;
2
3   import com.feup.contribution.aida.annotations.PackageName;
4
5   import call.Call;
6
7   @PackageName("Timing")
8   public aspect CallTiming {
9     private long Call.callStart;
10    private long Call.callStop;
11
12    pointcut connected(Call c) : execution(public void Call.connect(..)) && this(c);
13    pointcut finished(Call c) : execution(public void Call.hangup(..)) && this(c);
14
15    after(Call c) returning() : connected(c) {
16      c.callStart = System.currentTimeMillis();
17    }
18
19    after(Call c) returning() : finished(c) {
20      c.callStop = System.currentTimeMillis();
21    }
22
23    public long Call.getDuration() {
24      return callStop - callStart;
25    }
26  }
```

Listing 8.7: Fault 2: Billing Aspect

```
1  package billing;
2
3  import com.feup.contribution.aida.annotations.PackageName;
4
5  import call.Call;
6  import timing.CallTiming;
7
8  @PackageName("Billing")
9  public aspect CallBilling {
10   private double Call.cost;
11
12   declare precedence: CallBilling, CallTiming;
13
14   pointcut finished(Call c) : execution(public void Call.hangup(..)) && this(c);
15
16   after(Call c) returning() : finished(c) {
17     c.cost = Math.round(c.getDuration() / 1000) * 0.25;
18   }
19
20   public double Call.getCost() {
21     return cost;
22   }
23 }
```

Listing 8.8: Fault 2: Billing Aspect Test

```
1  package billing;
2
3  import junit.framework.TestCase;
4  import call.Call;
5
6  import com.feup.contribution.aida.annotations.TestFor;
7
8  @TestFor("Billing")
9  public class CallBillingTest extends TestCase {
10   public void testTiming() {
11     Call c = new Call();
12     c.connect();
13     try { Thread.sleep(2000); } catch (InterruptedException e) { }
14     c.hangup();
15     assertEquals(0.5, c.getCost());
16   }
17 }
```

Listing 8.9: Fault 2: Timing Aspect Test

```
1  package timing;
2
3  import com.feup.contribution.aida.annotations.TestFor;
4
```

```
5  import call.Call;
6  import junit.framework.TestCase;
7
8  @TestFor("Timing")
9  public class CallTimingTest extends TestCase {
10   public void testTiming() {
11     Call c = new Call();
12     assertEquals(0, c.getDuration());
13     c.connect();
14     try { Thread.sleep(1000); } catch (InterruptedException e) { }
15     c.hangup();
16     assertTrue(c.getDuration() > 0 && c.getDuration() < 2000);
17   }
18 }
```

This could be solved by adding a *declare precedence* instruction stating that the *timing* aspect had precedence over the *billing* aspect.

By using our proposed approach, the testing system would be able to infer the different orders in which the system could be composed. Neither one of the two possible orders would give a different result. Both would work or not work depending on the default precedence order. This would show the user that the problem is not an interference between aspects allowing him to explore other possible candidate faults. As soon as the problem has been fixed, the testing procedure would work without problems.

8.2.3 Failure to Preserve Postconditions and State Invariants

Clients expect method postconditions to be preserved even when aspects are woven into the system. The modification of method postconditions can cause method clients to behave in unexpected ways.

However, in some cases, the modification of postconditions and the subsequent modification of the behavior of clients might be the desired result of an aspect. This has proved to be a major source of errors in AOP applications.

In this section, an example illustrating the failure to preserve a postcondition (and a state invariant) after an aspect is applied is demonstrated.

The *Transfer* class represents a transfer between accounts (see Listing 8.10). The *TransferList* class represents several of these transfers and keeps a cache holding

the total amount of all the transfers it contains. The *addTransfer* method of the *TransferList* class has an implicit *postcondition* that states that the saved amount after its execution must be equal to the original value plus the value of the added transfer. The class also has an implicit state invariant that states that the total amount saved in the cache must be equal to the sum of the amounts of all transfers (see Listing 8.11).

Listing 8.10: Fault 3: Transfer Class

```
1  package transfer;
2  import com.feup.contribution.aida.annotations.PackageName;
3
4  @PackageName("Transfer")
5  public class Transfer {
6    private int ammount;
7
8    public Transfer(int ammount) {
9      setAmmount(ammount);
10   }
11
12   public void setAmmount(int ammount) {
13     this.ammount = ammount;
14   }
15
16   public int getAmmount() {
17     return ammount;
18   }
19 }
```

Listing 8.11: Fault 3: TransferList Class

```
1  package transfer;
2  import java.util.LinkedList;
3
4  import com.feup.contribution.aida.annotations.PackageName;
5
6  @PackageName("Transfer")
7  public class TransferList {
8    private LinkedList<Transfer> transfers = new LinkedList<Transfer>();
9    private int total = 0;
10
11   // Postcondition: total = total + t.getAmmount()
12   public void addTransfer(Transfer t) {
13     transfers.add(t);
14     total += t.getAmmount();
15   }
16
17   public int getTotal() {
18     return total;
19   }
20
21   public void setTotal(int total) {
```

```
22      this.total = total;
23    }
24  }
```

If we add an aspect that allows transfers to be verified (or not), and we state that the total amount of a transfer list is equal to the sum of the amounts of only the verified transfers, the postcondition and the state invariant are not preserved (see Listing 8.12).

Listing 8.12: Fault 3: VerifiedTransferOnly Aspect

```
1  package verified;
2  import transfer.Transfer;
3  import transfer.TransferList;
4
5  import com.feup.contribution.aida.annotations.PackageName;
6
7  @PackageName("VerifiedTransfer")
8  public aspect VerifiedTransferOnly {
9    private boolean Transfer.isVerified;
10
11   public void Transfer.setIsVerified(boolean isVerified) {
12     this.isVerified = isVerified;
13   }
14
15   pointcut transferAdded(TransferList tl, Transfer t) :
16     call(public void TransferList.addTransfer(..)) && target(tl) && args(t);
17
18   after(TransferList tl, Transfer t) : transferAdded(tl, t) {
19     if (!t.isVerified) tl.setTotal(tl.getTotal() - t.getAmmount());
20   }
21 }
```

A test for the *addTransfer* method will pass if the aspect is not compiled into the code but fail if it is (see Listing 8.13).

Listing 8.13: Fault 3: Transfer List Tests

```
1  package transfer;
2  import com.feup.contribution.aida.annotations.TestFor;
3
4  import junit.framework.TestCase;
5
6  @TestFor("Transfer")
7  public class TestTransferList extends TestCase{
8
9    public void testTotal() {
10     TransferList tl = new TransferList();
11     tl.addTransfer(new Transfer(10));
12     tl.addTransfer(new Transfer(20));
13     tl.addTransfer(new Transfer(40));
14     tl.addTransfer(new Transfer(25));
```

```
15      assertEquals(95, tl.getTotal());
16    }
17  }
```

By adding an annotation to the *VerifiedTransferOnly* test class, that states that the test *testVerifiedTransfer* replaces the previous test, and then test the system incrementally, both tests are run and both tests will pass. The first test will run without the aspect compiled while the second one will run with the aspect (see Listing 8.14).

Listing 8.14: Fault 3: Verified Transfer Only tests

```
1  package verified;
2
3  import junit.framework.TestCase;
4  import transfer.Transfer;
5  import transfer.TransferList;
6
7  import com.feup.contribution.aida.annotations.ReplaceTest;
8  import com.feup.contribution.aida.annotations.TestFor;
9
10 @TestFor("VerifiedTransfer")
11 public class TestVerifiedTransferOnly extends TestCase{
12
13   @ReplaceTest("transfer.TestTransferList.testTotal")
14   public void testVerifiedTransfer() {
15     TransferList tl = new TransferList();
16     Transfer t1 = new Transfer(10);
17     t1.setIsVerified(true); tl.addTransfer(t1);
18     Transfer t2 = new Transfer(20);
19     t2.setIsVerified(true); tl.addTransfer(t2);
20     Transfer t3 = new Transfer(40);
21     t3.setIsVerified(false); tl.addTransfer(t3);
22     Transfer t4 = new Transfer(25);
23     t4.setIsVerified(true); tl.addTransfer(t4);
24     assertEquals(55, tl.getTotal());
25   }
26 }
```

8.2.4 Incorrect Focus of Control Flow

Sometimes join points should only be selected in a particular context. For example, in a recursive call, we might want to select only the join point of the first call and not the subsequent calls. Failure to restrict the selection to the proper context can lead to extremely hard to reason problems.

In this section, we demonstrate a incorrect focus of control flow fault in AspectJ. Listing 8.15 shows a simple example of a class capable of sorting lists containing integer values using a recursive algorithm. For simplicity sake, this algorithm is not optimized in any way.

Listing 8.15: Fault 4: QuickSort Class

```
1  package sort;
2  import java.util.LinkedList;
3
4  import com.feup.contribution.aida.annotations.PackageName;
5
6  @PackageName("Sort")
7  public class QuickSort {
8    public static LinkedList<Integer> sort(LinkedList<Integer> list) {
9      if (list.size() <= 1) return list;
10     Integer pivot = list.getFirst();
11     list.removeFirst();
12     LinkedList<Integer> smaller = new LinkedList<Integer>();
13     LinkedList<Integer> greater = new LinkedList<Integer>();
14     for (Integer v : list) {
15       if (v.intValue() < pivot.intValue()) smaller.add(v);
16       else greater.add(v);
17     }
18     LinkedList<Integer> ret = new LinkedList<Integer>();
19     ret.addAll(sort(smaller));
20     ret.add(pivot);
21     ret.addAll(sort(greater));
22     return ret;
23   }
24 }
```

Listing 8.16 contains a simple test that asserts that the sorted list contains all elements of the original list and they are sorted. This test is done by generating 1000 random numbers and sorting them using the class in Listing 8.15.

Listing 8.16: Fault 4: QuickSort Tests

```
1  package sort;
2  import java.util.LinkedList;
3  import java.util.Random;
4
5  import com.feup.contribution.aida.annotations.TestFor;
6
7  import junit.framework.TestCase;
8
9  @TestFor("Sort")
10 public class QuickSortTest extends TestCase{
11   public void testSort() {
12     LinkedList<Integer> list = new LinkedList<Integer>();
13     LinkedList<Integer> copy = new LinkedList<Integer>();
14     Random random = new Random();
```

```
15      for (int i = 0; i < 10; i++) {
16        Integer v = new Integer(random.nextInt(10000));
17        list.add(v);
18        copy.add(v);
19      }
20      list = QuickSort.sort(list);
21      assertEquals(copy.size(), list.size());
22      for (Integer v : copy) {
23        assertTrue(list.contains(v));
24      }
25      for (int i = 0; i < list.size() - 1; i++) {
26        assertTrue(list.get(i).intValue() <= list.get(i + 1).intValue());
27      }
28    }
29  }
```

The aspect introduced in Listing 8.17 modifies the behavior of this algorithm by applying an around advice at the join point represented by the call to the sorting algorithm and returning an inverted version of the returning result. Line 12 of this aspect is extremely important. It prevents the advice from being applied in every step of the recursive algorithm which would result in an unsorted list.

Listing 8.17: Fault 4: Invert Sort Aspect

```
1   package invert;
2   import java.util.LinkedList;
3
4   import com.feup.contribution.aida.annotations.PackageName;
5
6   import sort.QuickSort;
7
8   @PackageName("Invert")
9   public aspect InvertSort {
10    pointcut listSorted() :
11      call(public LinkedList<Integer> QuickSort.sort(..)) &&
12      !within(QuickSort);
13
14    LinkedList<Integer> around() : listSorted() {
15      LinkedList<Integer> list = proceed();
16      LinkedList<Integer> inverted = new LinkedList<Integer>();
17      for (Integer v : list) {
18        inverted.addFirst(v);
19      }
20      return inverted;
21    }
22  }
```

Listing 8.18 tests the inverted version of the quicksort algorithm. It also states that the inverted sort test replaces the original sort test. This prevents the original test from throwing an error if run with the inverted aspect.

By removing line 12 of Listing 8.17, the original sorting test will still pass, as expected, but the inverted sort test will fail.

Listing 8.18: Fault 4: Invert Sort Tests

```
1  package invert;
2  import java.util.LinkedList;
3  import java.util.Random;
4
5  import com.feup.contribution.aida.annotations.ReplaceTest;
6  import com.feup.contribution.aida.annotations.TestFor;
7
8  import sort.QuickSort;
9
10 import junit.framework.TestCase;
11
12 @TestFor("Invert")
13 public class TestInverted extends TestCase{
14   @ReplaceTest("sort.QuickSortTest.testSort")
15   public void testInverted() {
16     LinkedList<Integer> list = new LinkedList<Integer>();
17     LinkedList<Integer> copy = new LinkedList<Integer>();
18     Random random = new Random();
19     for (int i = 0; i < 10; i++) {
20       Integer v = new Integer(random.nextInt(10000));
21       list.add(v);
22       copy.add(v);
23     }
24     list = QuickSort.sort(list);
25     assertEquals(copy.size(), list.size());
26     for (Integer v : copy) {
27       assertTrue(list.contains(v));
28     }
29     for (int i = 0; i < list.size() - 1; i++) {
30       assertTrue(list.get(i).intValue() >= list.get(i + 1).intValue());
31     }
32   }
33 }
```

8.2.5 Incorrect Changes in Control Dependencies

Around advices can significantly alter the behavior semantics of a method. Faults may arise from assumptions on control dependencies that are no longer valid in the woven code.

The previous example can also be used to show an incorrect change in control dependencies fault. The aspect introduced in Listing 8.17, modifies the control dependency by applying an around advice at the join point represented by the call to the sorting algorithm and returning an inverted version of the returning result.

The sorting test depicted in listing 8.16 will fail if the aspect defined in listing 8.17 is added to the system. However, as this is the intended behavior, the invert sort tests added in listing 8.18 state that they replace the original sort test. In this way, when compiled incrementally both tests are run and both tests succeed.

8.2.6 Incorrect Changes in Exceptional Control Flow

Listing 8.19 contains a simple property management class that can be used by applications to manage different configuration files. This base code does not cope with default values for properties that are not mentioned in the properties file or with files that are missing altogether.

Listing 8.19: Fault 6: Config Class

```
1  package config;
2
3  import java.io.FileInputStream;
4  import java.io.FileNotFoundException;
5  import java.io.IOException;
6  import java.util.Properties;
7
8  public class Config {
9    Properties properties;
10
11   public Config(String filename)
12     throws FileNotFoundException, IOException {
13     properties = new Properties();
14     properties.load(new FileInputStream(filename));
15   }
16
17   public String getValue(String key) {
18     return (String) properties.getProperty(key);
19   }
20 }
```

Listing 8.20 contains some tests for this class. Besides testing for normal behavior, these also test if exceptions are correctly thrown when a file is not found and if *null* is returned for nonexisting properties. Without any aspects applied all tests pass.

Listing 8.20: Fault 6: Config Tests

```
1  package config;
2
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5
```

```
6  import junit.framework.TestCase;
7
8  public class TestConfig extends TestCase {
9    public void testConfig()
10     throws FileNotFoundException, IOException {
11     Config config = new Config("test.properties");
12     assertEquals("foo", config.getValue("bar"));
13   }
14
15   public void testNull()
16     throws FileNotFoundException, IOException {
17     Config config = new Config("test.properties");
18     assertNull(config.getValue("foo"));
19   }
20
21   public void testFileNotFound() throws IOException {
22     try {
23       new Config("unexistant.properties");
24     } catch (FileNotFoundException e) {
25       return;
26     }
27     fail("Did not return FileNotFoundException");
28   }
29 }
```

Listing 8.21 adds an aspect to the system that allows for the existence of a default configuration file. When a non existent configuration file is loaded, the default configuration file is loaded instead. This behavior breaks the *testFileNotFound* test found in Listing 8.20. When a property is not found in the loaded configuration file, the aspect tries to find it in the default configuration file. This behavior breaks the *testNull* test.

Listing 8.21: Fault 6: Default Values Aspect

```
1  package defaultvalues;
2
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5
6  import config.Config;
7
8  public aspect DefaultValues {
9    pointcut configCreated() :
10     call(public Config.new(..)) &&
11     !within(DefaultValues);
12
13   pointcut getValue(String key) :
14     call(public String Config.getValue(..)) &&
15     args(key) && !within(DefaultValues);
16
17   Config around()
18     throws FileNotFoundException, IOException : configCreated() {
```

```
19      try {
20        Config config = proceed();
21        return config;
22      } catch (FileNotFoundException e) {
23        return new Config("default.properties");
24      }
25    }
26
27    String around(String key) : getValue(key) {
28      String value = proceed(key);
29      if (value == null) {
30        Config defaultValues;
31        try {
32          defaultValues = new Config("default.properties");
33          return defaultValues.getValue(key);
34        } catch (FileNotFoundException e) {
35        } catch (IOException e) {
36        }
37      }
38      return value;
39    }
40  }
```

Listing 8.22 contains the alternative tests for this aspect. These include two tests that replace the broken ones. By compiling incrementally, all tests are run at least once and all tests pass.

Listing 8.22: Fault 6: Default Values Tests

```
1   package defaultvalues;
2
3   import java.io.FileNotFoundException;
4   import java.io.IOException;
5
6   import com.feup.contribution.aida.annotations.ReplaceTest;
7
8   import config.Config;
9   import junit.framework.TestCase;
10
11  public class TestDefaultValues extends TestCase {
12    @ReplaceTest("config.TestConfig.testFileNotFound")
13    public void testUnexistantFile()
14      throws FileNotFoundException, IOException {
15      Config config = new Config("unenxistant.properties");
16      assertEquals("notfoo", config.getValue("bar"));
17    }
18
19    @ReplaceTest("config.TestConfig.testNull")
20    public void testUnexistantProperty()
21      throws FileNotFoundException, IOException {
22      Config config = new Config("test.properties");
23      assertEquals("bar", config.getValue("foo"));
24    }
```

```
25
26    public void testUnexistantDefault()
27      throws FileNotFoundException, IOException {
28      Config config = new Config("test.properties");
29      assertNull(config.getValue("notfoo"));
30    }
31  }
```

8.2.7 Failures due to Inter-type Declarations

If some part of the code depends on the static class structure of the application, changing that structure can alter the behavior of the application in unexpected ways.

This section shown an example of the ripples that can be caused by the introduction of inter-type declarations when the control flow depends on the static class structure.

Listings 8.23, 8.24, 8.25 and 8.26 depict a simple class hierarchy between users, employees, and managers. *User* is a superclass of both *Employee* and *Manager* but only the last of these two implements the *Administrator* interface.

Listing 8.27, shows a hypothetical operation that should only be performed by administrators (in this case by managers). Listing 8.28 shows two tests that verify that this is the case.

Listing 8.29 shows an aspect that changes the class hierarchy making the class *Employee* implement the *Administrator* interface. As soon as this module is introduced, the tests described in the previous paragraph will fail. By adding the tests for this new module, as depicted in Listing 8.30, and adding a replace test annotation, the system can be compiled and tested fully with or without the new invasive module.

Listing 8.23: Fault 7: User Class

```
1  package users;
2
3  public class User {
4    private static User currentUser;
5
6    public static void setCurrentUser(User user, String password) {
7      if (user.getPassword() == password) currentUser = user;
8    }
9
```

```
10    public static User getCurrentUser() {
11      return currentUser;
12    }
13
14    public User(String password) {
15      this.password = password;
16    }
17
18
19    public void setPassword(String password) {
20      this.password = password;
21    }
22    public String getPassword() {
23      return password;
24    }
25
26    private String password;
27  }
```

Listing 8.24: Fault 7: Employee Class

```
1  package users;
2
3  public class Employee extends User{
4    public Employee(String password) {
5      super(password);
6    }
7  }
```

Listing 8.25: Fault 7: Manager Class

```
1  package users;
2
3  public class Manager extends User implements Administrator{
4    public Manager(String password) {
5      super(password);
6    }
7  }
```

Listing 8.26: Fault 7: Administrator Interface

```
1  package users;
2
3  public interface Administrator {
4
5  }
```

Listing 8.27: Fault 7: Operation Class

```
1  package operation;
2
3  import users.Administrator;
4  import users.User;
```

```
5
6  public class Operation {
7    public void perform() throws Exception {
8      if (User.getCurrentUser() != null &&
9          User.getCurrentUser() instanceof Administrator) {
10       //doSomething
11     } else throw new Exception("Not␣allowed␣to␣perform␣operation");
12   }
13 }
```

Listing 8.28: Fault 7: Operation Tests

```
1  package operation;
2
3  import users.Employee;
4  import users.Manager;
5  import users.User;
6  import junit.framework.TestCase;
7
8  public class TestOperation extends TestCase {
9    public void testEmployee() {
10     Employee p = new Employee("abcd");
11     User.setCurrentUser(p, "abcd");
12     Operation o = new Operation();
13     try {
14       o.perform();
15     } catch (Exception e) {
16       return;
17     }
18     fail("Employee␣should␣not␣be␣able␣to␣perform␣operation");
19   }
20
21   public void testManager() {
22     Manager m = new Manager("abcd");
23     User.setCurrentUser(m, "abcd");
24     Operation o = new Operation();
25     try {
26       o.perform();
27     } catch (Exception e) {
28       fail("Manager␣should␣be␣able␣to␣perform␣operation");
29     }
30   }
31 }
```

Listing 8.29: Fault 7: Super Employee Aspect

```
1  package superemployee;
2
3  import users.*;
4
5  public aspect SuperEmployee {
6    declare parents: Employee implements Administrator;
7  }
```

Listing 8.30: Fault 7: Super Employee Tests

```
1  package superemployee;
2
3  import com.feup.contribution.aida.annotations.ReplaceTest;
4
5  import operation.Operation;
6  import users.Employee;
7  import users.User;
8  import junit.framework.TestCase;
9
10 public class TestSuperEmployee extends TestCase {
11   @ReplaceTest("operation.TestOperation.testEmployee")
12   public void testSuperEmployee() {
13     Employee p = new Employee("abcd");
14     User.setCurrentUser(p, "abcd");
15     Operation o = new Operation();
16     try {
17       o.perform();
18     } catch (Exception e) {
19       fail("Super employee should be able to perform operation");
20     }
21   }
22 }
```

8.2.8 Incorrect Changes in Polymorphic Calls

When aspect oriented programming is used to override a method inherited from a super class, calls to that method will start redirecting to the overwritten method instead of the superclass one. This might also be the cause of unexpected behavior.

This section shows an example of an error caused by overriding a method using an introduction.

Listing 8.31 and 8.32 show two simple classes called *User* and *Administrator*. The second of these classes is an extension of the other. The *User* class defines a method to set its password. This method is tested as can be seen in Listing 8.33.

Listing 8.31: Fault 8: User Class

```
1  package user;
2
3  public class User {
4    private String password;
5
6    public void setPassword(String password) {
7      this.password = password;
8    }
9  }
```

Listing 8.32: Fault 8: Administrator class

```
1  package user;
2
3  public class Administrator extends User{ }
```

Listing 8.33: Fault 8: User Tests

```
1  package user;
2
3  import junit.framework.TestCase;
4
5  public class TestUser extends TestCase{
6    public void testUserPassword() throws InterruptedException {
7      User user = new User();
8      user.setPassword("1234");
9    }
10
11   public void testAdminPassword() throws InterruptedException {
12     Administrator admin = new Administrator();
13     admin.setPassword("1234");
14   }
15 }
```

When the aspect described in Listing 8.34 is introduced, the test fails even though the aspect does not change the previous code but only introduces new code in the *Administrator* class.

Listing 8.34: Fault 8: Strong Password Aspect

```
1  package security;
2
3  import user.Administrator;
4
5  public aspect StrongPassword {
6    public void Administrator.setPassword(String password) {
7      if (password.length() <= 6) throw new SecurityException("Password␣is␣too␣small");
8      super.setPassword(password);
9    }
10 }
```

8.3 Summary

Two different approaches have been taken to validate the proposed approach.

In the first one, we created a small testbed that fulfilled the expectations we had for a well written AspectJ application. The developed testing process has then been applied to this testbed with success. The development of the testbed has

been coupled with the application of the testing process from the beginning. This allowed us to validate the feasibility of using the process in a real world situation. During the whole coding and testing process, the tools developed to implement the process have been helpful and easy to use.

The second approach was to apply the process to common AOP faults. The idea behind this approach, was not to test if the process was able to pinpoint these faults. Classical testing procedures already have no problem with that. What we wanted to validate, was that the faults were still detectable and the needed tests did not violate any modularity principles.

"Se podes olhar, vê. Se podes ver, repara."

José Saramago – 1995

9

Conclusions

Contents

Over the years, and after countless advances in software development, modularity continues to be, if not the most, at least one of the most important principles that support this craft. It is also one tough nut to crack, as no single paradigm, technique or tool has been completely successful at allowing developers to create perfectly modular programs.

Crosscutting concerns are one of those things that make a developer cringe when he is trying to create the perfect software design. Designs often start as clean and elegant but as soon as they become more complex, they become tangled and ugly. AOP was the first time I had hope for a good enough fix for the problem.

Unfortunately, the more I played around with AOP, the more convinced I became that Fred Brooks is still correct [Bro87], it really seems there is no silver bullet. There are still too many problems to address for AOP to become a widespread

solution. Finding a semantically strong join point model that allows developers to create better pointcuts, the added difficulty in understanding AOP code, and the fact that code is not the only type of artifact that gets tangled and scattered, are just some of the problems that make AOP not ready to become the solution for every single modularity problem. I still think AOP is useful in a plethora of situations, but it is my firm believe that it should not be overused.

9.1 Contributions

In this dissertation, we identified one of these problems (see Chapter 5). It is perfectly possible to use unit tests in conjunction with AOP. The problem is that these unit tests must always be prepared to test the system after the advises from any module containing aspects has been applied. If these aspects are invasive, then the tests are not testing the unit in isolation and they stop being unit tests.

The solution we proposed is based on having tests, that test modules that contain invasive aspects, annotated in such a way that they announce which tests test the functionality being modified by those aspects (see Chapter 6). Having these annotations in place would allow a testing technique based on incremental compilation that could test units in lower layers of the software, using their own unit tests, separately from invasive aspects from higher layers. We argued that this is similar to what *stubs*, *mocks*, *dummies* and *fakes* contributed to classical unit testing.

We do not argue that the proposed solution is usable in every situation, but we have shown that it can be used in several different scenarios (see Chapter 8). We envision it being used in software houses that have a large repository of modules that can be combined in different ways in order to compose different software solutions. Anyone that has tried to create such a system knows that crosscutting concerns are a big issue. AOP helps in achieving this *dream* but, as we have shown, unit tests would have to be created for each particular project or some of the code would not be tested at all; at least not in isolation.

The presented approach stemmed from the thesis that:

> *An incremental testing solution allows developers to keep the promise of modularity achieved by using AOP, without compromising the outcome of the testing process.*

Pursuing this solution produced four main contributions to the to the body of knowledge in software engineering:

- **The problem.** A detailed explanation, with examples, of the problems introduced by testing in aspect oriented programming. We have shown that using unit tests together with aspect oriented programming either breaks the modularity or the isolation of those tests (see Section 5.2).
- **The approach.** A technique that allows the usage of testing procedures with aspect oriented programming without breaking the modularity that it strives to achieve. We have shown that this technique can be used in several different scenarios making testing and aspect oriented programming more compatible (see Chapter 6).
- **The testbed.** A test application that can be used by the community to study the interactions between aspects and unit testing. We sincerely hope that the community uses and builds on top of this testbed as there are very few good sample applications that are oriented for academic purposes (see Section 8.1).
- **The implementation.** A plugin for Eclipse that supports the developed approach. Still a prototype but it can showcase the approach in a competent way (see Chapter 7).

Following are some of the publications done during the course of this work (see also Appendix A):

[RA07] André Restivo and Ademar Aguiar. Towards detecting and solving aspect conflicts and interferences using unit tests. In *Proceedings of the 5th Workshop on Software-Engineering Properties of Languages and Aspect Technologies (SPLAT'07)*, pages 1–5, Vancouver BC, Canada, 2007.

[RA08] André Restivo and Ademar Aguiar. Disciplined composition of aspects using tests. In *Proceedings of the 2008 AOSD Workshop on Linking Aspect Technology and Evolution*, LATE '08, pages 8:1–8:5, New York, NY, USA, 2008. ACM.

[RA09a] André Restivo and Ademar Aguiar. Testing for unexpected interactions in AOP. In *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*, pages 548–552. IEEE, 2009.

9.2 Future Work

As the work done in this thesis was being completed, several new ideas and ramifications of the work being done have been collected. We will now present some of them as pointers for future work.

More Testing. The approach has been tested on several small projects created just for the effect. These case studies are no substitute for testing on a full-fledged project. Unfortunately, there was no opportunity for this to happen during the time frame of this work but it would be interesting to see this work applied in the *real* world.

Plugin Improvements. The plugin developed during this work is nothing more than a crude prototype. We feel that it could be much improved by way of better auto-complete, auto-correct and automatic suggestions.

Performance. We have shown that the approach has several performance issues that might hinder its usage in larger projects. Using a smarter strategy to select the tests that are needed to run after a certain code is modified, or even to select which tests must be run after a certain aspect has been applied, would go a long way into making the process faster. We argue that this could be accomplished, in a future work, using code slicing.

Product Lines. We talked several times about the usefulness of this approach in situations where large repositories of modules exist. SPLs are one such case. Introducing this idea into the software product line research field would also be an interesting continuation of this work.

Other Artifacts. This dissertation only addressed the problem of unit tests and AOP, but there are several other artifacts that suffer from the same problem. One example is documentation. Be it technical documentation or user manuals, AOP still has to address the issue of their modularity. We think the presented approach could be easily ported to these other artifacts.

Other Languages. We only tested the approach on one language, AspectJ. Other aspect oriented programming languages exist, some of them with radically different approaches. It would be interesting to study how incremental compilation could be used in other languages.

Bibliography

[ABAA04] Roger T. Alexander, James M. Bieman, and Anneliese A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. Technical report, Department of Computer Science, Colorado State University, Fort Collins, Colorado, 2004. Cited on pp. 74 and 116.

[AEB01] Omar Aldawud, Tzilla Elrad, and Atef Bader. A UML profile for aspect oriented modeling. In Kris De Volder, Maurice Glandrup, Siobhán Clarke, and Robert Filman, editors, *Workshop on advanced separation of concerns in object-oriented systems (oopsla 2001)*, 2001. Cited on p. 38.

[AJD10] AJDT. Eclipse AspectJ development tools (AJDT) home page, December 2010. http://www.eclipse.org/ajdt/. Cited on pp. 43 and 97.

[AKJ06] Andy Kellens, Kris Gybels, and Johan Brichau. A Model-driven Pointcut Language for More Robust Pointcuts. In *Software Engineering Properties of Languages for Aspect Technology (SPLAT) workshop, AOSD 2006*, 2006. Cited on p. 44.

[AMBR02] João Araújo, Ana Moreira, Isabel Brito, and Awais Rashid. Aspect-oriented requirements with UML. In Mohamed Kandé, Omar Aldawud, Grady Booch, and Bill Harrison, editors, *Second international workshop on aspect-oriented modeling with uml (uml 2002)*, 2002. Cited on p. 36.

[Asp10a] AspectJ Team. AspectJ Programming Guide. http://bit.ly/9GrAHM, 14 July 2010. Cited on pp. 29 and 30.

[Asp10b] AspectJ Team. AspectJ Programming Guide : Language Semantics. http://bit.ly/qqcirO, 14 July 2010. Cited on p. 27.

[Asp15] AspectJ Team. ajdoc, the aspectj documentation tool. https://eclipse.org/aspectj/doc/released/devguide/ajdoc-ref.html, 8 2015. Cited on p. 43.

[AT98] Mehmet Aksit and Bedir Tekinerdogan. Aspect-Oriented Programming Using Composition Filters. In *Proceedings of the AOP Workshop at ECOOP'98*, 1998. Cited on p. 33.

[AWK04] João Araújo, Jon Whittle, and Dae-Kyoo Kim. Modeling and composing scenario-based requirements with aspects. In *Proceedings of the requirements engineering conference, 12th ieee international (re'04)*, pages 58–67, Washington, DC, USA, 2004. IEEE Computer Society. Cited on p. 36.

[BA92] Jan Bosch and Mehmet Aksit. Composition-Filters Based Real-Time Programming. In *Proceedings of the OOPSLA'92 Workshop on Evaluation of Object-Oriented Technology in Real-Time Systems*, 1992. Cited on p. 33.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM. Cited on p. 16.

[BC04] Elisa Baniassad and Siobhan Clarke. Finding aspects in requirements with theme/doc. In Bedir Tekinerdou gan, Ana Moreira, João Araújo, and Paul Clements, editors, *In proceedings of early aspects 2004 workshop*, March 2004. Cited on p. 36.

[Bec94] Kent Beck. Simple Smalltalk Testing: With Patterns. Technical report, First Class Software, Inc., 1994. Cited on p. 49.

[Ber88] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, 21 March 1988. Cited on p. 13.

[BGKV06] Mathieu Braem, Kris Gybels, Andy Kellens, and Wim Vanderperren. Inducing evolution-robust pointcuts. In *Proceedings of the International ERCIM Workshop on Software Evolution, Lille, France*, volume 180, 2006. Cited on p. 72.

[Bin99] Robert Binder. *Testing object-oriented systems: models, patterns, and tools (the addison-wesley object technology series).* Addison-Wesley Professional, October 1999. Cited on p. 42.

[Bin00] Robert Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 2000. Cited on p. 115.

[BJ94] Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994. Cited on pp. 13 and 15.

[BM04] D. Balzarotti and M. Monga. Using program slicing to analyze aspect-oriented composition, 2004. Cited on p. 68.

[BR08] Cristiano Breuel and Francisco Reverbel. User-Defined Join Point Selectors. *Journal of Object Technology*, 7(9):5–24, December 2008. Special Issue: SPLAT. Cited on p. 24.

[Bro87] Frederick. P. Brooks. No Silver Bullet - Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, April 1987. Cited on pp. 12 and 137.

[CB05] Siobhán Clarke and Elisa Baniassad. *Aspect-oriented analysis and design: the theme approach.* Addison Wesley Professional, October 2005. Cited on pp. 36 and 38.

[Chr97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of the European Conference on Object-oriented Programing, ECOOP'97*, 1997. Cited on p. 19.

[CTR05] Mariano Ceccato, Paolo Tonella, and Filippo Ricca. Is AOP Code Easier to Test than OOP Code? In *Workshop on Testing Aspect-Oriented Programs, International Conference on Aspect-Oriented Software Development*, Chicago, Illinois, March 2005. Cited on pp. 73, 74, and 116.

[CVL05] Trung Chi Ngo Cristina Videira Lopes. Unit Testing Aspectual Behavior. In *Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs, AOSD'05*, March 2005. Cited on p. 75.

[CW01] Siobhán Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proc. 23rd int'l conf. software engineering (icse)*, pages 5–14, May 2001. Cited on p. 39.

[Dav72] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. Cited on pp. 13 and 17.

[DBG+11] Romain Delamare, Benoit Baudry, Sudipto Ghosh, Shashank Gupta, and Yves Le Traon. An approach for testing pointcut descriptors in aspectj. *Software Testing, Verification and Reliability*, 21(3):215–239, 2011. Cited on p. 74.

[DFS02] Rémi Douence, P. Fradet, and M. Südholt. Detection and resolution of aspect interactions. Technical Report RR-4435, INRIA, April 2002. Cited on p. 71.

[DFS04] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on aspect-oriented software development (aosd)*, pages 141–150, 2004. Cited on p. 71.

[Ecl10] Eclipse Foundation. The Eclipse Foundation open source community website, December 2010. http://www.eclipse.org/. Cited on pp. 43 and 97.

[Eds68] Edsger W. Dijkstra. Letters to the Editor: Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, March 1968. Cited on p. 12.

[Eds76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 28 October 1976. Cited on p. 13.

[Eds82] Edsger W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982. Cited on p. 13.

[EL79] Edward Yourdon and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 17 February 1979. Cited on p. 13.

[Fai04] George Fairbanks. Hyper/j. Slides for the course Objects and Aspects: Language Support for Extensible and Evolvable Software, November 2004. Cited on p. 34.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. Cited on p. 42.

[FNRM10] Fabiano Cutigi Ferrari, Elisa Yumi Nakagawa, Awais Rashid, and José Carlos Maldonado. Automating the mutation testing of aspect-oriented java programs. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 51–58, New York, NY, USA, 2010. ACM. Cited on p. 74.

[Fow07] Martin Fowler. Mocks arent stubs. http://martinfowler.com/articles/mocksArentStubs.html, 2007. Cited on p. 50.

[FR04] Kathleen Fisher and John Reppy. A typed calculus of traits. In *Proceedings of the 11th Workshop on Foundations of Object-oriented Programming*, 2004. Cited on p. 16.

[GGB+07] Phil Greenwood, Alessandro F Garcia, Thiago Bartolomei, Sergio Soares, Paulo Borba, and Awais Rashid. On the design of an end-to-end aosd testbed for software stability. In *Proceedings of the 1st International Workshop on Assessment of Aspect-Oriented Technologies (ASAT. 07), Vancouver, Canada*. Citeseer, 2007. Cited on p. 107.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994. Cited on p. 39.

[GJA+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-

oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of 11th Europeen Conference on Object-oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997. Cited on p. 23.

[Gra06] Kasper Bilsted Graversen. *The nature of roles - A taxonomic analysis of roles as a language construct*. PhD thesis, IT University of Copenhagen, Denmark, 2006. Cited on p. 19.

[Gru99] John Grundy. Aspect-oriented requirements engineering for component-based software systems. In *4th ieee international symposium on requirements engineering*, pages 84–91. IEEE Computer Society, 1999. Cited on p. 35.

[GSF+05] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *Aosd 05*, pages 3–14, 2005. Cited on pp. 39 and 40.

[GSS+06] William G. Griswold, Macneil Shonle, Kevin Sullivan, Yuanyuan Song, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE - Software*, 23(1):51–60, January/February 2006. Cited on p. 72.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th acm conference on object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002. Cited on p. 39.

[HNBA07] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *Aosd '07: proceedings of the 6th international conference on aspect-oriented software development*, pages 85–95, New York, NY, USA, 2007. ACM Press. Cited on p. 71.

[HPJP00] Wai-Ming Ho, Francois Pennaneac'h, Jean-Marc Jézéquel, and Noël Plouzeau. Aspect-oriented design with the UML. In Peri Tarr, Anthony Finkelstein, William Harrison, Bashar Nuseibeh, Harold Ossher, and Dewayne Perry, editors, *Workshop on multi-dimensional*

separation of concerns in software engineering (icse 2000), 2000. Cited on p. 38.

[HU01] Stefan Hanenberg and Rainer Unland. Using and reusing aspects in AspectJ. In Kris De Volder, Maurice Glandrup, Siobhán Clarke, and Robert Filman, editors, *Workshop on advanced separation of concerns in object-oriented systems (oopsla 2001)*, 2001. Cited on p. 41.

[JDT10] JDT. Eclipse Java development tools (JDT), December 2010. http://www.eclipse.org/jdt/. Cited on p. 97.

[Kat04] Shmuel Katz. Diagnosis of harmful aspects using regression verification, 2004. Cited on pp. 67 and 70.

[KC03] Mohamed Kande and Valentin Crettaz. Towards patterns for concern-oriented software architecture. In Omar Aldawud, Mohamed Kandé, Grady Booch, Bill Harrison, Dominik Stein, Jeff Gray, Siobhán Clarke, Aida Zakaria Santeon, Peri Tarr, and Faisal Akkawi, editors, *Workshop on aspect-oriented modeling with UML (AOSD-2003)*, 2003. Cited on p. 39.

[KI89] Karl J. Lieberherr and Ian Holland. Assuring Good Style for Object-oriented Programs. *IEEE - Software*, 6(5):38–48, September 1989. Cited on p. 14.

[KIA88] Karl J. Lieberherr, Ian Holland, and Arthur J. Riel. Object-oriented Programming: an Objective Sense of Style. In Norman K. Meyrowitz, editor, *Proceedings of ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 323–334, September 1988. Cited on p. 14.

[KKS02] Mohamed Mancona Kandé, Jörg Kienzle, and Alfred Strohmeier. From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach. In *the Second International Workshop on Aspect-Oriented Modeling with UML, in conjunction with the Fifth International Conference on the Unified Modeling Language - the Language and its Applications (UML2002), September 30 - October 4, 2002, Dresden, Germany*, 2002. Also available as Technical Report IC/2002/58, Ecole Polytechnique Fédérale de Lausanne

(EPFL), School of Computer and Communication Sciences. Cited on p. 38.

[Koo95] Piet Koopmans. *On the Definition and Implementation of the Sina/st Language*. PhD thesis, University of Twente, 1995. Cited on p. 33.

[KT06] Benoit Kessler and Éric Tanter. Analyzing interactions of structural aspects. In *Proceedings of the ECOOP Workshop on Aspects, Dependencies and Interactions (ADI)*, 2006. Cited on p. 71.

[KYX03] Jörg Kienzle, Yang Yu, and Jie Xiong. On composition and reuse of aspects. In *Software engineering properties of languages for aspect technologies*, 2003. Cited on pp. 67 and 70.

[LF14] Thiago Gaspar Levin and Fabiano Cutigi Ferrari. Is it difficult to test aspect-oriented software? preliminary empirical evidence based on functional tests. In *Proceedings of the 11th Workshop on Software Modularity*, 2014. Cited on p. 74.

[LJDW04] Bert Lagaisse, Wouter Joosen, and Bart De Win. Managing semantic interference with aspect integration contracts. In *Software engineering properties of languages and aspect technologies*, 2004. Cited on p. 69.

[LSS03] David Larochelle, Karl Scheidt, and Kevin Sullivan. Join point encapsulation. In *Software engineering properties of languages and aspect technologies*, 2003. Cited on p. 72.

[Mar98] Robert C. Martin. *Java Gems*. Cambridge University Press, New York, NY, USA, 1998. Cited on p. 15.

[MBB08] Freddy Munoz, Benoit Baudry, and Olivier Barais. A classification of invasive patterns in AOP. Research Report RR-6501, INRIA, 2008. Cited on p. 68.

[MC04] Maximilian Störzer and Christian Koppen. PcDiff: attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004. Cited on p. 116.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *IEEE - Computer*, 25(10):40–51, 25 October 1992. Cited on p. 69.

[MF05a] Miguel Pessoa Monteiro and João Miguel Fernandes. The search for aspect-oriented refactorings must go on. In Tom Tourwé, Andy Kellens, Mariano Ceccato, and David Shepherd, editors, *Linking aspect technology and evolution*, 2005. Cited on p. 42.

[MF05b] Miguel Pessoa Monteiro and João Miguel Fernandes. Towards a catalog of aspect-oriented refactorings. In Peri Tarr, editor, *Proc. 4rd int' conf. on aspect-oriented software development (AOSD-2005)*, pages 111–122, 2005. Cited on pp. 20 and 42.

[MMvD07] Leon Moonen Marius Marin and Arie van Deursen. An integrated crosscutting concern migration strategy and its application to jhotdraw. Technical report, Delft University of Technology Software Engineering Research Group, 2007. Cited on p. 107.

[Mon05] Miguel Pessoa Monteiro. *Refactorings to evolve object-oriented systems with aspect-oriented concepts*. PhD thesis, Departamento de Informática, Universidade do Minho, Portugal, 2005. Cited on p. 42.

[Mor05] Michael Mortensen. An approach for adequate testing of aspectj programs. In *In 2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD*, 2005. Cited on p. 73.

[MR02] Katharina Mehner and Awais Rashid. Towards a standard interface for runtime inspection in AOP environments. In Mark C. Chu-Carroll, Gail C. Murphy, Siobhan Clarke, Jacky Estublier, Anthony Finkelstein, Bill Harrison, and Elissa Newman, editors, *Workshop on advanced separation of concerns in object-oriented systems (oopsla 2002)*, 2002. Cited on p. 42.

[MT02] Maja D'Hondt and Theo D'Hondt. The Tyranny of the Dominant Model Decomposition. In *OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Seattle, Washington, 2002. Cited on pp. 17 and 18.

[Mye79] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. Cited on pp. 47, 49, and 55.

[PDF+02] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. A UML notation for aspect-oriented software design. In Omar Aldawud, Grady Booch, Siobhán Clarke, Tzilla Elrad, Bill Harrison, Mohamed Kandi, and Alfred Strohmeier, editors, *Workshop on aspect-oriented modeling with UML (AOSD-2002)*, 2002. Cited on p. 38.

[PGAA09] Reza Meimandi Parizi, Abdul Azim Abdul Ghani, Rusli Abdullah, and Rodziah Atan. On the applicability of random testing for aspect-oriented programs. *International Journal of Software Engineering and its Applications*, 3(4):1–20, 2009. Cited on p. 74.

[PGL15] Reza Meimandi Parizi, Abdul Azim Abdul Ghani, and Sai Peck Lee. Automated test generation technique for aspectual features in aspectj. *Information and Software Technology*, 57:463 – 493, 2015. Cited on p. 74.

[PHWSMS99] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st international conference on Software engineering, ICSE'99*, 1999. Cited on p. 34.

[PTP07] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*, pages 535–552, Montreal, Canada, October 2007. Cited on p. 42.

[RA07] André Restivo and Ademar Aguiar. Towards detecting and solving aspect conflicts and interferences using unit tests. In *Proceedings of the 5th Workshop on Software-Engineering Properties of Languages and Aspect Technologies (SPLAT'07)*, pages 1–5, Vancouver BC, Canada, 2007. Cited on p. 98.

[RA08] André Restivo and Ademar Aguiar. Disciplined composition of aspects using tests. In *Proceedings of the 2008 AOSD Workshop on*

Linking Aspect Technology and Evolution, LATE '08, pages 8:1–8:5, New York, NY, USA, 2008. ACM. Cited on p. 98.

[RA09a] André Restivo and Ademar Aguiar. Testing for unexpected interactions in AOP. In *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*, pages 548–552. IEEE, 2009. Cited on p. 66.

[RA09b] André Restivo and Ademar Aguiar. DrUID unexpected interactions detection. Demonstration at the Aspect Oriented Software Development Conference (AOSD'09), 2009. Cited on p. 101.

[RD00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented Programming is Quantification and Obliviousness. In *OOPSLA Workshop on Advanced Separation of Concerns*, 2000. Cited on pp. 24 and 65.

[Res09] André Restivo. DrUID: Unexpected interactions detection, 2009. Cited on p. 99.

[Res10] André Restivo. Aida: Automatic interference detection for aspectj, 2010. Cited on p. 101.

[Res14] André Restivo. School-aspectj-testbed, 2014. Cited on p. 108.

[RMA03] Awais Rashid, Ana Moreira, and João Araújo. Modularization and composition of aspectual requirements. In *Proc. 2nd int' conf. on aspect-oriented software development (AOSD 2003)*, 2003. Cited on p. 36.

[RMA+10] Awais Rashid, Ana Moreira, João Araújo, Paul Clements, Elisa Baniassad, and Bedir Tekinerdogan. Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, 2010. http://www.early-aspects.net/. Cited on p. 36.

[Rob96a] Robert C. Martin. The Dependency Inversion Principle. *Engineering Notebook, The C++ Report*, 8, May 1996. Cited on p. 13.

[Rob96b] Robert C. Martin. The Interface Segregation Principle. *Engineering Notebook, The C++ Report*, June 1996. Cited on p. 14.

[Rob97] Robert C. Martin. Stability. *Engineering Notebook, The C++ Report*, February 1997. Cited on p. 14.

[Rob00] Robert C. Martin. Design Principles and Design Patterns. Technical report, Object Mentor, 2000. Cited on p. 13.

[Rob01] Robert E. Filman. What is Aspect-oriented Programming, Revisited. Technical report, RIACS, 2001. Cited on pp. 24, 65, and 157.

[ROM01] Rémi Douence, Olivier Motelet, and Mario Südholt. A Formal Definition of Crosscuts. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns*, 2001. Cited on p. 35.

[Sch09] Fabian Schmied. What can we do for you? (features of re-motion mixins), January 2009. Cited on p. 16.

[SHU02] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Designing aspect-oriented crosscutting in UML. In Omar Aldawud, Grady Booch, Siobhán Clarke, Tzilla Elrad, Bill Harrison, Mohamed Kandi, and Alfred Strohmeier, editors, *Workshop on aspect-oriented modeling with UML (AOSD-2002)*, 2002. Cited on p. 38.

[SK03] Maximilian Störzer and Jens Krinke. Interference analysis for AspectJ. In *Foundations of aspect-oriented languages (foal)*, 2003. Cited on p. 70.

[SMSR02] Jr. Stanley M. Sutton and Isabelle Rouvellou. Modeling of software concerns in cosmos. In *Aosd '02: proceedings of the 1st international conference on aspect-oriented software development*, pages 127–133, New York, NY, USA, 2002. ACM Press. Cited on p. 36.

[SPB06] Stephanie Balzer, Patrick T. Eugster, and Bertrand Meyer. Can Aspects Implement Contracts? In *Proceedings of Rapid Implementation of Engineering Techniques (RISE)*, 2006. Cited on p. 14.

[SY99] Junichi Suzuki and Yoshikazu Yamamoto. Extending UML with aspects: aspect support in the design phase. In *ECOOP workshops*, pages 299–300, 1999. Cited on p. 37.

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002. Cited on p. 48.

[TBB04] Francis Tessier, Mourad Badri, and Linda Badri. A model-based detection of conflicts between crosscutting concerns: towards a formal approach. In *International workshop on aspect-oriented software development*, 2004. Cited on p. 66.

[THP93] Walter F Tichy, Nico Habermann, and Lutz Prechelt. Summary of the dagstuhl workshop on future directions in software engineering: February 17–21, 1992, schloß dagstuhl. *ACM SIGSOFT Software Engineering Notes*, 18(1):35–48, 1993. Cited on p. 4.

[vdBCC05] Klaas van den Berg, José María Conejero, and Ruzanna Chitchyan. AOSD Ontology 1.0 - Public Ontology of Aspect-orientation. Technical Report AOSD-Europe-UT-01 D9, AOSD-Europe, Enschede, May 2005. Cited on pp. 17 and 155.

[vDMM05] A. van Deursen, M. Marin, and L. Moonen. A Systematic Aspect-Oriented Refactoring and Testing Strategy, and its Application to JHotDraw. Technical report, Centrum voor Wiskunde en Informatica (CWI), March 2005. Cited on p. 116.

[WG10] Fadi Wedyan and Sudipto Ghosh. A dataflow testing approach for aspect-oriented programs. In *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, pages 64–73. IEEE, 2010. Cited on p. 74.

[WH93] William Harrison and Harold Ossher. Subject-Oriented Programming - A Critique of Pure Objects. In *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPLSA'93*, September 1993. Cited on p. 20.

[Whi87] Lee J. White. Software testing and verification. *Advances in Computers*, 26(1):335–390, 1987. Cited on pp. 48 and 49.

[WZ12] Peng Wanga and Xiaochun Zhao. The research of automated select test cases for aspect-oriented software. In *Proceedings of the 2nd*

International Conference on Mechanical, Industrial, and Manufacturing Engineering, MIME'2012, 2012. Cited on p. 73.

[XEAXW12] Dianxiang Xu, Omar El-Ariss, Weifeng Xu, and Linzhang Wang. Testing aspect-oriented programs with finite state machines. *Softw. Test. Verif. Reliab.*, 22(4):267–293, June 2012. Cited on p. 74.

[XX06] Dianxiang Xu and Weifeng Xu. State-based Incremental Testing of Aspect-oriented programs. In *Proceedings of the 5th international conference on aspect-oriented software development, AOSD '06*, pages 180–189, New York, NY, USA, 2006. ACM. Cited on p. 74.

[YSM+05] Yudai Yamazaki, Kouhei Sakurai, Saeko Matsuura, Hidehiko Masuhara, Hiroaki Hashiura, and Seiichi Komiya. A unit testing framework for aspects without weaving. In *Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs, AOSD'05*, March 2005. Cited on p. 75.

[ZA06] Haibin Zhu and Rob Alkins. Towards role-based programming. In *Proceedings of the Workshop on Role-Based Collaboration, CSCW'06*, 2006. Cited on p. 19.

[Zha02] Jianjun Zhao. Tool support for unit testing of aspect-oriented software. In Mark C. Chu-Carroll, Gail C. Murphy, Siobhan Clarke, Jacky Estublier, Anthony Finkelstein, Bill Harrison, and Elissa Newman, editors, *Workshop on advanced separation of concerns in object-oriented systems (oopsla 2001)*, 2002. Cited on p. 43.

[Zha03] Jianjun Zhao. Unit testing for aspect-oriented programs. Technical Report SE-141-6, Information Processing Society of Japan (IPSJ), May 2003. Cited on p. 43.

[ZW98] Marvin V. Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, May 1998. Cited on pp. 4 and 5.

[ZZR04] Yuewei Zhou, Hadar Ziv, and Debra J. Richardson. Towards A Practical Approach to Test Aspect-Oriented Software. In *Proceed-*

ings of Testing of Component-Based Systems and Software Quality, 2004, January 2004. Cited on p. 73.

Glossary

advice A certain function, method or procedure that is to be applied at a given join point of a program. 24–32, 38, 40, 41, 43, 59, 62, 68, 72, 73, 103, 109, 127

aspect A unit for modularising an otherwise crosscutting concern. 6, 21, 25, 26, 28–30, 32, 35, 37–45, 58–60, 62, 63, 66–75, 77–80, 88, 91, 94, 98, 100, 101, 117–119, 121, 123, 124, 126–131, 135, 138

aspect oriented programming A programming paradigm that allows the separation of crosscutting concerns. 5, 6, 14, 21, 72–74, 91, 94, 95, 134, 139, 140, 159

AspectJ An AOP language based on Java. 6, 21, 25, 26, 28, 30–32, 35, 42, 43, 55, 56, 58, 60, 62, 65, 74, 84, 97, 107, 125, 135, 140

AspectJ Development Tools A project that provides Eclipse platform based tool support for AOSD with AspectJ. 159

composition The integration of multiple modular artefacts into a coherent whole. 17

composition filters A language-independent specification, based on filters, on top of the classical OOP model, that provides a mechanism to separate crosscutting concerns from the primary concerns. 159

concern An interest, which pertains to the system's development, its operation or any other matters that are critical or otherwise important to one or more stakeholders [vdBCC05]. 2, 13, 14, 17–20, 25, 32, 44, 45, 55, 58, 60–63

crosscutting concern A concern, which cannot be modularly represented within the selected decomposition Consequently, the elements of crosscutting concerns are scattered and tangled within elements of other concerns. 2, 18, 21, 23, 31, 34, 35, 56, 75, 137

decomposition The breaking down of a larger problem into a set of smaller problems which may be tackled individually. 12, 17, 18

directed acyclic graph A directed graph where there is no way to

start at some vertex v and follow a sequence of edges that eventually loops back to v again. 92, 159

encapsulation The ability of an object to hide its internal representation. 12, 15

event based AOP A framework for AOP where aspects are defined in terms of events emitted during program execution and crosscuts relate sequences of events. 159

fault model A model that identifies relationships and components of a system under test that are most likely to have faults. 115

feature oriented programming A programming paradigm where features are raised to first-class entities and are seen as small increments in program development or functionality. 159

hypermodule A set of hyperslices. 34

hyperslice A set of conventional modules that contain units that pertain to a single concern. 34, 35

inheritance A mechanism for code reuse where classes are based on other classes. 12, 15–17

inter-type declaration Declaration of members (fields, methods, and constructors), class extensions and interface implementations on other types. 58

invasive aspect An aspect that modifies the external behavior of the advised module. 63, 68, 78–80, 82, 83, 91, 94, 111, 114, 138

Java A general-purpose computer programming language designed to produce programs that will run on any computer system. 21, 25, 56–58, 97

Java Development Tools A project that provides the tool plugins that implement a Java IDE supporting the development of any Java application, including Eclipse plugins. 159

join point A point in the control flow of a program. 24–32, 35, 38, 44, 59, 62, 66, 72, 79, 116, 118, 119, 124, 126, 127

join point model The way in which join points are quantified and advices are woven in a AOP language. 24–26

modularity The degree to which a program is decomposed so that each module has its own clear set of responsibilities. 1–3, 6, 11, 12, 14, 15, 17, 18, 21, 23, 58, 95

module A self-contained part of a program having its own logic and a clear set of responsibilities. 2, 3, 13, 14, 18, 31, 34, 37, 40, 42,

object oriented programming A programming paradigm based on the concept of objects, which are data structures that contain data and code. 31, 58, 73, 74, 159

obliviousness The idea that the places quantifications are applied to, did not have to be specifically prepared to receive these enhancements [Rob01]. 24, 31

pointcut A program element that picks out join points and exposes data from their execution context. 25–28, 31, 32, 35, 38–42, 44, 69, 72–74, 116, 118, 138

polymorphism The ability of a type to appear to be another type. 12, 15

procedural oriented programming A programming paradigm based upon the concept of procedure calls. 159

programming paradigm A fundamental style of computer programming. 12, 17

quantification The idea that one can write unitary and separate statements that have effect in many, non-local places in a programming system [Rob01]. 24, 31

reusability The degree to which a module is reusable in other programs as a standalone artefact without modifications. 15

role oriented programming A programming paradigm where objects are seen as the composition of different roles. 159

scattering The occurrence of elements that pertain to one concern in several different modules. 17, 18, 25, 94

separation of concerns A design principle for separating a program into distinct units of modularity, each one addressing a different concern. 13, 18, 19, 24, 32, 45, 52, 78, 159

singleton A design pattern that restricts the instantiation of a class to one object. 28

software product line A collection of similar software systems created from a shared set of software assets. 140, 159

subject oriented programming A programming paradigm where behaviors of objects are provided by the various other *subjects* of the objects which are beyond the scope and control of the author of the original object. 159

tangling The occurrence of elements pertaining to different concerns in the same module. 17, 18, 25, 32, 63, 94

unit test An automated piece of code

that checks a single assumption about the behavior of a single unit of work. 56–58, 78, 84, 94, 138

unit testing The use of unit tests to verify the correctness of programs. 6, 49, 55, 95

Acronyms

AJDT AspectJ Development Tools. 97, 102

AOP Aspect Oriented Programming. 2–4, 6, 20, 21, 23–25, 31–33, 35, 37–40, 42, 43, 45, 52, 55, 65, 69–72, 74, 75, 77–79, 88, 91, 97, 107, 116, 121, 136–138, 140

CF Composition Filters. 33, 34

DAG Directed Acyclic Graph. 78, 81, 92

EAOP Event Based AOP. 35

FOP Feature Oriented Programming. 19, 20

GoF The authors of the most influential book on Design Patterns: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. 39

JDT Java Development Tools. 97, 102

OOP Object Oriented Programming. 2, 12, 15–17, 20, 24, 25, 28, 34, 42, 52, 69, 74, 78, 79, 88, 91

POP Procedural Oriented Programming. 12, 24

ROP Role Oriented Programming. 19

SoC Separation of Concerns. 13, 14, 16, 17, 21

SOP Subject Oriented Programming. 20

SPL Software Product Line. 19, 140

A

Published Articles

Towards Detecting and Solving Aspect Conflicts and Interferences Using Unit Tests

André Restivo
Faculdade de Engenharia da Universidade do Porto
arestivo@fe.up.pt

Ademar Aguiar
Faculdade de Engenharia da Universidade do Porto,
INESC Porto
aaguiar@fe.up.pt

Abstract

Aspect Oriented Programming (AOP) is a programming paradigm that aims at solving the problem of crosscutting concerns being normally scattered throughout several units of an application.

Although an important step forward in the search for modularity, by breaking the notion of encapsulation introduced by Object Oriented Programming (OOP), AOP has proven to be prone to numerous problems caused by conflicts and interferences between aspects.

This paper presents work that explores the proven unit testing techniques as a mean to help developers describe the behavior of their aspects and to advise them about possible conflicts and interferences.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Design, Languages, Verification

Keywords AOP, Conflicts, Interferences, Unit Testing

1. Introduction

Separation of concerns (SoC) has always been the main goal of software engineering. It refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose [1].

Aspect-Oriented Programming(AOP) [2] is a new programming paradigm that builds on the success of proved paradigms, like Object-Oriented Programming(OOP). The main idea behind AOP is that concerns crosscutting several modules of an application can be developed as single units of modularity and weaved into the application, through a process of composition, in specific points called joinpoints. The motivation behind AOP is to improve the overall modularity of an application by enabling the ability to develop crosscutting concerns as separate units.

AOP also aims for obliviousness [3], i.e. a developer should not have to know about any other aspects that are being weaved into the application code. To make this a reality, aspects have to be allowed to attach virtually to any position of the original source code. In this way, each developer only has to be concerned about his own modules.

Workshop SPLAT '07 March 12-13, 2007 Vancouver, British Columbia, Canada

However, *obliviousness* has been harder to achieve than expected, as conflicts between aspects are prone to occur in large applications with an heavy usage of AOP. In the OOP world conflicts are avoided by using encapsulation techniques and also by using Unit Testing or the Design by Contract (DbC) [4] approaches. Encapsulation allowed developers to know that changing the inner working of an unit would not break another part of the application as long as their public interfaces would be kept the same. Unit testing and the DbC approaches helped developers detect problems caused by an unit changing its public behavior in an easier way (i.e. Regression Testing).

As most AOP languages allow to weave aspects into the private methods of an unit, thus changing their inner workings, encapsulation is no longer a guarantee. However, this is an important feature of the AOP paradigm that allows crosscutting concerns, such as *logging* and *security*, to be easily added by means of a separate modular unit. The problem is that conflicts will certainly arise due to the new possibilities this brings.

Besides the loss of encapsulation, unit tests and contracts suddenly became harder to use mainly because aspects can change the public behavior of an unit thus making obsolete some unit tests and contracts attached to that unit. Unit testing and the DbC approaches have therefore to be rethought in order to accommodate this new way of programming.

In this position paper we will argue how the unit testing approach can still be effectively used in AOP, retaining the same characteristics that made it a very popular Regression Testing methodology, and also be helpful in detecting conflicts between aspects.

Our final objective is to create a methodology that will improve the current state of managing conflicts in AOP and to develop tools to support that methodology.

The paper is organized in the following sections: Section 1, this section, introduced the problem of conflicts and interferences as an important issue in AOP; Section 2 will describe some proposed categorization of both aspects and conflicts; Section 3 will explain how these conflicts could be detected using unit tests; Section 4 will show how the different types of conflicts described in Section 2 can be identified by using the methodology explained in Section 3; Section 6 will introduce a small example that will help understanding how the presented methodology could be used in a real situation; Section 7 will describe several important works in the conflict detection field; finally Section 8 will list some conclusions and pointers for future work.

2. Conflicts and Interferences

Before tackling the problems posed by the introduction of aspects into an application, we have to understand the different type of changes they can perform and the objectives behind these changes.

Several attempts of categorizing aspects have been done in recent literature.

2.1 Types of Interferences

Tessier [5] classified aspects by the different type of interferences they can cause. In his work, he identified problems like:

- the use of wildcards leading to accidental joinpoints;
- conflicts between aspects and the importance of the order in which these are weaved into the application;
- circular dependencies between aspects;
- conflicts between concerns where a concern needs to change a functionality needed by another concern.

2.2 Types of Changes

Katz [6] took a different approach by classifying aspects according to the type of changes they introduce in an application. According to this author three types of aspects can be identified:

- *spectative aspects*, that only gather information about the system to which they are woven, usually by adding fields and methods, but do not influence the possible underlying computations;
- *regulatory aspects*, that change the flow of control (e.g., which methods are activated in which conditions) but do not change the computation done to existing fields;
- *invasive aspects*, that change values of existing fields (but still should not invalidate desirable properties).

2.3 Types of Dependencies

Kienzle [7] approached the problem from a different point of view by considering only the dependency relationships between aspects and the original code. Three different kinds of aspect dependencies have been identified:

- *orthogonal aspects*, that provide functionality to an application that is completely independent from the other functionalities to the application;
- *uni-directional aspects*, that depend from some functionality of the application (these can be further divided as preserving, if the application functionality is maintained or enhanced without any current functionalities being altered or hidden, or modifying, if the application functionality is altered or hidden);
- *circular aspects*, which are aspects that are mutually dependent of each other.

Katz and Tessier works are extremely interesting as a starting point for our research. We intend to analyze how the different type of changes introduced by aspects (as defined by Katz) can create different types of interferences (as defined by Tessier) and how these can be tackled using unit testing. The following section explains our approach to this problem.

3. Detecting Conflicts Using Unit Tests

Unit testing is used to informally proof the correctness of modules. Each module has its own set of unit tests. By running these tests one can verify if changes to a module have changed its tested external behavior. Unit tests can be seen as a specification of the desired behavior of a module. With the introduction of aspects these specifications and their respective implementations can be easily changed by external entities, so units may no longer behave as expected. When an aspect is weaved into the code of an application, other aspects might have been weaved before and changed the expected behavior of the affected unit in a way that interferes with the new aspect being weaved.

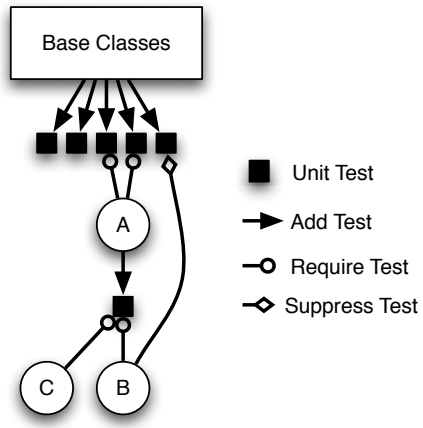

Figure 1. Aspects and Unit Tests

In this way, it should be possible to specify which unit tests have to be valid for an aspect to be correctly weaved into the system. In the same way, it should be possible for an aspect to determine which tests it expects to break.

Many times aspects depend on each other. This happens when one aspect needs some behavior to be present in the system to work properly and this behavior is introduced by another aspect. It should also be possible for aspects to introduce new unit tests into the system specifying which new behaviors are being introduced by them.

It might also happen that an aspect needs a certain behavior to be present in the system but the unit providing this behavior does not have a specific unit test for this particular behavior. Aspects should be able to add new unit tests to code already in the application.

Figure 1 shows a possible diagram for an example where aspects add, remove and depend from unit tests. In this figure the dark square boxes are unit tests. The arrowed lines identify which unit or aspect created the unit test. The circled lines identify a dependency relationship, while lines with a diamond represent an invalidation of an unit test by an aspect (the big circles). From this explanation we can see that the initial OOP code already provided several unit tests. Aspect ”A” depends on two of those unit tests and adds another one. Aspect ”B” depends on the unit test created by Aspect ”A” and at the same time suppresses one of the initial unit tests. And finally, Aspect ”C” also depends on the unit test created by Aspect A.

From this simple example we can already extract some conclusions: Aspect ”A” is probably a *spectative* aspect (as defined by Katz) that simply added some new fields and methods to the unit; Aspect ”B”, on the other hand, has probably changed the behavior of the original code. We can also easily conjecture a possible order for the weaving process (e.g. ”A” followed by ”B” followed by ”C”).

Finding a possible weaving order in which dependencies between aspects are assured can probably be accomplished by using a simple Breadth-First Search (BFS) or using the A* algorithm (as this is a typical path finding in a graph problem) . If such an ordering cannot be found then we are facing a conflict between aspects. In this case, an error message should be presented stating which aspects failed to weave, which unit tests are missing for these aspects, and which aspects removed them (if any).

The next section will explain how this approach relates to Tessier and Katz classification of aspect interferences and conflicts.

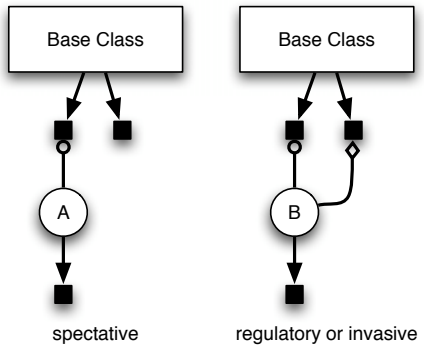

Figure 2. Different types of aspects

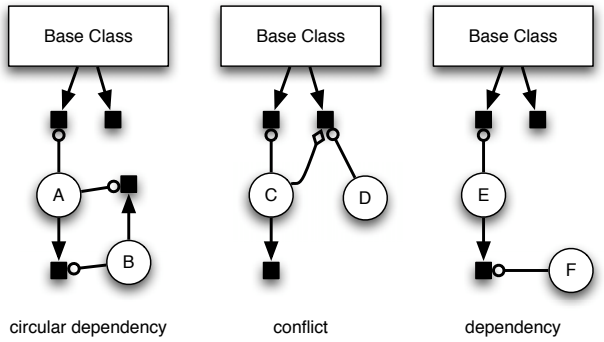

Figure 3. Different types of interferences

4. Mapping aspects and interferences to unit tests

As we have seen in Section 2, aspects can be classified as being *spectative*, *regulatory* or *invasive*. Using the notation introduced in Section 3 we can depict these different type of aspects with relation to the unit tests they add, depend on, or suppress.

In Figure 2 there are two different aspects. Aspect ”A” is probably a *spectative* aspect as it doesn't suppress any existing unit tests. It could also be an *invasive* aspect that happened to be ”lucky” enough to change something the original developer wasn't expecting to be changed and didn't include in his unit tests. Aspect ”B”, on the other hand, is clearly a *regulatory* or *invasive* aspect as it suppresses some of the original unit tests that would fail after it had been weaved into the system.

In Figure 3, three types of interferences or conflicts are depicted. In the first one, aspects ”A” and ”B” are creating a circular dependency problem. The middle diagram depicts a conflict between two concerns, where aspect ”C” is changing some functionality needed by aspect ”D”. The rightmost diagram shows aspects that need to be weaved in the correct order to function properly and at the same time a dependency between aspect ”F” and ”E”.

This shows that if unit tests are correctly used they can help detecting most of the conflicts that aspects can introduce and that have been plaguing AOP. In the following sections we will show how these conflicts can be tackled with our proposed methodology with the help of a short example a short example.

5. Using annotations to specify changes to unit tests

As has been stated before, breaking an unit test is not a clear sign of an aspect misbehaving. Due to their own nature, aspects are bound to change the functionality of other units of code and hence break their unit tests. In this way, aspects must have a way of announcing what unit tests they expect to break.

Very often, aspects are also depending on some functionality to be present into the system. This functionality can be delivered by the system base code or by other aspects. Conflicts between aspects are often caused by one aspect removing a functionality needed by another aspect, and dependency problems are commonly caused by one aspect expecting another aspect to deliver some functionality which somehow is not effectively delivered.

Therefore we claim that, there is a clear need for aspects to be able to announce which aspects they are expected to break, which aspects they depend on, and which they are adding to the overall system. In this paper we propose that aspects should be able to make this announcements using Java annotations. An example will now be introduced to explain how this could be attainable.

6. An example of conflicting aspects

Imagine a simple class depicting an User. This class would have fields like its username and password. It would also have setters and getters for those fields and a *verifyPassword* method. Listing 1 shows some simple unit tests that could have been used to ensure that the class was working properly.

Listing 1. User Class Unit Tests

```
public void testSetGetPassword() {
  user.setPassword("foo");
  assertEquals("foo", user.getPassword());
}

public void testVerifyPassword() {
  user.setPassword("foo");
  assertEquals(true, user.verifyPassword("foo"));
  assertEquals(false, user.verifyPassword("bar"));
}
```

It is also common that, for security reasons, passwords do not get stored in clear text. It is a common practice to store them using some hash function. However, to achieve a clear separation of concerns between the user data model and the security concern, this feature should be coded as a separate aspect. Listing 2 shows how this aspect could have been coded. By introducing these aspects some of the unit tests shown in Listing 1 get broken.

Listing 2. Encrypted Password Aspect

```
@SupressTest("user.UserTest.testSetGetPassword")
privileged aspect EncryptedPassword {
  protected pointcut
    passwordChanged(User user, String password):
    target(user) && args(password)
    && call(void setPassword(String));

  protected pointcut
    verifyPassword(User user, String password):
    target(user) && args(password)
      && call(boolean verifyPassword(String));

  void around(User user, String password)
    : passwordChanged(user, password)
  {
    // ... calculates md5 hash
    user.password =  md5hash;
  }
```

```
  boolean around(User user, String password)
    : verifyPassword(user, password)
  {
    // ... calculates md5 hash
    return (user.password.equals(md5hash);
  }
}
```

After introducing the aspect into the system, the developer should be warned that his aspect broke some unit tests. This could be easily computed by compiling and testing the system with and without the aspect. The developer could then inspect the broken unit test and decide if that would be an expected result from his aspect. In this case he would decide that it was because the getter and setter methods of the User class would not work as expected so he could just add a notation expressing that. The first line of Listing 2 shows how that notation could look like.

It is also common to prevent users from using passwords that are easily retrievable using brute force attacks. One way of doing it is to prevent them from using passwords that are too small. Once again, preventing this should be considered a separate aspect from the user data model and could be coded as seen in Listing 3. Notice that this aspect could have been coded in a much better fashion but for demonstration purposes it has been coded in a way that it needed the getter and setter methods of the original user class to work as originally intended. The developer should then announce that this aspect depends on the *testSetGetPassword* unit test. He could easily do so by adding a single line stating that in the beginning of the aspect.

Listing 3. Minimum Password Size Aspect

```
@RequiresTest("user.UserTest.testSetGetPassword")
@SupressesTest("user.UserTest.testVerifyPassword")
@AddsTest("user.UserTest.testVerifyPasswordML")
public aspect MinimumLengthPassword {
  protected pointcut
    setPassword(User user, String password)
    : target(user) && args(password)
      && call(void setPassword(String))
      && !within(MinimumLengthPassword);

  after(User user, String password)
  : setPassword(user, password)
  {
    if (user.getPassword().length()<6) {
      user.setPassword(password);
      throw new RuntimeException();
    }
  }
}
```

However, after introducing the aspect into the system, the developer would be warned that another aspect has suppressed that unit test. Besides that, this aspect would break the *testVerifyPassword* unit test. This is a typical case of a conflict between aspects. To solve this problem the aspect has to be rewritten in a different way and the broken unit test must be suppressed and, perhaps, a new unit test should be added to verify if everything is still working.

This example shows how unit tests, if correctly used, can help detecting conflicts between aspects. It has also shown that the developer of each different concern did not have to know about other aspects being weaved into the system, at least until conflicts occurred thus promoting obliviousness.

In the following section some of the related work done in this field will be presented and discussed.

7. Related Work

In this section some of the work that has been done in the field of conflict detection in AOP will be described briefly. The work done so far can be divided into two different categories. Automatic detection of conflicts without human intervention and forcing aspect developers to somehow express the possible points of conflict. The first two works described fall in the first category, while the remaining three fall in the second one.

7.1 Program Slicing

Balzarotti [8] claims that this problem can be solved by using a technique proposed in the early 80's called program slicing. A slice of a program is the set of statements that affect a given point in an executable program. According to the author the following holds:

> Let A1 and A2 be two aspects and S1 and S2 the corresponding backward slices obtained by using all the statements defined in A1 and A2 as slicing criteria. A1 does not interfere with A2 if $A1 \cap S2 = \emptyset$;

According to the author, this technique is accurate enough to identify all interferences introduced by an aspect but can also detect some false-positives. Furthermore, the existence of pointcuts that are defined based on dynamic contexts, forces the analysis of every execution trace increasing the number of these false-positives. However the approach has the advantage of removing the burden of having to declare formally the expected behavior of each aspect.

7.2 Introduction and Hierarchical Changes Interferences

Störzer [9] developed a technique to detect interferences caused by two different, but related, properties of AOP languages.

Störzer claims that the possibility of aspects introducing members in other classes can lead to undesired behaviors as it can result in changes of dynamic lookup if the introduced method redefines a method of a superclass. He calls this type of interference *binding interference*.

The other problem Störzer refers to is the possibility of aspects changing the inheritance hierarchy of a set of classes. He claims that this type of changes can also give place to *binding interferences* as well as some unexpected behavior caused by the fact that *instanceof* predicates will no longer give the same results as before.

To detect this kind of conflicts the author proposes an analysis based on the lookup changes introduced by aspects.

Kessler [10] also studied how structural interferences could be detected. However, his approach is based in a logic engine where programmers can specify rules (ordering, visibility, dependencies, ...). In [10], Kessler also described the different type of interferences that are possible with introductions and hierarchical changes and proposed solutions for each one of them.

7.3 Aspect Integration Contracts

Contracts have been introduced by Meyer [4] as a defensive solution against dependency problems in OOP. Some authors claim that contracts can be imported into the AOP world in order to assist programmers in avoiding interference problems.

Lagaisse [11] proposed an extension to the Design by Contract (DbC) approach by allowing aspects to define what they expect of the system and how they will change it. This will allow the detection of interferences by other aspects that were weaved before, as well as the detection of interferences by aspects that are bounded to be weaved later in the process. According to the author, for an Aspect A bound to a component C the following should be defined:

1. The aspect should specify what it requires from component C and possibly from other software components.
2. The aspect also needs to specify in which way it affects the component C and the functionality it provides (if applicable).
3. The specification of component C must express which interference is permitted from certain (types of) aspects.

This approach has the disadvantage of forcing the programmer to verbosely specify all requirements and modifications for each aspect as well as permitted interferences. On the other hand, the formal specification of behaviors has be proven to be a valuable tool in Software Engineering. However, the major drawback we se in this approach is the necessity of components to specify which interferences are permitted thus breaking obliviousness.

7.4 Regression Testing

Katz [6] proposed the use of *regression testing* and *regression verification* as tools that could help identifying harmful aspects. The idea behind this technique is to use regression testing as normally and then weave each aspect into the system and rerun all regression tests to see if they still pass.

Katz approach is very similar to ours but does not specify the addition and removal of unit tests by aspects. Katz argues that to use his technique one will have to specify what are the desired properties of the augmented system (after the aspect being tested is added) but does not explain how this can be done.

7.5 Service Based Approach

It has been noticed by Kienzle [7] that aspects can be defined as entities that require services from a system, provide new services to that same system and removes others. If some way of explicitly describing what services are requires by each aspect it would be possible to detect interferences (for example, an aspect that removes a service needed by another aspect) and to choose better weaving orders.

8. Conclusions and Future Work

Detecting conflicts caused by the introduction of aspects is an area where much research is being carried on and much more still needs to be done. This position paper presented a methodology that uses unit tests to tackle the this problem.

Our work, although very similar to the approaches of both Lagaisse and Katz (see Sections 7.3 and 7.4), has what we think are some major and important differences: Lagaisse forces components to specify permitted interferences thus breaking obliviousness; Katz does not specify the possibility of aspects removing unit tests in order to announce what functionalities have been changed from the original code.

Therefore, we believe our approach as promising and deserves further research and exploration to firmly confirm its value.

There is still a lot of ground to cover for this methodology to be usable. We are planning to develop plugins for some important development tools and IDEs that will allow the use of annotations to help developers on describing the behavior of their aspects. This would also allow these same development tools to give feedback about possible conflicts and interferences.

Acknowledgments

We will like to thank Miguel Pessoa Monteiro for the help in developing this paper and for the constant broadening of our perspectives. We will also like to thank FCT for the support provided through scholarship SFRH/BD/32730/2006.

References

[1] Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the Hyperspace approach. In: Software Architectures and Component Technology: The State of the Art in Research and Practice. (2000)

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: 11th Europeen Conf. Object-Oriented Programming. Volume 1241 of LNCS., Springer Verlag (1997) 220–242

[3] Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness (2000)

[4] Meyer, B.: Applying "design by contract". IEEE - Computer **25**(10) (1992) 40–51

[5] Tessier, F., Badri, M., Badri, L.: A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In: International Workshop on Aspect-Oriented Software Development. (2004)

[6] Katz, S.: Diagnosis of harmful aspects using regression verification (2004)

[7] Kienzle, J., Yu, Y., Xiong, J.: On composition and reuse of aspects. In: Software engineering Properties of Languages for Aspect Technologies. (2003)

[8] Balzarotti, D., Monga, M.: Using program slicing to analyze aspect-oriented composition (2004)

[9] Störzer, M., Krinke, J.: Interference analysis for AspectJ. In: Foundations of Aspect-Oriented Languages (FOAL). (2003)

[10] Kessler, B., Tanter, É.: Analyzing interactions of structural aspects. ECOOP Workshop on Aspects, Dependencies and Interactions (ADI) (2006)

[11] Lagaisse, B., Joosen, W., De Win, B.: Managing semantic interference with aspect integration contracts. In: Software Engineering Properties of Languages and Aspect Technologies. (2004)

Disciplined Composition of Aspects using Tests

André Restivo
LIAAC - NIAD&R, Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
arestivo@fe.up.pt

Ademar Aguiar
INESC Porto, Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
aaguiar@fe.up.pt

ABSTRACT

A large part of the software development effort is typically spent on maintenance and evolution, namely on adding new and unanticipated features. As aspect-oriented programming (AOP) can be easily used to compose software in non-planned ways, many researchers are investigating AOP as a technique that can play an important role in this particular field. However, unexpected interactions between aspects are still a major problem that compromise AOP's applicability, especially in large projects where many developers, often including new team members, are involved in the process. This paper addresses the issues of aspect conflicts and interactions and proposes a technique to help compose aspects in a disciplined way using a test-driven development approach. A simple example for a banking system helps on illustrating the technique.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Languages, Verification

Keywords

Aspect-Oriented Programming, Tests, Software Evolution

1. INTRODUCTION

Successful software projects do not end after their first release. As users discover bugs to be fixed and new features to be added, several code changes are required to extend and modify the existing system. Such changes can be done by the original developers, third-party developers, or even by the users themselves. Either way, changes can often lead to the introduction of more bugs and it is usually complex to understand how a new feature or a changed feature will affect the overall system.

The special characteristics of AOP languages make them good candidates for software evolution. As most evolutionary changes are unexpected, the power of AOP to change the system behavior in unforeseen ways without the need to modify the original code seems tailored for the job. However, AOP is prone to create unexpected feature interactions that can possibly lead to conflicts.

TDD is currently a popular technique for software development. Unit tests are used to test if individual units of source code work properly while integration and system tests are used to test if these same units work well together.

When using AOP, unit tests are good enough to test single units of code, but when new behavior is added by aspects, the test results might no longer be valid. In fact, new behavior might break some of the unit tests and still this might be the desired behavior for the aspects.

One possible approach would be to write code, into the aspects, that would change not only the units' behavior but also their unit tests. However, this would not suffice as other units might depend on the original behavior and tracing an integration or system test failure back to its origin might not be as easy as in classical object-oriented programming.

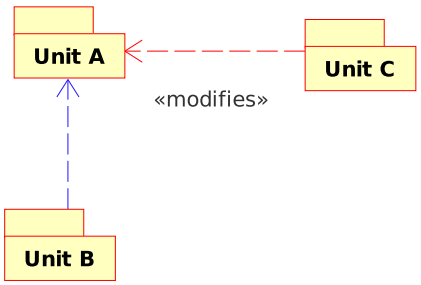

Figure 1: A simple dependency example

For example (see Figure 1), consider 2 units complemented by its unit tests that we will call *Unit A* and *Unit B*. By running its unit tests we can verify that both work correctly.

Units *A* and *B* are then integrated into a complex system and integration tests show that everything is working correctly. But after adding a new unit into the system, *Unit C*, which is a AOP unit that changes the behavior of several units in the system (including their unit tests), we find that

Unit B stopped working.

However, *Unit C* didn't modify *Unit B*. A developer would be puzzled by this behavior and classical TDD wouldn't help him to discover that *Unit B* stopped working because *Unit A*, which it depends on, was modified by *Unit C*.

In this paper we propose a test-based technique to compose aspect-oriented modules in a disciplined way. The technique aims at making AOP code easier to compose and more robust to evolution.

2. INTERACTIONS AND CONFLICTS

Software systems are composed of several artifacts (e.g. modules, packages, and classes) that interact with each other. Interactions occur when (i) artifacts depend upon other artifacts, (ii) modify other artifacts, or (iii) modify artifacts others depend upon.

Artifacts can be modified by other artifacts either by modifying its internal state (e.g. an object changing another object) or, when using AOP, by modifying its own behavior (e.g. an aspect changing the behavior of a class).

When two artifacts, that work independently as expected, interact in undesirable and unexpected ways, we are facing a conflict. The origin of conflicts was already thoroughly studied by several authors. Tessier [7] has identified the different types of conflicts as: accidental joinpoints, ordering problems, circular dependencies, and conflicts between different concerns (all of them unexpected and undesirable interactions).

To detect these conflicts we must first be capable of detecting interactions and then find out if they are conflicts or not. We could easily identify interactions if we knew the dependencies between artifacts, and also how an artifact is influencing other artifacts.

In our approach, we consider the relevant artifacts as the modules that compose the system. A module is a self-contained component of a system, which has a well-defined interface to the other components. In this way, a software system is composed by several modules and evolves when new modules are added or existing ones are modified.

Detecting if a module was modified by an aspect, although not trivial, was previously studied by others, of which we highlight here the work of Balzarotti [1]. However, to know a change occurred is not enough. It is also necessary to understand how the modification altered the behavior of the module and if it is harmless and desired.

The behavior of a module can be characterized by the features or services it provides. If we can find somehow that a feature was modified then we can try to understand how the behavior of the module has changed.

In the next section we will present a test-driven development approach that can help on detecting which features have changed due to the introduction of an aspect-oriented module.

3. TESTING FOR CONFLICTS

Tests are a simple way of analyzing if a feature provided by a module is correctly implemented, and was not modified by another module. Tests can be seen as incomplete specifications of the behavior of a module. If we create tests for each feature provided then we can use them to understand if a certain feature was not altered in some way.

In addition, if we also define exactly the features a module depends on, we can reason about interactions in a very simple way. However, not all interactions are conflicts. For example, the ultimate objective of an aspect-oriented module might be to change a particular feature. Therefore, we also need to know how each module intends to change the system. This information will enable us to distinguish between desired and undesired interactions.

In our approach, a module is defined by its code, tests and also meta-information describing the features provided, modified, and all those it depends on.

By incrementally adding each module to the system we can identify if all features needed by each module are still present, and if we are modifying the system behavior in an unexpected way.

If, when a module is being added to the system, it is detected that a feature it depends on has been removed or modified by a previously added module, there is a chance that we are facing a conflict. However, modifying the behavior of a module in order to modify the behavior of modules that depend on the modified one is not uncommon. So, a module must also be able to announce that intention by deprecating dependencies and possibly adding new alternative ones.

To clarify these concepts we will introduce a small example in the next section.

4. A SMALL EXAMPLE: A BANK

Consider the example of a banking software system that stores information about customers, their accounts, and respective transactions.

One possible way to implement this system would be to develop three separate modules, one for each of these entities, with the transaction module depending on the account module, and the account module depending on the customer module.

Consider now that the software has later evolved to accommodate two new features, persistence and security, and that both would be implemented as new separate modules using AOP.

4.1 Adding Persistence

The persistence module would probably change the customer, account and transaction classes in a way that whenever a new instance of them is created or modified that would be reflected in a storage system like, for instance, a relational database.

Tests should be implemented to verify if the internal structure of each one of these classes is persistent amongst dif-

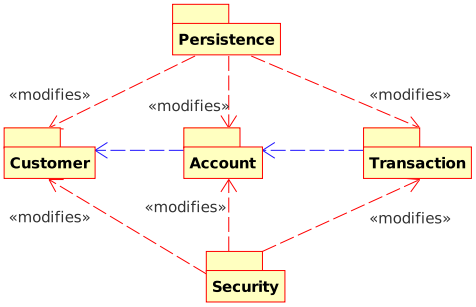

Figure 2: Module Dependencies and Modifications

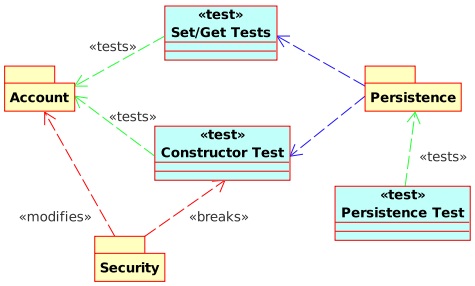

Figure 3: Security and Persistence Interactions

ferent system runs. For example, we could add a customer to an empty bank instance, recreate the bank instance and test if the customer is still present. For example, to test if the customer internal state is the same, we can serialize the customer and verify if the serialized result is the same before and after the customer has been reloaded from its persistent storage.

4.2 Adding Security

The security module would (i) add a login and password fields to the customer class, (ii) add a feature that allowed people to login into the system, and (iii) add a mechanism that would only allow new transactions to be added if the logged in person is the owner of the account in question.

Each of these features should also be tested.

The dependencies and modifications of such system are shown in Figure 2.

Both modules would announce the features they depend upon. The persistence module might depend upon the fact that the constructors, setters and getters of the base classes worked as expected.

Tests would probably exist for these features. The security module would probably need the feature that added new transactions to work properly.

Figure 3 shows the main features provided, features depended upon, and features modified by this interaction.

4.3 Persistence conflicts with Security

If the base classes are implemented correctly both modules would have the features present in the system.

However, when adding both new modules, the security module would modify the customer class in an unexpected way. If the security module is incorporated into the system after the persistence module, by running the persistence tests after the security module has been added to the system, we would identify an unexpected interaction, as the serialized version of the customer would be different from the stored one (due to the addition of the login and password fields by the security module).

On the other hand, if the persistence module is added after the security module, by running the constructor tests which the persistence module relies upon, we would discover that it was changed by the security module.

Either way, an interaction between both modules would be uncovered and it would be up to the developers to find the best way to solve it.

5. MODULE META-INFORMATION

The previous version of this technique, presented in [6], used Java annotations to store all the meta-information needed. Although sufficient for many cases, this could be insufficient as features are seldom implemented by a single class, but rather by the collaboration of several classes. Therefore, even knowing that it is not the best solution, at the moment, we are using XML files to explicitly store the needed information. Although these files are manually written at present, they could be produced automatically by tools supporting the technique. These XML files are used to store several characteristics that we need to know about each module:

- An identifier that will allow other modules to refer to it.
- A set of features provided by the module and, for each one of them, the classes and methods used to test them.
- Features from other modules that the module depends on.
- Features the module is supposed to break intentionally.
- Dependencies from other modules that this module intentionally expects to break.

Below in Listing 1, we show an example configuration file for the security module presented in Section 4.

Listing 1 Module Definition Example

```
<module name="security">
  <feature name="accountHasOwner">
    <test class="com.bank.security.accountOwner" method="*" />
  </feature>
  <feature name="accessRestrictions">
    <test class="com.bank.security.accessTest" method="*" />
  </feature>
  <depends module="transaction" feature="addTransaction" />
</module>
```

For convenience purposes of the supporting tools developed till present, we expect that modules will be self contained in either their own directories or in jar files. This way, each module could have its own meta-information file.

The next section will explain how this technique can be used to compose a software system in a disciplined way.

6. ASPECTS COMPOSITION

The process of composing software using the technique described in this paper can be automated using the following steps:

- Find the best order in which the modules can be incrementally built into the system. This can be accomplished by using a slightly modified topological sort of the dependency graph.

Then, for each module:

- Start by verifying if all needed features are still present in the system. If they are not, warn the developer of which features are missing and which module has broken them and stop.
- Incrementally compile and add the module into the system.
- Test if all features provided by the module are working. If not warn the developer and stop.
- Test if all previously tested features are still working. If some of them fail, warn the developer and stop, unless some module states that it intentionally broke the feature.
- Test if the features that some module stated it had intentionally broken are really not working. If this is not true warn the developer and stop.
- Verify if any intentionally broken features were needed by a previously compiled module. If so warn the developer and stop.

Although a supporting tool implementing this technique was already developed, it still uses the earlier ideas presented in [6]. A new version using the XML annotation files described in Section 5 is still under development.

This tool will be capable to read the meta-information files from an entire project and incrementally compile the whole system warning the developers of any unexpected interaction it finds. The tool will also support the creation of the XML files and will provide a visual representation of the dependency graph.

Using this tool for the project presented in Section 4 is expected to be easy. After importing the AspectJ project into the tool, the user would only need to use a wizard to create each meta-information file. At first, each file would only contain information about the features provided and those it depends on. After compiling the project for the first time, the tool would warn the user about the security module changes to the customer class constructor. These changes would have been detected because a test existed for that feature or service. The user would be given the choice to add information to the XML file about that change being intentional. A second attempt at compiling the project would warn the user about the dependency between the persistence module and the customer constructor having been broken by the security module. This would allow the developer to understand exactly how both modules interact with each other.

7. RELATED WORK

Several researchers have been working to solve this same problem. In this section we present some of the other studies in this field that we consider to be the most relevant. We separated the techniques into automatic analysis techniques and user controlled analysis techniques.

7.1 Automatic Conflict Detection

Balzarotti [1] claims that the interaction detection problem can be solved by using a technique proposed in the early 80s, called program slicing. Although totally automatic, this technique does not account for intended interactions.

Havinga [2] proposed a method based on modeling programs as graphs and aspect introductions as graph transformation rules. Using these two models it is then possible to detect conflicts caused by aspect introductions. Both graphs, representing programs, and transformation rules, representing introductions, can be automatically generated from source code. Although interesting, this approach suffers the same problem of other automatic approaches to this problem as intentional interferences cannot be differentiated from unintentional ones.

7.2 User Controlled Conflict Detection

Lagaisse [5] proposed an extension to the Design by Contract (DbC) paradigm by allowing aspects to define what they expect of the system and how they will change it. This will allow the detection of interferences by other aspects that were weaved before, as well as the detection of interferences by aspects that are bounded to be weaved later in the process.

Katz [3] proposed the use of regression testing and regression verification as tools that could help identifying harmful aspects. The idea behind this technique is to use regression testing as normally and then weave each aspect into the system and rerun all regression tests to see if they still pass. If an error is found, either the error is corrected or the failing tests have to be replaced by new ones specific for that particular aspect.

It has been noticed by Kienzle [4] that aspects can be defined as entities that require services from a system, provide new services to that same system and removes others. If there is some way of explicitly describing what services are required by each aspect it would be possible to detect interferences (for example, an aspect that removes a service needed by another aspect) and to choose better weaving orders.

7.3 Related Work Analysis

Automatic conflict detection techniques all suffer from the same problem. Although they are easy to use, they detect a large amount of false positives. Unless this problem can be tackled in some unforeseen way they seem to be just interesting analysis tools but not good enough for software testing and validation.

The user controlled techniques that were exposed are similar in their core ideas to our own work. Although each one uses a different approach (test, contracts and services), the main idea is always the same. Our own approach builds on these ideas by adding the notion of dependency (proposed by Kienzle) on top of Katz ideas of using regression testing.

8. CONCLUSIONS

Using AOP as a tool for software evolution is useful but due to the decoupling of features, reasoning about the code becomes harder, thus making interactions more difficult to understand. This leads to a bigger risk of conflicts and more difficulty on understanding why they happen.

This paper presented a technique based on tests that allows developers to better understand how their code is affecting other parts of the system and also to help them correct their code. Conflicts can then be detected easier and expected interactions can be described using external XML configuration files avoiding the detection of false positives.

Although recognizing that this technique does not solve all problems caused by the use of AOP, such as poorly written code or tests, or even quality code that might have conflict problems not detected by this technique, we envision that, as a complement to other analysis-based approaches, a tool based on these ideas would prove useful for software evolution using AOP.

The major difficulty that this approach yields is that it depends a lot on the developers and their skills with unit-testing. With the increasing popularity of TDD and with the help of the support tools (that are still to be developed) we believe this problem can be overcome.

9. FUTURE WORK

In the near future we plan to create a catalog containing a list of several conflict types. This catalog will also contain recipes and running examples that will allow developers to test and compare their conflict solving techniques.

We have also planned to finish and release the tool that will support the technique in question and improve the presented method based on experiments.

The proposed approach might also be valid for other kinds of problems like using AOP in Software Product Lines or simple composition problems in AO systems. Subsequent work will explore the use of this technique in these particular cases.

Finally, we expect to be able to prove formally that the technique works for all cataloged cases.

10. ACKNOWLEDGMENTS

We will like to thank Ana Moreira, Cristina Videira Lopes, and Miguel Pessoa Monteiro for their help on constantly broadening our perspectives regarding conflicts due to aspect composition.

A word of appreciation goes also to all the reviewers that helped improve this paper with their insightful comments, namely LATE 2008 workshop reviewers.

We will also like to thank FCT for the support provided through scholarship SFRH/BD/32730/2006.

11. REFERENCES

[1] D. Balzarotti and M. Monga. Using program slicing to analyze aspect-oriented composition, 2004.

[2] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 85–95, New York, NY, USA, 2007. ACM Press.

[3] S. Katz. Diagnosis of harmful aspects using regression verification, 2004.

[4] J. Kienzle, Y. Yu, and J. Xiong. On composition and reuse of aspects. In *Software engineering Properties of Languages for Aspect Technologies*, 2003.

[5] B. Lagaisse, W. Joosen, and B. De Win. Managing semantic interference with aspect integration contracts. In *Software Engineering Properties of Languages and Aspect Technologies*, 2004.

[6] A. Restivo and A. Aguiar. Towards detecting and solving aspect conflicts and interferences using unit tests. In *SPLAT '07: Proceedings of the 5th workshop on Engineering properties of languages and aspect technologies*, page 7, New York, NY, USA, 2007. ACM.

[7] F. Tessier, M. Badri, and L. Badri. A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In *International Workshop on Aspect-Oriented Software Development*, 2004.

Testing for Unexpected Interactions in AOP

André Restivo
Artificial Intelligence and Computer Science Laboratory
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
arestivo@fe.up.pt

Ademar Aguiar
INESC Porto
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
aaguiar@fe.up.pt

***Abstract*—Aspect Oriented Programming (AOP) is a recent and powerful programming technique with the objective of improving modularity by encapsulating crosscutting concerns. The nature of AOP makes it prone to unexpected and harmful interactions between the different components of a system.**

The claim behind this PhD is that unit tests can be used to detect these interactions.

In this paper we explain how these can be accomplished. A brief state of the art, work plan and a support tool (drUID) are also presented.

I. Introduction

Aspect Oriented Programming (AOP) is a programming technique that allows crosscutting concerns to be separated into their own units of modularity.

This improved modularization is usually achieved by allowing developers to identify points in the control flow of the program, called joinpoints, where pieces of code, called advices, can be applied. Joinpoints can also be grouped together into something that is usually called a pointcut. An aspect is then composed of pointcuts and advices in such a way that concerns that usually crosscutted several modules of the application can be kept in a single place.

AOP has already proven to be an important step towards better code modularization. However, by allowing developers to change the control flow of a program, it increases the chances of unexpected interactions between different modules occurring.

For example, imagine an application where messages are exchanged between a server and several clients. This could be easily done by using a classical Object Oriented approach by creating a *server* class and a *client* class. These classes would know how to communicate between them by using some other *TCP* class.

Suppose that we want to use AOP in order to cypher the messages between the server and the client. This could be easily done by targeting the joinpoints where each class sends their messages and cypher the messages before they are sent. We would then only need to do the same thing in the receiver side and target the joinpoints where the messages are received and decrypted. This could be done in a single aspect thus keeping the *cypher* concern in a single unit of modularity. If everything goes as planned, the main concern of the application, sending and receiving messages, would interact with the *cypher* concern in a correct an expected way.

However, if another aspect that removed curse words from the text being sent was added to the system, it could easily behave in an unexpected way. This would happen if the *cypher* concern was applied before the *remove curses* concern as it would impede it from having access to the plain text version of the message being sent.

In this example, we have two aspects that when applied individually to the base code work correctly but interact with each other in and unexpected and incorrect way.

We claim that using a carefully planned testing methodology would allow to detect at least some of these unexpected interactions.

II. Related Work

Several researchers have been working to solve this same problem. In this section we present some of the other studies in this field that we consider to be the most relevant. We separated the techniques into automatic analysis techniques and user controlled analysis techniques. A more complete review of the state of the art can be found in [8].

A. Automatic Interactions Detection

Balzarotti [2] claims that the interaction detection problem can be solved by using a technique proposed in the early 80s, called program slicing. Although totally automatic, this technique does not account for intended interactions.

Havinga [3] proposed a method based on modeling programs as graphs and aspect introductions as graph transformation rules. Using these two models it is then possible to detect interactions caused by aspect introductions. Both graphs, representing programs, and transformation rules, representing introductions, can be automatically generated from source code. Although interesting, this approach suffers the same problem of other automatic approaches to this problem as intentional interferences cannot be differentiated from unintentional ones.

B. User Controlled Interaction Detection

Lagaisse [6] proposed an extension to the Design by Contract (DbC) paradigm by allowing aspects to define what they expect of the system and how they will change it. This will allow the detection of interferences by other aspects that were weaved before, as well as the detection of interferences by aspects that are bounded to be weaved later in the process.

Katz [4] proposed the use of regression testing and regression verification as tools that could help identifying harmful aspects. The idea behind this technique is to use regression testing as normally and then weave each aspect into the system and rerun all regression tests to see if they still pass. If an error is found, either the error is corrected or the failing tests have to be replaced by new ones specific for that particular aspect.

It has been noticed by Kienzle [5] that aspects can be defined as entities that require services from a system, provide new services to that same system and removes others. If there is some way of explicitly describing what services are required by each aspect it would be possible to detect interferences (for example, an aspect that removes a service needed by another aspect) and to choose better weaving orders.

III. Research Objectives and Approach

Unexpected interactions are one of the main issues preventing AOP from becoming a mainstream development approach. Obliviousness, the capability of developing aspects without having to reason about other concerns of the problem at the same time, has been tagged as the culprit of this problem.

Several ways of tackling this issue have been proposed in literature but none has proven sufficiently effective or even widely accepted by the community.

Having in mind the difficulties of achieving modularity with AOP, the proposed work has the following objectives:

1) To describe, formally, the different types of interactions in order to better understand the scope of the problem.
2) To develop a test driven approach to allow the specification of incompatibilities and dependencies between aspects.
3) To develop a supporting tool, integrated into an existing development platform, to aid in the development of aspects by ensuring compatibility between them.

All these objectives aim to prosecute a more fundamental goal that is to improve the usability of AOP languages making obliviousness, when desired, a reality.

IV. Current Work and Preliminary Results

In this section we will present the current state of our work. We will start by explaining the concept of dependency graphs in this context. Then we will show how these graphs can be used to compile programs incrementally in order to allow the detection of interactions. We will by presenting a support tool that implements the technique presented.

A. Dependency Graphs

Each module that composes an application has a set of other modules that it depends upon. These dependencies can be of different forms. For example, a module that calls a function from another module has a dependency relationship with that module as it will not work if the module, where the called method is implemented, is not present in the system.

We can obtain the dependency graph of an application manually or by using some kind of code inspection technique.

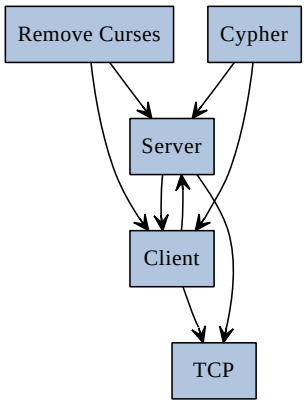

Fig. 1: Dependencies between Modules

Figure 1 contains the dependency graph for the example introduced earlier.

This graph can be used to choose an order in which the system can be composed module by module. This incremental composition allows us to test the system step by step while searching for interactions between modules.

To obtain this ordering we first use a *strongly connected components* (SCC) algorithm to find cycles in the graph. These cycles will be treated as single modules. And then we do a *topological sort* (TS) to compute one possible composition order.

In the next section we will show how this graph can be used to detect interactions between modules.

B. Testing for Interactions

With the dependency graph sorted, we start by choosing the module that does not depend on any other modules, compile it and run unit tests against it. In the example that we have been using this would be the *TCP* module.

If the module is correct, the unit tests will all pass. We then proceed by choosing one of the modules that depend only on modules already tested. In this case there is a circular dependency between the *Server* and *Client* modules. This dependency would be detected by the *SCC* algorithm and both modules would be treated as one.

This means that the next step is to compile the *TCP*, *Server* and *Client* modules together into a single application. We then proceed by running tests against the modules composed in previous steps. If one of these tests fail, it means that there is an interaction between the modules compiled previously and the ones just added to the system. If all tests pass then we run the tests against the new modules to assure they are working properly.

This process continues by choosing either the *Remove Curses* module or the *Cypher* as both depend only on modules already tested. Assuming we choose to compile the *Remove Curses* module first and after applying the same process all tests are successful we are left with only the *Cypher* module. By applying the same process to this last module, we would

find that the tests for the *Remove Curses* module would fail. This means that an interaction between these 2 modules exist as the module worked before the *Cypher* had been added.

We can summarize our claim by using a more generic example. If we have a program *P* composed by modules *M1* through *Mn*, and we take a subset *P'* of *P* where all dependencies are satisfied and tests for all the modules contained in *P'* pass, and if we take a module *Mi* from *P - P'* and add it to *P'* creating a new subset *P"*, we can get three different scenarios:

- All tests on *P"* pass. This means there are no interactions between the modules in *P'* and module *Mi*.
- All tests on *P'* pass but some tests for module *Mi* fail. This means that there is an interaction between module *Mi* and one of the modules contained in *P'*.
- Some tests on *P'* fail. These means that there is an interaction between the module of *P'*, whose test failed, and module *Mi*.

This of course assumes that the tests are well designed and have sufficient code coverage to detect the fault introduced by the interaction.

C. Unexpected Interactions

Not all interactions detected by the process just proposed are unexpected or even harmful. Invasive aspects can change the behavior or flow of a program in an intended way. After all, they have a specific purpose and most of the times that purpose is exactly to change how the code works.

After an expected interaction has been detected, the developer could simply change the unit tests of the modules whose behavior was changed by aspects in other modules. This would not be a good solution as it would break modularity and prevent the reuse of modules.

In our approach, we expect developers to specify which behaviors they expect to break with their aspects. This can be done before the detection of an interaction, preventing false positives, or after, preventing the interaction from being detected the next time around. Besides stating which behaviors are broken, developers can add tests for the new behavior of the altered modules.

In order to improve the granularity of the process, dependencies can be stated at the feature level. Each module exports several features (i.e. the behaviors of the module as seen by the user or by other modules through an API) and each one of them can depend on one or more features from other modules. This still allows us to calculate the dependency graph and, at the same time, improves the feedback obtained by the process. Other advantage of this approach is that it makes it easier to insert information about the dependencies inside the source code of the application making the whole process easier to use.

D. Tool Support

drUID [7] is an Eclipse plug-in that helps AspectJ developers to detect interactions between aspects using the technique that has been presented in this paper. In order to accomplish this, the plug-in allows developers to define several characteristics about system artifacts using Java annotations:

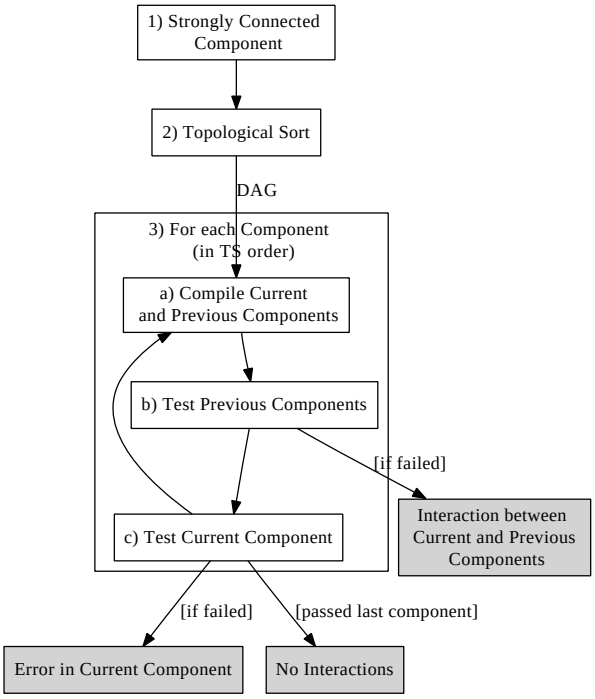

Fig. 2: Interaction Detection Process

- Which features are provided by each artifact.
- Which JUnit tests are used to test each feature.
- Which features depend on other features.

Several aids have been implemented to guide the developer in this process in the form of Eclipse quick fixes and quick assists. Each time a file is saved in Eclipse, the annotations are inspected and any errors are reported. Besides that, a dependency graph is created and shown in a graphical form that allows the developer to navigate through the code following the dependencies between artifacts.

With the dependency graph created, the developer can order drUID to execute an interaction test analysis. This analysis executes the following steps (Figure 2):

1) Execute a Strongly Connected analysis of the dependency graph transforming it into a Directed Acyclic Graph (DAG) of components (sets of artifacts).
2) Execute a Topological Sort in order to determine in which order the components must be compiled.
3) For each component:
   a) Compile it.
   b) Execute the tests defined for the features provided by this component.
   c) Execute the tests defined by previous components.

Step 3b verifies if the component is working as expected, while step 3c verifies if previous features have been broken by the newly introduced component.

When the later happens, an interaction has been detected and the analysis stops. This interaction is reported to the developer with an explanation of which feature has been broken by which component. This will allow the developer to understand what went wrong and act accordingly. Two different conclusions can be extrapolated by the output generated by the tool: (i) the interaction is expected (ii) the interaction reveals a unexpected interaction between aspects.

In the first case, the developer simply has to declare the interaction by using, once again, annotations. By using Eclipse quick fixes added by the plug-in this can be done easily by the developer.

In the later case, we are confronted with a unexpected interaction that has to be dealt with in another way. If possible the developer can change the implementation of one of the aspects in order to remove the interaction. But in some cases, the interaction might have been caused by two incompatible requirements and, in that case, it can only be corrected by changing the software requirements themselves.

E. Evaluation

There are several aspects of the technique that need to be validated:

1) Is the technique applicable to all possible setups?
2) Does the technique detect all kinds of interactions or are there some that simply don't fit into our model?
3) Is it feasible to use the technique in a real world scenario or is the added workload too much of a burden for developers?

We expect to validate these points using three different approaches:

1) Creating a catalog of different interactions, with code examples, and applying the technique to them. This will allow a certain degree of confidence about the universality of the technique.
2) Creating a mathematical model of the claim and formally proving that it works for all possible cases. Or, if we end up discovering it does not, explaining which interactions it fails to detect.
3) Using the technique in real world, or almost real world, scenarios and annotating the overhead introduced by the extra work needed to implement the technique.

V. Work Plan and Implications

The plan devised for this research work encompasses seven main phases and will make use of diverse research methods and experimental approaches. These phases are:

1) State of the Art Review - This task aims at gathering information about the several topics relevant for the considered problem, namely: Aspect-Oriented Programming, Formal Methods (in particular Design by Contract) and Regression Testing (in particular Unit Testing).
2) Exploratory Projects Development - The development of one or several small projects using AOP in order to better understand the interference problems existing in the field and how they can be tackled.
3) Hypothesis and Problem Definition - The precise definition of the research questions and the formulation of a hypothesis will be the main result of this task. Furthermore, it comprises the definition of the parameters that will be used for validating and assessing the approach.
4) Conceptual Framework Development - Development of the conceptual framework that will allow the detection of interferences and conflicts between aspects.
5) Development of Supporting Tools - Development of a supporting tool, integrated into an existing development platform, to aid developing aspects using formal methods or regression testing to ensure compatibility between them.
6) Result Consolidation - This phase starts after the definition of validation criteria in the scope of the Hypothesis Definition, and gains momentum after a first version of the approach can be validated experimentally.
7) Writing of the Thesis - This task will accompany the considered phases for the research work as the respective milestones contribute for its completion.
8) Participation in Conferences - As the thesis progresses the parcial results will be presented in conferences related to the AOP subject. Papers have already been presented in the SPLAT'07 [8] and LATE'08 [9] workshops. These workshops were part of the AOSD conference (the most important AOP conference). Papers were also presented in the CoMIC'06 [10] and CoMIC'07 [11] Doctoral Symposiums.

VI. Conclusions

We presented a technique that allows the detection of interactions between aspects using unit tests. We showed how the technique can be used by developers to better understand how their code interacts with other aspects in the system and also what they can do to correct eventual unexpected interactions. We also showed how annotations can be used to signal false positives and improve the overall performance of the technique.

We recognize that this technique does not detect all possible interactions introduced by AOP languages. The quality of its response will always be tied to the quality of code and tests. Even quality code with good tests can sometime have interactions that are not detected by the technique if the changes introduced by the aspects are of such an unexpected nature that they are not accounted for in the unit tests.

Currently we are working on evaluating the technique but there are still several improvements to be done:

1) Automatically detect dependencies using code analysis. Some dependencies should be impossible to find but detecting most of them would provide a faster and more usable process.
2) Improve the Eclipse plug-in.
3) Investigate a possible relation with Feature Driven Development.

ACKNOWLEDGMENT

We will like to thank Ana Moreira, Cristina Videira Lopes, and Miguel Pessoa Monteiro for their help on constantly broadening our perspectives regarding interactions due to aspect composition.

We will also like to thank FCT for the support provided through scholarship SFRH/BD/32730/2006.

REFERENCES

[1] A. Restivo, *Disciplined Reuse of Aspects. State of the Art and Work Plan.*, Technical Report, Faculdade de Engenharia da Universidade do Porto, 2007.

[2] D. Balzarotti and M. Monga, *Using program slicing to analyze aspect-oriented composition*, Proceedings of Foundations of Aspect-Oriented Languages (FOAL) Workshop at AOSD 2004, Lancaster, UK, 2004.

[3] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit, *A graph-based approach to modeling and detecting composition conflicts related to introductions*. Proceedings of the 6th international conference on Aspect-oriented software development, pages 85–95, New York, NY, USA, 2007. ACM Press.

[4] S. Katz, *Diagnosis of harmful aspects using regression verification*, Proceedings of Foundations of Aspect-Oriented Languages (FOAL) Workshop at AOSD 2004, Lancaster, UK, 2004.

[5] J. Kienzle, Y. Yu, and J. Xiong, *On composition and reuse of aspects*, Software engineering Properties of Languages for Aspect Technologies, 2003.

[6] B. Lagaisse, W. Joosen, and B. De Win, *Managing semantic interference with aspect integration contracts.*, Software Engineering Properties of Languages and Aspect Technologies, 2004.

[7] A. Restivo and A. Aguiar, *DrUID : Unexpected Interaction Detection*, Demo at the Aspect Oriented Software Development Conference (AOSD) 2009, Charlotesville, VA, USA.

[8] A. Restivo and A. Aguiar, *Towards Detecting and Solving Aspect Conflicts and Interferences Using Unit Tests* Software Engineering Properties of Languages and Aspect Technologies (SPLAT'07), Vancouver, B.C., Canada, 2007.

[9] A. Restivo and A. Aguiar, *Disciplined Composition of Aspects using Tests*, Linking Aspect Technology and Evolution (LATE'08), Brussels, April 2008.

[10] A. Restivo, *The Case for Aspect Oriented Programming*, 1ª Conferência em Metodologias da Investigação (CoMIC'06), Portugal, January 2006.

[11] A. Restivo and A. Aguiar, *Aspects: Conflicts and Interferences (A Survey)*, 2ª Conferência em Metodologias da Investigação (CoMIC'07), Portugal, February 2007.

Incremental Modular Testing

(to be submitted to Modularity'16)

André Restivo
Faculdade de Engenharia da Universidade do Porto, Portugal
arestivo@fe.up.pt

Ademar Aguiar
Faculdade de Engenharia da Universidade do Porto, Portugal
aaguiar@fe.up.pt

Ana Moreira
NOVA LINCS, Universidade Nova de Lisboa, Portugal
amm@fct.unl.pt

Abstract

By designing systems as sets of modules that can be composed into larger applications, developers unleash a multitude of advantages. The promise of AOP is to enable developers to organize crosscutting concerns into separate units of modularity making it easier to accomplish this vision. However, AOP does not allow the untanglement of unit tests, which impairs the development of properly tested independent modules.

This paper presents a technique that enables developers to encapsulate crosscutting concerns using AOP and still be able to develop reusable unit tests. Our approach uses incremental testing and invasive aspects to modify and adapt tests.

The approach was evaluated in a medium scale project with promising results. Without using the proposed technique, due to the presence of invasive aspects, some unit tests would have to be discarded or modified to accommodate the changes made by them. This would have a profound impact on the overall modularity and, in particular, on the reusability of those modules. We will show that this technique enables proper unit tests that can be reused even when coupled with aspect-oriented code.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Testing and Debugging

Keywords testing, aspects, modularity

1. Introduction

The development of large software projects is a complex task. Unfortunately, humans often struggle when asked to cope with complex problems. The way we usually deal with this is by decomposing the larger problem into several smaller and more manageable ones. When talking about actual code, we usually call these smaller pieces of software as *modules*.

To reap as many advantages as possible from this division into smaller modules, several important aspects should be taken into consideration. Modules should have low coupling and high cohesion between them, and concerns should not be spread over several modules or tangled inside one. A good decomposition should lead to modules that can be described, reused, replaced and tested in isolation.

Aspect oriented software development promises to enable developers to achieve this kind of isolation, not only at the code level, but also at the requirements and design phases. By using aspects, developers are able to separate each concern into its own unit of modularity. Having concerns untangled improves reusability as the code of each module pertains only to a single concern.

However, when modules are reused, there are other artifacts, besides the actual code, that must be transferred between projects. Some of those artifacts are all the tests that help to develop reliable software.

In this paper, we argue that due to the nature of aspects, some unit tests cannot be reused in different contexts thus impeding module reusability. Testing modules in isolation also becomes harder. We will show how using a testing technique based on incremental compilation can help mitigate this problem.

In Section 2 the problem we propose to tackle will be identified. Section 3 describes a technique, based on incremental testing, that aims at solving the proposed problem. Section 4 presents a *Eclipse* based implementation of the technique. Our efforts to validate the solution are presented in Section 5, and Sections 6 and 7 describe related work and our conclusions.

Modularity '2016, March 14–17, 2016, Málaga, Spain.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. The Problem

Let us start with a simple example. Imagine we have a base class called *Question* that represents a multiple choice question in an exam (see Listing 1 for a simplified sample of the class).

Listing 1. Question Class

```
public class Question {
  public void addChoice (String choice) {...}
  public String getChoice(int number) {...}
}
```

This class is part of a module with the same name and has several tests. One of these tests adds some choices to the question and verifies if they are present and in the correct order (see Listing 2 for a simplified version of the test).

Listing 2. Question Test

```
public void testChoices() {
  Question q = new Question("Choose a color?");
  q.addChoice("blue");
  q.addChoice("red");
  q.addChoice("green");

  assertEquals("blue", q.getChoice(0));
  assertEquals("red", q.getChoice(1));
  assertEquals("green", q.getChoice(2));
}
```

As we have postulated before, this test should also be reusable. If the *Question* module is reused in another project, it should also be possible to reuse the test in that project without any modifications. In fact, both test and code should be seen as parts of a single reusable artefact.

In a classical object-oriented environment, this would not be a problem. This class and its test would always work in the same way regardless of any other artifacts present in the system. This does not happen when aspects are present. Imagine that we add an aspect called *RandomizeChoices* that changes the positions of the choices at random as they are added (see Listing 3 for a small excerpt of this aspect).

Listing 3. Randomize Aspect

```
pointcut addChoice(List list, String choice) :
  cflow(call(void Question.addChoice(..))) &&
  !within(Randomize) &&
  target(list) && args(choice) &&
  call(boolean List.add(..));

boolean around(List list, String choice) :
    addChoice(list, choice) {
 int position = random.nextInt(list.size() + 1);
 list.add(position, choice);
 return true;
}
```

When this aspect is added, our *testChoices* test will start failing five times out of six. We did not change the code of the *Question* class or any code this class depends on. In object-oriented programming, the problem of having code from outside a module influence the outcome of a test was solved by using *mocks* and *stubs* [4]. The difference is that, when dealing with objects, only the code the module depends on can alter its behavior. As we just saw, that is not the case when coding with aspects. In the next paragraphs we will describe some naive solutions for this problem.

Moving the test. The most simple solution would be to move the offending test from the *Question* module to the *RandomChoices* module and change it to accommodate the new concern. This could be easily achieved by using the *contains* method to test for the presence of a choice instead of looking for it in the expected location. The problem with this approach is that the *Question* module would lose the *testChoices* test. This would make it harder to reuse this module in other systems. At the same time, the basic functionality of being able to add choices to questions would no longer be tested separately.

Changing the test. By altering the *testChoices* test to accommodate the changes introduced by the *RandomChoices* aspect, we could easily make it work again. This is the same solution as the one we saw before but instead of moving the test to the other module and making the changes there, we just change the test we already have. This would also have the effect of making the *Question* and *RandomChoices* concerns tangled with each other – not at the working code level but at the testing level – preventing the *Question* module from being easily reused and leaving the *RandomChoices* without any tests.

Using aspects to change the test. We could keep the *Question* module code unchanged and use an aspect to change the *testChoices* test behavior whenever the *RandomChoice* module is present in the system. This way the *Question* tests would work as planned when the module is used in isolation and the *RandomChoices* module would have a different test for its own behavior. Although having some advantages, the problem with this approach is the same as in the previous one. The difference is that the tangling now happens in the *RandomChoices* module.

None of these solutions gives us a scenario where modularity is preserved in its entirety. However, we could summarize the principles of what would be a good solution:

1. **Obliviousness.** Tests should only test the behavior of their own modules.
2. **Completeness.** All concerns should have their own tests and all tests should run at least once.
3. **Correctness.** When a module is reused in a different context, tests should still work correctly.

In the next section, we will describe a technique based on incremental compilation that allows the usage of unit tests without breaking modularity. For the sake of completeness we will explore four different ideas that will converge into

our final proposition and we will explain how these compare to one another in several different aspects.

3. Incremental Testing

A well-designed software system should be built in such a way that low-level modules do not depend on higher level modules. Software systems should be built layer by layer, with each layer adding more functionalities. If this is accomplished, then the modules dependency graph becomes a *directed acyclic graph* (DAG).

Unfortunately, not all software systems follow this recommendation and it is common to find circular dependencies even in the most well-designed software systems. In graph theory, these collection of nodes that form circular dependencies are called *strongly connected components*, and although they cannot be easily removed, they can be isolated. We do this by considering each *strongly connected component* in the graph as a super module. In this way, we can consider that all software systems can be thought as being composed as a DAG of module dependencies.

If we are able to extract this dependency graph from the source code, we can be sure that we will have at least one low-level module, lets call it module *A*, that does not depend upon any other module. This module can be compiled and tested in isolation shielding it from the potential influence of higher level aspects.

After this initial module is tested, we can take another module that only depends on this module, lets call it module *B* and test both of them together. Tests from module *B* can be easily shielded from eventual errors in the source code of module *A* by using classic object-oriented unit testing techniques like *mocks* and *stubs*. On the other hand, tests from module *A* can still be influenced by aspects on module *B* making them fail. We already ran the tests of module *A* once, so all we need to do is to make sure that tests that fail under the influence of aspects from module *B* are not run again. This process can then be repeated for every module in the system until all tests have run at least once.

When we first started researching this idea [14], we postulated that tests could be used to detect unexpected interferences caused by aspects. An unexpected interference happens when a module containing invasive aspectual code changes the behavior of another module in unforeseen and undesirable ways.

If tests are in place, these interferences can be easily detected. However, there will be no apparent difference between an unexpected interference and an expected interaction. The developer must be able to differentiate between the two of them and act accordingly. Interferences must be fixed and interactions must be dealt with; not because they are wrong but because they impede the testing process.

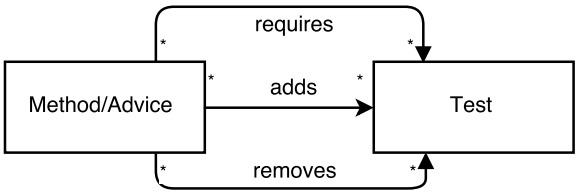

Figure 1. Method-Test approach

To fix the testing process, the developer must be able to specify that a certain interaction is desirable. After that, the testing process can ignore any tests that fail due to that interaction but only after the test has been successfully executed without the offending aspect. In our initial approach we considered using code annotations to enable the developer to specify these interactions. In the following sections, we will demonstrate how that initial approach evolved and present the advantages and drawbacks of each step.

3.1 Method-Test Approach

Our initial idea was to consider tests as being the proof that a certain concern was implemented correctly. Ideally, for every concern in the system, the developer should be able to create a test for it. Class methods are the artefacts that end up implementing those concerns. So we could have annotations in each method with a reference to the test for the concern that the method was implementing.

In order to create the DAG of dependences needed for our incremental testing process, we proposed another annotation where each method could declare the tests it depends on. Notice that we do not specify which methods the method depends on, but the tests that were created to test that method. This means that every time an aspect is added to the system, in our incremental testing process, and a previously tested test fails, we can pinpoint which methods are affected by that interaction.

Finally, we proposed another annotation that allows developers to declare expected interactions. This annotation would be used by developers on advices to pinpoint which tests they expect to break. Figure 1 contains a representation of the connections achieved by these annotations. Listing 4 represents a sample of the code needed to implement this approach for the example described in the beginning of this paper.

Listing 4. Method-Test Approach

```
public class Question {
  @Adds("QuestionTest.testChoices")
  public void addChoice (String choice) {...}
  @Adds("QuestionTest.testChoices")
  public String getChoice(int number) {...}
}

/* Inside Question Test Suite */
public void testChoices() {...};
```

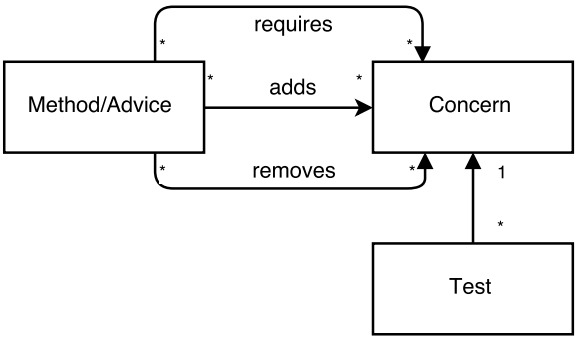

Figure 2. Concern-Test approach

```
/* Inside Randomize Aspect */
@Removes("QuestionTest.testChoices")
boolean around(List list, String choice) :
  addChoice(list, choice) {...};
```

Having these annotations in place, our testing process would be able to create the DAG of dependencies, using the *requires* and *adds* annotations, and run only the tests that have not been removed by subsequent advices by means of the *removes* annotation.

Besides the extra effort put in by the developer, the problem with this approach is that the relation between methods/advices and tests is artificial.

3.2 Concern-Test Approach

To mitigate the artificiality of our first approach, we decided to add a new annotation that would depict a *concern*. In this approach, each method has an annotation stating which concern it implements. These concerns can even be derived from the requirements phase.

In this way, methods and advices no longer add, remove or depend on tests but on concerns. To know which concern is tested by each test we need an annotation that will be applied to each test with a reference to the concern. Figure 2 shows the relationships between the code artifacts derived from the annotations in the code. Listing 5 represents a sample of the code needed to implement this approach for the example described in the beggining of this paper.

Listing 5. Concern-Test Approach

```
public class Question {
  @Implements("questionHasChoices")
  public void addChoice (String choice) {...}
  @Implements("questionHasChoices")
  public String getChoice(int number) {...}
}

/* Inside Question Test Suite */
@Tests("Question.questionHasChoices")
public void testChoices() {...};

/* Inside Randomize Aspect */
@Removes("Question.questionHasChoices")
boolean around(List list, String choice) :
  addChoice(list, choice) {...}
```

To apply our proposed testing process using this approach, we start by selecting a module whose methods do not depend on any concern from another module. Tests for the concerns defined in the module are run. In each step we add another module that only has *requires* annotations referencing concerns added by modules that already have been tested. If a test that passed in a previous step fails after a new module is added we can infer that there is an interaction between a concern implemented in that module and the concern that the failing test was testing.

In comparison with the first approach, this one has a richer set of metadata on the implemented concerns and their tests. This extra knowledge allows us to better understand which concerns are interacting with each other. Developers can therefore reason more easily if the interaction is expected or if it is an unexpected interference.

3.3 Module-Test Approach

The previous two approaches imposed an heavy burden on the developers as they had to add a lot of annotations to the code. In this iteration we tried to reduce the amount of extra work needed by removing most of them.

We started by considering modules as being defined by the way the used language, in this case *AspectJ*, defined its own units of modularity – *Java* packages. To prevent cases where the relation between the language defined units and the intended modules is not a direct one, we added an optional annotation so that each class/aspect could define to which module it belongs.

Tests defined inside a module are considered as being used to test some concern of the module. This removed the burden to add annotations for each test.

The only annotations really needed, are between tests. The *replaces* annotations identify cases where a test represents a concern, developed as an invasive aspect, that changes the behavior of another concern that is tested by the other test. Figure 3 shows the relationships between the code artifacts derived from the annotations in the code. Listing 6 represents a sample of the code needed to implement this approach for the example described in the beginning of this paper.

Listing 6. Module-Test Approach

```
public class Question {
  public void addChoice (String choice) {...}
  public String getChoice(int number) {...}
}

/* Inside Question Test Suite */
public void testChoices() {...};

/* Inside Randomize Aspect */
boolean around(List list, String choice) :
```

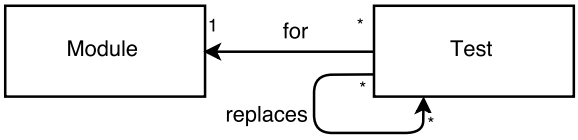

Figure 3. Module-Test approach

```
  addChoice(list, choice) {...}

/* Inside Randomize Test Suite */
@Replaces("Question.QuestionTest.testChoices")
public void testRandomChoices() {...};
```

This approach drastically reduced the amount of extra work by the developer. However, the information gathered is much less. But still, when interactions are detected we can get information about which test failed and which modules caused the interaction. This information should be enough for the developer to identify the origin of the problem and act accordingly.

3.4 Advice-Test Approach

The last approach considered was an easy evolution from the previous one. The only mandatory annotation in our previous approach was used to remove a test from the system when a module containing invasive aspects was added to the system changing the behavior the test was testing.

An alternative would be to use an advice to disable the test. Listing 7 shows how that can be accomplished by simply adding an around advice that does not call the original captured joinpoint from the test.

Listing 7. Module-Test Approach

```
public class Question {
  public void addChoice (String choice) {...}
  public String getChoice(int number) {...}
}

/* Inside Question Test Suite */
public void testChoices() {...};

/* Inside Randomize Aspect */
boolean around(List list, String choice) :
  addChoice(list, choice) {...}
void around() :
    testChoices() {}

/* Inside Randomize Test Suite */
public void testRandomChoices() {...};
```

Although this approach does not use any annotations, besides the optional one that changes the way modules are defined as language constructs, the incremental compilation process is still needed to ensure that disabled tests are run at least once during testing.

3.5 The Process

The process used for all these approach is, in its essence, the same:

1. Identify all modules, their tests and dependencies.
2. Execute a *strongly connected* analysis of the dependency graph transforming it into a DAG of modules.
3. Execute a *topological sort* to determine in which order the components must be compiled.
4. For each component:
   (a) Compile it together with the previously tested components.
   (b) Execute the tests defined for the features provided by this component.
   (c) Execute the tests defined by previous components.

When a test fails in step 4b, it means that the module being added to the system has an error. This error can either be caused by a test not working properly, an error in the code of this module, or even a problem related to errors in the previously compiled modules that was not detected by the implemented tests.

When a test fails in step 4c, it means we have encountered an interaction between the code of the module being tested and one of the modules previously compiled. Depending on the selected approach, the information given to the developer can be different. If using the *Concern-Test*, it should be possible to pinpoint the concern that is being interfered with. The other approaches would only reveal the test being broken.

4. Implementation

During the course of the work, two different plugins were developed: *DrUID* and *Aida*. Both are based on the usage of annotations throughout the code that contain information about which interferences are expected. These tools were implemented as *Eclipse* plugins.

AspectJ was chosen as the target language for several reasons. First, it is one of the most used aspect-oriented languages. Secondly, as it is Java based, it can be used with *Eclipse*, an IDE where plugin development is straightforward. With *Eclipse* we also get two other important benefits, JDT and AJDT , tools for, respectively, the *Java* and *AspectJ* languages that allow access to the source code abstract syntax tree. Although the implementation is *AspectJ* oriented, the technique we proposed is apliable to other aspect-oriented languages following the same principles.

4.1 DrUID

DrUID (UID as in Unexpected Interference Detection) [11, 15] was the first attempt at creating a plugin to help developers follow the methodology being explored throughout this

paper. In order to accomplish this, the plugin allows developers to define several characteristics about system artifacts using Java annotations.

Several aids have been implemented to guide the developer in this process in the form of Eclipse *quick fixes* and *quick assists*. Each time a file is saved in Eclipse, the annotations are inspected and any errors are reported. Besides that, a dependency graph is created and shown in a graphical form that allows the developer to navigate through the code following the dependencies between artifacts.

4.2 Aida

Aida [12] is an evolution of the DrUID tool, built from scratch, having the main objective of removing most of the burden put on the developer to annotate his code. It also has a bigger focus on the testing process. In this tool, we started by removing the notion of annotating features manually. We did this by considering each test as a feature. This means that the developer only needs to create test cases for each individual behavior. Obviously, this also removed the need to specify which test case tests what feature.

Using code inspection, we were also able to remove the need of specifying the dependencies between features. At the cost of losing some of the details of the dependency graph used in DrUID, with Aida we rely only on the dependencies between units. In the end, we were down to only two types of annotations:

- **@TestFor** Used to indicate which unit each test is testing.
- **@ReplacesTest** Used to indicate that a test replaces another test. It also indicates that if the unit the test is related to is present in the system, then the replaced test does not have to be run.

Units are defined as being contained inside Java packages by default. A third optional annotation (**@Unit**) can be used to alter this behavior. The dependencies between units are automatically calculated by using the information provided by the JDT and AJDT Eclipse plugins.

With the dependency graph calculated, the test process is very similar to that of *DrUID*. We start by extracting the dependency graph from the source code, then we order the units by sorting them topologically and test them adding each unit incrementally to the system.

After running the complete set of tests, *Aida* is capable of reporting, both graphically and in text, eventual errors and interferences detected. This allows the developer to add **@ReplaceTest** annotations, when an interaction is expected, or correct his code if the interaction was unexpected.

4.3 Current Issues

There are still some issues with the implementation of these tools. *Aida* has been a major step forward as it removes most of the burden of declaring the dependencies from the developer, but there are still a couple of issues.

The first problem is that not all dependencies can be detected. At the moment, Aida is able to detect dependencies caused by: import declarations, method and constructor calls, type declarations and advices. These encompass most of the cases, but soft dependencies, like the ones created using reflection are not detected.

The second problem is that every time the project is tested, all the tests have to be run again. This problem is augmented by the fact that most tests are being run several times. This problem could be mitigated by doing some code analysis to figure which tests might have their results altered by the introduction of a new unit in the incremental compilation process.

5. Validation

To validate the approach we used it in several small sized projects and a medium sized one. The characteristics that we were looking for in a candidate project were that it had to be developed in *AspectJ*, it had to have few circular dependencies between modules and it had to have a test framework.

Unfortunately, all the existing open source projects we considered fail in one of these three aspects. For example, the two most used testbed projects for *AspectJ* are *AJHotDraw* [10] and *Health Watcher* [5]. The first of these has an architecture with a dependency graph so complicated that most of the code is part of a mass of 14 different packages that depend on each other forming a *strongly connected component*. The second one is a much cleaner project, but, unfortunately, there are no tests developed for it.

Having failed to elect a good and popular testbed where to run our testing process, we ended up developing our own testbed. A simple school information system [13] was implemented featuring personal information for students, teacher and administrators, course information, class schedules, infrastructure information and grading. Figure 4 shows the dependencies between the implemented packages.

After implementing the base packages, some packages containing aspects were added to the system:

Authentication Spectative aspect that adds a login and password attributes to the Person class. Offers methods to login and logoff as well as a way to verify who is logged in.

Attendance Adds a list of students that attended a certain lecture and methods to manage that list.

Security Invasive aspect that assures that the passwords are hashed using a secure hashing algorithm. For this, it advises the methods that set and verify passwords of the Authentication module.

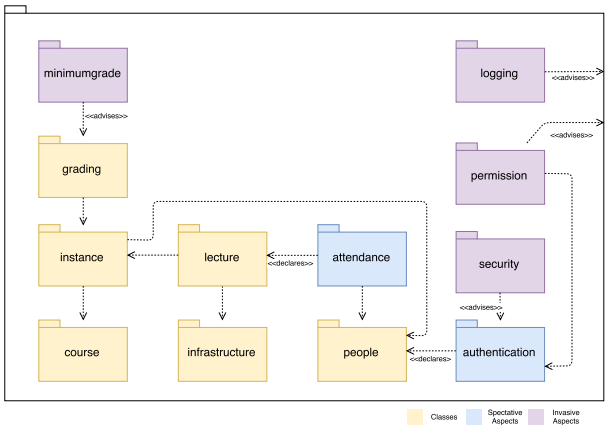

Figure 4. School Testbed Packages

Permission Invasive aspect that verifies that the logged in user has permissions to execute the command being executed. Advises almost every method in the code in order to do this verification.

Logging Spectative aspect that logs to a file important information. At the moment only the creation of new objects and login attempts are logged. To do this, it advises the object creation methods but does not change their behavior.

Minimum Grade Invasive aspect that adds the possibility of a course evaluation having a minimum grade that the student must attain to pass the course. Adds methods to define this minimum grade and advises the methods that calculate the student final grade.

Each one of the packages in the system was thoroughly tested. The total number of tests amounts to 55 with most of them belonging to the *Permission* package. This happens as this package crosscuts the entire application and modifies the behavior of almost all methods by adding a permission system. This makes it important to test if those methods are still working when the user has permission to use them, and also if access is denied when the user has no permission to use them.

By using *Aida*, interferences were easily spotted. Each time an invasive aspect was added, a test broke somewhere. In the rare event where that did not happen, it was due to an error in the implementation of the new aspect or a poorly written test. By using the technique described in this document, we were able to test all the packages of the system, in isolation, without compromising modularity.

After testing the complete system we tried to test smaller subsets of the system where some packages were not considered. We counted 77 different possible valid configurations with only some of the non-aspectual packages being used. If we add the other four invasive packages, in any possible combination, we get eight times more possibilities. Or a grand total of 616 configurations. We were able to test all of these, successfully, using *Aida* without having to add, remove or change any of the tests.

6. Related Work

Katz [7] proposed the use of regression testing and regression verification as tools that could help identifying harmful *aspects*. The idea behind this technique is to use regression testing as normally and then weave each *aspect* into the system and rerun all regression tests to see if they still pass. If an error is found, either the error is corrected or the failing tests have to be replaced by new ones specific for that particular *aspect*.

Ceccato [3] proposed a technique to establish which tests had to be rerun when incrementally adding *aspects* to a system.

Balzarotti [2] claims that the interaction detection problem can be solved by using a technique proposed in the early 80s, called program slicing. Although totally automatic, this technique does not account for intended interactions.

Havinga [6] proposed a method based on modeling programs as graphs and *aspect* introductions as graph transformation rules. Using these two models it is then possible to detect conflicts caused by *aspect* introductions. Both graphs, representing programs, and transformation rules, representing introductions, can be automatically generated from source code. Although interesting, this approach suffers the same problem of other automatic approaches to this problem, as intentional interactions cannot be differentiated from unintentional ones.

Lagaisse [9] proposed an extension to the Design by Contract paradigm by allowing *aspects* to define what they expect of the system and how they will change it. This will allow the detection of interactions by other *aspects* that were weaved before, as well as the detection of interactions by *aspects* that are bounded to be weaved later in the process.

It has been noticed by Kienzle [8] that *aspects* can be defined as entities that require services from a system, provide new services to that same system and removes others. If there is some way of explicitly describing what services are required by each *aspect* it would be possible to detect interactions (for example, an *aspect* that removes a service needed by another *aspect*) and to choose better weaving orders.

A state-based testing method for aspect-oriented software has been developed by Silveira [16]. According to the authors, this method provides class–aspect and aspect–aspect faults detecting capabilities.

Assunção [1] explored different ways to determine the order for integration and testing of aspects and classes. Two different strategies, incremental and combined, for integration testing were evaluated.

7. Conclusions

In this paper, we identified a problem that makes it hard to use unit testing in conjunction with aspect-oriented code. The problem is that unit tests in AOP systems must always test the system after the advises from any modules containing aspects have been applied. If these aspects are invasive, then the tests are not testing the unit in isolation and they stop being unit tests.

The solution we proposed is based on having tests, that test modules that contain invasive aspects, annotated in such a way that they announce which tests test the functionality being modified by those aspects. Having these annotations in place would allow a testing technique based on incremental compilation that could test units in lower layers of the software, using their own unit tests, separately from invasive aspects from higher layers. We argued that this is similar to what *stubs* and *mocks* contributed to object-oriented unit testing.

We do not argue that the proposed solution is usable in every situation, but we have shown that it can be used in several different scenarios. We envision it being used in software houses that have a large repository of modules that can be combined in different ways in order to compose different software solutions. Anyone that has tried to create such a system knows that crosscutting concerns are a big issue.

8. Acknowledgements

We would like to thank FCT for the support provided through scholarship SFRH/BD/32730/2006.

References

[1] W. K. G. Assunção, T. E. Colanzi, S. R. Vergilio, and A. T. Ramirez Pozo. Evaluating different strategies for integration testing of aspect-oriented programs. *Journal of the Brazilian Computer Society*, 20(1):9, 2014. ISSN 0104-6500. . URL `http://bit.ly/1HkTigI`.

[2] D. Balzarotti and M. Monga. Using program slicing to analyze aspect-oriented composition, 2004.

[3] M. Ceccato, P. Tonella, and F. Ricca. Is AOP Code Easier to Test than OOP Code? In *Workshop on Testing Aspect-Oriented Programs, International Conference on Aspect-Oriented Software Development*, Chicago, Illinois, Mar. 2005.

[4] M. Fowler. Mocks aren't stubs. Online article at martinfowler.com, January 2007. URL `http://bit.ly/18BPLE1`.

[5] P. Greenwood, A. F. Garcia, T. Bartolomei, S. Soares, P. Borba, and A. Rashid. On the design of an end-to-end aosd testbed for software stability. In *Proceedings of the 1st International Workshop on Assessment of Aspect-Oriented Technologies (ASAT. 07), Vancouver, Canada*. Citeseer, 2007.

[6] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *Aosd '07: proceedings of the 6th international conference on aspect-oriented software development*, pages 85–95, New York, NY, USA, 2007. ACM Press. ISBN 1-59593-615-7. .

[7] S. Katz. Diagnosis of harmful aspects using regression verification, 2004.

[8] J. Kienzle, Y. Yu, and J. Xiong. On composition and reuse of aspects. In *Software engineering properties of languages for aspect technologies*, 2003.

[9] B. Lagaisse, W. Joosen, and B. De Win. Managing semantic interference with aspect integration contracts. In *Software engineering properties of languages and aspect technologies*, 2004.

[10] L. M. Marius Marin and A. van Deursen. An integrated crosscutting concern migration strategy and its application to jhotdraw. Technical report, Delft University of Technology Software Engineering Research Group, 2007.

[11] A. Restivo. DrUID: Unexpected interactions detection, 2009. URL `https://github.com/arestivo/druid`.

[12] A. Restivo. Aida: Automatic interference detection for aspectj, 2010. URL `https://github.com/arestivo/aida`.

[13] A. Restivo. School-aspectj-testbed, 2014. URL `http://bit.ly/1NWM9Tl`.

[14] A. Restivo and A. Aguiar. Disciplined composition of aspects using tests. In *Proceedings of the 2008 AOSD Workshop on Linking Aspect Technology and Evolution*, LATE '08, pages 8:1–8:5, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-147-7. . URL `http://bit.ly/1QaMllo`.

[15] A. Restivo and A. Aguiar. DrUID – unexpected interactions detection. Demonstration at the Aspect Oriented Software Development Conference (AOSD'09), 2009.

[16] F. F. Silveira, A. M. da Cunha, and M. L. Lisbôa. A state-based testing method for detecting aspect composition faults. In B. Murgante, S. Misra, A. Rocha, C. Torre, J. Rocha, M. Falcão, D. Taniar, B. Apduhan, and O. Gervasi, editors, *Computational Science and Its Applications – ICCSA 2014*, volume 8583 of *Lecture Notes in Computer Science*, pages 418–433. Springer International Publishing, 2014. ISBN 978-3-319-09155-6. . URL `http://bit.ly/1QaMcyj`.