

**FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO**



**FEUP**

# **Catalogue of Unexpected Interactions Between Aspects**

**Mário Rui Cabral Aguiar**

Thesis Report

Master in Informatics and Computing Engineering

Supervisor: Ademar Manuel Teixeira de Aguiar (Ph.D.)

Co-Supervisor: André Monteiro de Oliveira Restivo (M.Sc.)

29<sup>th</sup> June, 2009



# **Catalogue of Unexpected Interactions Between Aspects**

**Mário Rui Cabral Aguiar**

Thesis Report

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Eugénio da Costa Oliveira (Ph.D.)

---

External Examiner: Fernando Brito e Abreu (Ph.D.)

Internal Examiner: Ademar Manuel Teixeira de Aguiar (Ph.D.)

17<sup>st</sup> July, 2009



# Abstract

This dissertation focuses on problems related with Aspect-Oriented Programming, in concrete the unexpected interactions between aspects. These unexpected interactions are an important issue of AOP software evolution, they may compromise the entire development of a software project. It is important to address these issues by studying and documenting the origin of its problems.

It is important to make clear at this point what these interactions are. We state in this dissertation that unexpected aspect interactions are the result of interferences between two or more aspects. Interference is the action of one aspect disrupting or changing the behavior of another fine working aspect. A conflict is the resulting problem that arises from the interference that an aspect does.

By studying other author's classifications systems of aspects, interactions and techniques for the detection of these unwanted interactions, we started tracing the main causes of these issues. With this information it is possible to document each different interaction that can occur between aspects.

A catalogue describing these conflicts was produced to be useful for future work in this field of computer science. The catalogue describes the conflicts, by naming each one and gives a small description about them. It introduces a context where the conflict may appear and defines the category of the conflict according to its origin. A small example is also given to understand how it can appear in a program source code, and, finally, it suggests how one can fix the conflict or even prevent it.

Currently, there is not yet satisfactory tool support for detecting these conflicts that appear on AOP applications. A conflict detector was another objective of this thesis. A small experimental prototype was implemented and it can detect three of the eight conflicts documented in the catalogue.



# Resumo

Esta dissertação foca-se nos problemas relacionados com Aspect-Oriented Programming em concreto as interacções inesperadas que ocorrem entre aspectos. Estas interacções inesperadas são o principal obstáculo na evolução da AOP, estas podem comprometer o desenvolvimento de um projecto de software. É importante encontrar soluções para esta questão e documentar quais as possíveis origens dos problemas.

É necessário desde já esclarecer o que são estas interacções. Esta tese defende que uma interacção inesperada é o resultado de interferências entre dois ou mais aspectos. Uma interferência é a acção que ocorre quando um aspecto interrompe ou altera o comportamento de um outro aspecto que estava funcionando correctamente. O conflito é o resultado que ocorre da interferência que um aspecto faz.

Através do estudo do trabalho de outros autores, sobre sistemas de classificação de interacções e de técnicas de detecção destas interacções é possível descobrir quais são as principais causas destes problemas. Com esta informação pode-se documentar cada interacção que pode ocorrer entre aspectos. Foi produzido um catálogo descrevendo estes conflitos que tem como objectivo ajudar outros investigadores que trabalhem nesta área de informática.

O catálogo descreve os conflitos de acordo com os seus nomes e apresenta uma pequena descrição sobre cada conflito. Também introduz o contexto no qual o conflito pode aparecer e categoriza os conflitos de acordo com a sua origem. É também apresentado um pequeno exemplo para compreender como o conflito poderá aparecer no código e finalmente é sugerido como se pode reparar ou até mesmo prevenir o conflito.

Actualmente, não há um grande suporte de ferramentas para detecção destes conflitos que aparecem em aplicações AOP. Um detector de conflitos era outro objectivo da tese. Um pequeno protótipo foi implementado e consegue detectar três dos oito conflitos documentados no catálogo.





# Acknowledgements

Finally this long journey comes to an end. It was not an easy task to perform as this is one of the hardest courses in the faculty. Of course I couldn't do it without the help of the many people that received me here.

My first thanks go to my friends that shared with me the same problems and sleepless nights here at the Faculty of Engineering.

Secondly I wish to thank my two thesis supervisors Ademar Aguiar and André Restivo for being always available, during the entire semester.

Finally the most important people who helped me a lot were my mother Eduarda Cabral, my father Carlos Aguiar and my little sister Filipa Aguiar they were always there to support me in the moments were I needed them most.

Mário Rui Cabral Aguiar



*“Because a thing seems difficult for you,  
do not think it impossible for anyone to accomplish. ”*

Marcus Aurelius



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context . . . . .	1
1.2	Motivation and Objectives . . . . .	2
1.3	Structure of the dissertation . . . . .	3
<b>2</b>	<b>Aspect-Oriented Programming</b>	<b>5</b>
2.1	Introduction . . . . .	5
2.2	Separation of Concerns . . . . .	5
2.3	Object-Oriented Programming . . . . .	6
2.4	Java . . . . .	6
2.4.1	Example Shopping List . . . . .	7
2.5	Aspect Oriented Programming . . . . .	9
2.5.1	Code Scattering and Tangling . . . . .	10
2.5.2	Quantification and Obliviousness . . . . .	10
2.6	AspectJ . . . . .	11
2.6.1	Joinpoints . . . . .	11
2.6.2	Pointcuts . . . . .	12
2.6.3	Advice . . . . .	13
2.6.4	Inter-Type Declarations and Reflection . . . . .	14
2.6.5	Aspects . . . . .	14
2.7	Key Issues of AOP . . . . .	16
<b>3</b>	<b>Aspect Interactions and Conflicts</b>	<b>17</b>
3.1	Introduction . . . . .	17
3.2	Aspect Interactions Classifications . . . . .	17
3.2.1	Katz Classification of Aspects . . . . .	17
3.2.2	Munoz, Barais and Baudry Classification . . . . .	18
3.2.3	Kienzle Classification . . . . .	19
3.2.4	Frans Sanen et al. Classification . . . . .	19
3.2.5	Rinard, Sălcianu and Bugrara Classification . . . . .	20
3.2.6	Clifton and Leavens Classification . . . . .	21
3.3	Conflicts Detection . . . . .	21
3.3.1	Violation of Language Rules . . . . .	25
3.3.2	Introduction has unintended effects . . . . .	25
3.3.3	Ambiguous aspect specification . . . . .	26
3.4	Program Slicing . . . . .	28
3.5	Regression Testing . . . . .	29

## CONTENTS

3.6	Aspect Interactions Charts . . . . .	29
3.7	Summary . . . . .	29
<b>4</b>	<b>Catalogue of Conflicts</b>	<b>31</b>
4.1	Catalogue Structure . . . . .	31
4.2	Conflicts Categories . . . . .	32
4.3	Conflicts Documented . . . . .	32
4.4	Conflict Fragile Pointcuts . . . . .	34
4.5	Conflict Shared Joinpoint . . . . .	36
4.6	Conflict Shared Variables . . . . .	38
4.7	Conflict Dependent Aspects . . . . .	41
4.8	Conflict Advice Interference . . . . .	43
4.9	Conflict Incorrect Method Call . . . . .	45
4.10	Conflict Dependent Annotations . . . . .	47
4.11	Conflict Incorrect Aspect Precedence . . . . .	49
4.12	Summary . . . . .	52
<b>5</b>	<b>Prototype</b>	<b>53</b>
5.1	Eclipse Architecture . . . . .	53
5.2	Conflict Detector Plug-in . . . . .	54
5.2.1	Design and Implementation . . . . .	55
5.3	Plug-in Demonstration . . . . .	58
5.4	Execution Results . . . . .	60
5.5	Summary . . . . .	61
<b>6</b>	<b>Conclusions and Future Work</b>	<b>63</b>
6.1	Objectives Satisfaction . . . . .	63
6.1.1	Catalogue . . . . .	64
6.1.2	Prototype . . . . .	64
6.2	Future Work . . . . .	64
6.3	Concluding Remarks . . . . .	65
	<b>References</b>	<b>71</b>

# List of Figures

2.1	Decomposition of requirements into a set of concerns. Extracted from [Lad03]. . . . .	6
2.2	Code scattering in different classes. The same concern (grey rectangles) is defined in several places. . . . .	10
2.3	Code tangling. This software module contains several concerns. Extracted from [Lad03]. . . . .	10
2.4	Aspect weaver. . . . .	11
2.5	Joinpoint inside an Object. . . . .	12
3.1	Analysis of a semantic conflict. Adapted from [DBA05]. . . . .	22
3.2	Detection of shared join points. Adapted from [ARS09]. . . . .	24
4.1	Two aspects are superimposed in the same joinpoint. . . . .	36
4.2	Transitive interaction, the value of x is passed to y . . . . .	40
4.3	Advice around replaces the method execution . . . . .	43
4.4	Introduction of a new method . . . . .	45
4.5	Conflict Resolution Categories. Extracted from [CPSM07]. . . . .	51
5.1	Eclipse extensibility, products can extend other products functionalities. Adapted from [iS09]. . . . .	54
5.2	Eclipse SDK layers. Adapted from [iS09]. . . . .	55
5.3	Build project on menu bar. . . . .	58
5.4	Build project on popup menu. . . . .	58
5.5	Open views menu. . . . .	59
5.6	Open conflict detector view. . . . .	59
5.7	Conflict detector view. . . . .	60





# List of Listings

2.1	Class ShoppingList . . . . .	8
2.2	Class Logger . . . . .	8
2.3	New ShoppingList Class . . . . .	9
2.4	Aspect Logging . . . . .	15
2.5	Final ShoppingList Class . . . . .	15
3.1	Violation of Language Rules . . . . .	25
3.2	Introduction has unintended effects . . . . .	26
3.3	Ambiguous aspect specification . . . . .	26
3.4	Class Product . . . . .	27
3.5	Declaration of a pointcut . . . . .	27
3.6	Introduction of annotations . . . . .	28
4.1	Pointcut payColaborators . . . . .	34
4.2	Define precedence between two aspects . . . . .	37
4.3	Class Object and aspects DoSomething and DoSomethingElse . . . . .	38
4.4	Solution . . . . .	39
4.5	Aspect checkRaise . . . . .	41
4.6	Aspect DBEmployeePersistence . . . . .	42
4.7	Annotation @TransientClass . . . . .	47
4.8	Annotation @TransientField . . . . .	47
4.9	Example . . . . .	49
4.10	Association LogAcc . . . . .	50
4.11	Example of a solution . . . . .	50
5.1	Method CompilationUnit . . . . .	55
5.2	ClassVisitor . . . . .	56
5.3	AspectVisitor . . . . .	56



# Abbreviations

AIC	Aspect Interactions Charts
AJDT	AspectJ Development Tools
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
API	Application Programming Interface
IDE	Integrated Development Environment
JDT	Java Development Tools
LSC	Live Sequence Charts
MVC	Model View Controller
PDE	Plug-in Development Environment
OOP	Object-Oriented Programming
RCP	Rich Client Platform
SDK	Software Development Kit
SoC	Separation of Concerns



# Chapter 1

## Introduction

The most important stages in software life cycle are the maintenance and the evolution, namely the addition of new modules that weren't anticipated in previous stages.

Because Aspect-Oriented Programming (AOP) can be used to develop software in a better modularized way, many researchers are studying AOP as the ideal paradigm for this type of requirement. However there are some issues with this paradigm, unexpected interactions between aspects, that are still an obstacle in the applicability of AOP. These situations usually occur in big software projects where many developers and often new team members are involved in the process.

This dissertation addresses the issues of aspects interactions and the conflicts that arise from these interactions. A catalogue containing a description about these conflicts was produced as well as a small prototype to detect these conflicts in a software project.

### 1.1 Context

This master thesis addresses the feature interaction problem [AG95] in software development. A feature can be a component that adds a new functionality to the core of a software application. Generally features are added in different stages in the lifecycle of software development and usually this is done by different developers. The problem with feature interaction is that some features can compromise the correct behavior of other features. The behavior of a feature can be defined by the flow of its execution and the output for a given input. This means that the unwanted interaction changes the execution flow and output of the interacting features for a given input. These unwanted and unexpected interactions can really bring some serious issues on the development and maintenance of software.

It is important to develop tools that can in fact detect and manage the interactions between the different features of a software application. The problem with managing feature interactions is how to differentiate the good and harmful ones and how to enable the desirable interactions. There are some techniques that can help to avoid or diminish unexpected feature interactions. Some of these techniques consist in the specification and development of system architectures with well defined interfaces that are used in the implementation of services or by establishing guidelines that are incorporated into the creation of services.

The same problems can occur in AOP, with unexpected interactions occurring when adding new modules to a system. We define here some of the most important concepts which will be discussed through this thesis.

The interference is the concept that will be recurrent. We can define interference as the action that one module does when disturbs the correct behavior of another module. The result of this interference is a conflict which can be of various types.

These unwanted interactions can bring some serious delay in the development of application. It is imperative to detect and solve or prevent these interactions to occur.

## 1.2 Motivation and Objectives

AOP is a programming paradigm that aims to improve modularity by allowing crosscutting concerns to be developed as single units of modularity. One of the major drawbacks of AOP is that unexpected interactions between aspects are more likely to occur than in normal Object Oriented Programming (OOP).

Unexpected interactions are a key issue with the evolution of AOP, which can seriously compromise the development process. It is important to address this problem by studying and documenting the origins of its problems.

The first objective of the thesis is to study AOP, this will be done by practicing programming exercises with the AspectJ language and understanding the architecture of AOP applications. The second objective is to study AOP interaction detection and to find out which classification systems researchers on this field use to classify interactions between aspects. The third and main objective of this thesis is the writing of a catalogue to document how these unexpected interactions occur and how they can be detected and dealt with, or even which preventive measure should be applied in order to avoid them. The catalogue will contain a description and several examples of each type of interaction. This will be useful for future programmers when studying this new programming paradigm. The last objective is the implementation of a small prototype to detect unwanted interactions between aspects.

### **1.3 Structure of the dissertation**

This dissertation is divided in six chapters. The first chapter introduces the context of this thesis as well as its motivation and objectives. Chapter 2 introduces fundamental concepts about this field, compares OOP and AOP and explains the main advantages of AOP. Chapter 3 describes the several classifications of aspect interactions that authors have proposed and interaction detection techniques. Chapter 4 contains the catalogue with the description of each conflict. Chapter 5 explains the implementation of the prototype to detect conflicts between aspects. Finally chapter 6 presents the final conclusions about the thesis as well as some proposals for improvements on the catalogue and prototype.

## Introduction



## Chapter 2

# Aspect-Oriented Programming

### 2.1 Introduction

This chapter introduces the basics concepts of Aspect-Oriented Programming (AOP) and the main issues that led to the development of this paradigm. First we introduce one of the major problems that AOP addresses, the separation of concerns (SoC), it is explained and then a comparison is made between Object-Oriented Programming (OOP) and AOP, using the refactoring of a small example.

In the end of this section the Java and AspectJ languages are also compared in order to understand which are the new mechanisms that AspectJ has to offer to software programmers.

### 2.2 Separation of Concerns

Before introducing the concepts of aspect-oriented software development (AOSD), one must have to consider the principle of separation of concerns (SoC), which is one of the most important principles in software engineering. This principle considers that a supposed engineering problem involves several kinds of concerns, which can be identified and separated to handle with the complexity and to ensure the necessary software quality factors such as robustness, adaptability, maintainability and reusability. Figure 2.1 illustrates the process of decomposing the requirements into a set of concerns.

Although there is a consensus that is important to design software systems having in attention the principle of separation of concerns, many ways to implement this feature have been created. In OOP the separated concerns are modeled as objects and classes. For example, the design pattern model-view-controller (MVC) separates content from presentation and data-processing from the content. Procedural programming separate

concerns into procedures. Service-oriented design separates concerns into services. AOP separates concerns into its basic unit the aspect.

## 2.3 Object-Oriented Programming

OOP defines objects as the basic building block of a computer program. These "objects" are introduced as a mean to define data or information and to represent interactions between the several components of a program. OOP languages offer some powerful techniques like, information hiding (encapsulation), data abstraction, modularity, polymorphism, and inheritance.

The definition and rules established in object oriented-programming paradigm make available some very important benefits: the mechanism of class inheritance makes it possible to create subclasses of data objects, which inherit all of the parent class characteristics. This technique reduces development time and ensures a more robust coding. Security is one of the most important properties of a programming language. Since the class, only represents the information that it needs to be concerned with when it runs it will never accidentally access other program data. This is a great benefit since the system security is guaranteed and data will never be corrupted. The classes are reusable, not only to the application for which they are created but also by other object-oriented projects, and for this they are extremely useful for distributed applications to use in computer networks. Finally one of the major characteristic is that it enables us to create new data types, that were not defined previously in the programming language, as they usually are user-defined.

## 2.4 Java

Java is a OOP language. Unlike other programming languages Java isn't compiled into native code, instead it is compiled into bytecode, which is later interpreted by a Java Vir-

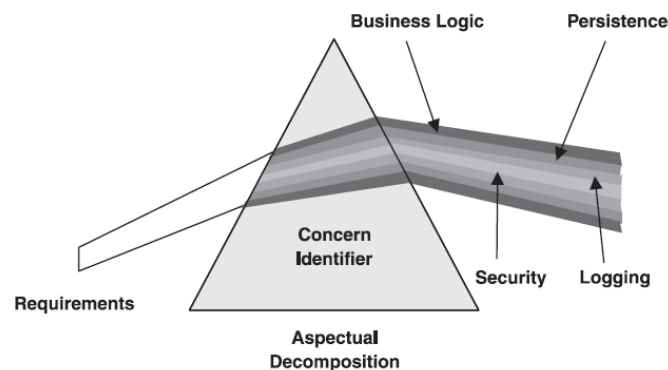


Figure 2.1: Decomposition of requirements into a set of concerns. Extracted from [Lad03].

tual Machine, making Java a truly portable programming language. Java is distributed and is designed to supply powerful libraries with networking ability helping in the development of distributed software

It is a multithread programming language that enables to create a program with different threads at the same time. Java is also robust and secure due to its automatic garbage collector and its security manager that allows determining the accessibility options of a class.

Being an OOP language, Java offers some more technical features expected to this kind of programming languages. The Java inheritance allows classes to inherit methods and behaviors from its parent class; Java does not support multiple inheritance i.e. a class can't have more than one class parent. One of the resources that inheritance provides is polymorphism, i.e. the ability that a method has to respond in different ways. This is called method overriding and it happens when a sub class redefines a method that was already declared on its parent class, with the same parameters and return type. A similar mechanism that Java offers is method overloading. It enables a class to define several with the same name, but have different numbers and type of parameters and may also have a different return type.

One of the most interesting features of Java is reflection. It enables one to examine or modify the runtime behavior of applications running in the Java virtual machine. Although reflection may be a powerful tool, we must have in consideration that some issues may appear if it is not used correctly. We must have in consideration the following concerns when using Java reflection:

- Performance overhead since reflection need to resolve types dynamically.
- Security restrictions. Java has a security manager that controls accesses from classes, but this might not be possible with reflection because it requires a runtime permission that may not be present when running under a security manager.
- Exposure of Internals. As reflection enables operations that were not legal in a non reflective code, such as accessing private fields and methods, this can lead to unexpected side effects.

### 2.4.1 Example Shopping List

Despite OOP introduced some of the most useful concepts for software development, there are some issues with this paradigm.

Consider the following requirements for a software project. A client needs an application to register products added to a shopping list.

For this problem there are two main concern: one is to register the total whenever a new product is added, and the second is log the changes to the total of the shopping list.

We can model the shopping list in the class described in listing 2.1, below

Listing 2.1: Class ShoppingList

```
1 public class ShoppingList {
2     private double total;
3
4     ShoppingList() {
5         total=0;
6     }
7     public void addProduct(Product p){
8         //business logics add a new product to the list
9     }
10    public double getTotal(){
11        return total;
12    }
13    public void setTotal(double t){
14        total=t;
15    }
16 }
```

According to the first requirement, not everything is implemented, but only the essential for this demonstration. So far the properties of OOP are maintained, however the second requirement is not implemented yet and will be demonstrated that it will break some of the SoC properties.

The next requirement is about logging all the changes to the total in the shopping list. For that we, must implement a class that logs the changing event. In listing 2.2 we can see the logging code.

Listing 2.2: Class Logger

```
1 public class Logger {
2
3     Logger() {
4         //logger constructor
5     }
6
7     void writeLog(String message) {
8         //log the new value of total
9     }
10 }
```

Again the fundamentals of OOP remain intact as the class *Logger* only implements its own concern. To use the *Logger* class we must initialize a logger object and then use the *writeLog* method to log the changes to the total variable.

To log the changes of the total variable the class *ShoppingList*, needs to be updated with the new functionalities.

Listing 2.3: New ShoppingList Class

```
1 public class ShoppingList {
2
3     private double total;
4     Logger logger;
5
6     ShoppingList(){
7         total=0;
8         logger = new Logger();
9     }
10    public void addProduct(Product p){
11        //business logics add a new product to the list
12        logger.writeLog("total has been updated to "+ total);
13    }
14    public double getTotal(){
15        return total;
16    }
17    public void setTotal(double t){
18        total=t;
19    }
20 }
```

Now with the inclusion of the logger class code to the *ShoppingList*, the two concerns are now implemented in the same module making it less readable for future analysis.

Like any other technology, computer programming is constantly evolving according to the needs of programmers. With all the new programming paradigms, some problems are solved but unfortunately other issues appear. OOP isn't satisfactory enough to implement some of the design decisions that are desirable for the program. Traditional OOP languages and modularization mechanisms suffer from a limitation called the Tyranny of the Dominant Decomposition[DD02]. This limitation refers to the fact that the program can be modularized in only one way at a time, and the many kinds of concerns that do not align with that modularization end up scattered across many modules and tangled with one another. This "tangled" code is very hard to develop and to maintain leading to great costs if some new functionality is needed to implement.

## 2.5 Aspect Oriented Programming

AOP is an emerging paradigm in the computer programming science. It can provide the modularity and separation of crosscutting concerns that others programming paradigms

## Aspect-Oriented Programming

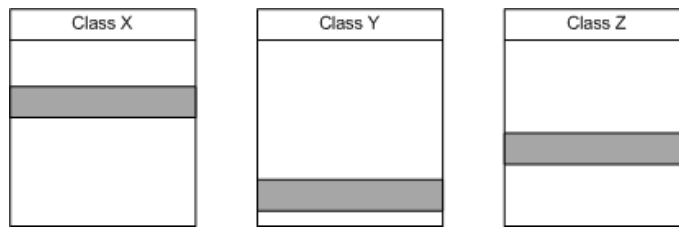


Figure 2.2: Code scattering in different classes. The same concern (grey rectangles) is defined in several places.

cannot achieve. With these new features that AOP brings to software development an easier solution to develop and maintain software code.

### 2.5.1 Code Scattering and Tangling

OOP has some flaws which can bring some issues that can be a problem to software evolution. Code scattering and tangling are the main problems. Code scattering means that the same piece of code is scattered across different modules of software system Figure 2.2 depicts this situation. Code tangling Figure 2.3 is when one module contains a mix up of code that addresses several concerns. These two issues can increase software complexity, evolution and maintenance cost.

### 2.5.2 Quantification and Obliviousness

AOP comes to aid software developers to attend those two previous flaws. Robert Filman [FF00] [Fil01] states that an AOP application must have two important properties, quantification and obliviousness. Quantification is the ability to write small piece of code in one place that can affect other places of a programming system. In other words Filman says that quantification must support the expression of statements of the form:

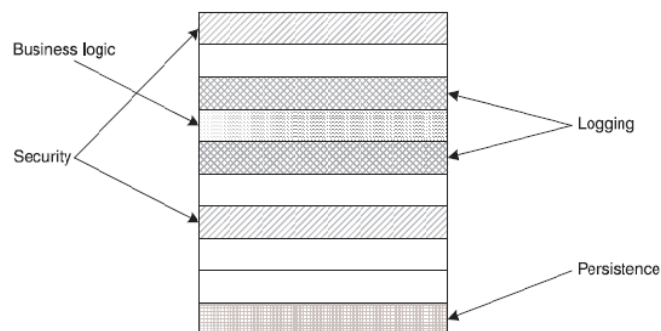


Figure 2.3: Code tangling. This software module contains several concerns. Extracted from [Lad03].

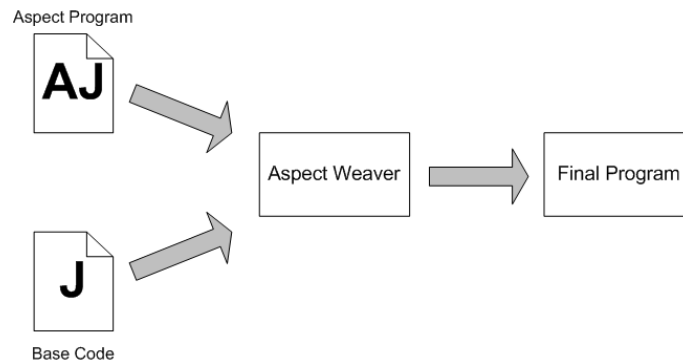


Figure 2.4: Aspect weaver.

*In programs  $P$ , whenever condition  $C$  arises, perform action  $A$ .*

Quantification involves the reuse of the functionality defined only in action  $A$  without requiring the programmer to write in all of the places in program  $P$  where condition  $C$  arises.

Obliviousness is achieved when software modules are not aware that other software module is collaborating with it. It also guarantees that the code for each concern is modularized making it easier to read and understand.

To implement an AOP application, first we start by coding the requisites that are needed for the project in question in an object-oriented programming language and then the programmer should deal with the separation of concerns by implementing the aspects. In the end, the code from the classes and the code from the aspects are weaved together into a final program, Figure 2.4 illustrates this final step.

## 2.6 AspectJ

AspectJ is the most commonly used programming language, when one uses the AOP paradigm, because of its simplicity and usability. AspectJ is an extension to the Java programming language, thus making its learning easier, for experienced programmers.

The primary element of AspectJ is the aspect. Aspects, looks just like a class, it has methods, variables, access restrictions. AspectJ, introduces the AOP elements that compose an aspect, joinpoints, pointcuts and advices [KM05][KHH<sup>+</sup>01][SB06].

### 2.6.1 Joinpoints

Joinpoints are identifiable at any point during the AspectJ implementation of a program. AspectJ allows different types of join points: entries and exits of methods, treatment of

exceptions, access to variables of objects, constructors, among others.

To better understand AspectJ joinpoints, one must consider the basic principles of the Java programming language. For instance consider the previous class *ShoppingList* and its method *setTotal(double)*. This piece of code states that whenever the method *setTotal* with a double argument is called on an object of the type *ShoppingList*, then its method body `{ total=t; }` is executed.

A pattern can be identified with this description. It says that when something happens, then something gets executed. In object-oriented applications, there are some "things that happen", that are determined by the language. These are the joinpoints of Java Figure 2.5, which can be method calls, executions of methods, instantiation of new objects, executions of constructors, references to object fields and handler executions.

### 2.6.2 Pointcuts

Pointcuts are program constructs that can select one or more joinpoints and collect context at these joinpoints. For example, a pointcut can pick up an joinpoint that is a call to a method and capture the target object on which the method was invoked as well as its arguments.

AspectJ offers a variety of pointcuts constructs, some of the most important are presented in the following list.

- Method execution - `execution(void setTotal(double));`
- Method call - `call(void setTotal(double));`
- Read access of a field - `get(double ShoppingList.total);`
- Write access of a field - `set(double ShoppingList.total);`
- Exception handling - `handler(IOException);`
- When the executed code belongs to the *ShoppinList* class - `within(ShoppinList);`

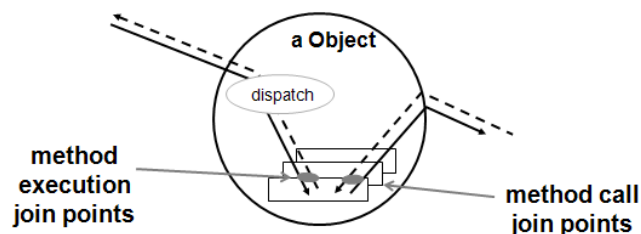


Figure 2.5: Joinpoint inside an Object.



- When the joinpoint is in the control flow of call to the method *setTotal -cflow(void setTotal(double))*;
- When the executed object is of the type *ShoppingList - this(ShoppingList)*;
- When the target object is of the type *ShoppingList - target(ShoppingList)*.

To select the joinpoints, AspectJ offers operators to compose the pointcuts.

- And operator selects a and b joinpoints - *a && b*;
- Or operator selects a or b joinpoints - *a | | b*;
- The not operator to exclude a joinpoint - *!a*.

Because pointcuts are built at a semantic level, AspectJ offers the use of wildcards as a means to construct pointcuts. This is very useful since one can select several joinpoints that have some different characteristics, such as return type, number of arguments, similar methods names etc.

- Selects all the joinpoints of any return type - *call(\* setTotal(double))*;
- All the joinpoints that start with the set word - *call(\* set\*)*;
- Any joinpoints with different number of arguments - *call(\* setTotal(..))*;

### 2.6.3 Advice

Joinpoints and pointcuts don't do anything by themselves it's the advices that really do the work in AOP. The advice is a bit of code to be executed at the junction selected for a pointcut. AspectJ has three different types of advices.

- *Before advice* runs as a joinpoint is reached, before the program proceeds with the join point. For example, a before advice on a method call join point runs before the actual method starts running, just after the arguments to the method call are evaluated.
- The *after advice* is opposite of the *before advice*; it runs just after the program proceeds with the joinpoint. There are different kinds of *after advice*, the *after returning advice* executes after the joinpoints returns something, the *after throwing* handles the throw of an exception and the plain *after advice* handles the two situations described here.
- The *around advice* is the most powerful, it has the ability to be executed before and after the joinpoint, and it also can even replace the execution of the joinpoint.

#### 2.6.4 Inter-Type Declarations and Reflection

AspectJ can provide some other mechanisms like introduction and reflection. Introductions or inter-type declarations are the possibility to add new methods and variables to classes. The inheritance from a class can be modified with introductions too. Aspects can modify classes to implement one or more interfaces or to make one class inherit methods from a parent class. This last example follows the Java inheritance rules, i.e. multiple inheritance isn't allowed in AspectJ.

The second mechanism, reflection, allows one to examine information about the execution of a pointcut. This reflective process is very useful in logging and debugging. Despite all these new functionalities that AspectJ offers, there are some conflicts that may arise between aspects. Since the main function of aspects is to interact with classes and others aspects, it is obvious at some point of a development of a new application that some aspects that were working previously with the expected behavior, they won't work anymore with the introduction of new aspects or others modifications to the code.

#### 2.6.5 Aspects

The more important element that AOP introduces is the aspect. Aspects are entities like Java classes, which are the basic block to implement aspect-oriented crosscutting concerns. The AspectJ compiler analyses the instructions defined in each aspect of the system and uses them to wove in with the Java code, modifying the behavior of the system.

These instructions that are in the aspect code are, of course, joinpoints, pointcuts, advices and member introductions.

We can see an aspect as a normal class, as it also has similar properties that a class has, like the ones listed below:

- Aspects can also have members and methods;
- Aspects can define access specifications;
- An aspect can be abstract;
- Aspects can extend classes and abstract aspects, as well as implement interfaces;
- Aspects can be defined inside classes and interfaces;
- An aspect cannot be instantiated;
- Aspects cannot inherit from concrete aspects;
- Aspects can be privileged.

Now that some of the basic concepts of aspect-oriented programming have been introduced we can implement the previous example of the *ShoppingList*, modularizing it according to its concerns.

Let's recall its previous requirements, the first one was to register the new total in *ShoppingList* and the second one was to log the action of adding a new product. The problem with the object-oriented approach was that the crosscutting concern of logging the change was introduced in the business logic of adding new product to the shopping list. With AspectJ it is possible to define the second concern in an independent aspect, that should be responsible for the logging operation. In the Logging aspect we can use a pointcut to capture the method which will add a new product. We can define a pointcut like the one in listing 2.4, that will crosscut the method *addProduct*, and then the after advice will execute the logging operations defined in the software requirements. Listing 2.5 shows the class *ShoppingList* without the concern of logging action thus making it a truly oblivious system, which doesn't know that the logging action will take place in another software module.

Listing 2.4: Aspect Logging

```
1 public aspect Logging{
2
3     Logger logger = new Logger ( ) ;
4
5     public pointcut change(ShoppinList l, double total):
6         execution(void addProduct ( Product)) && this(l) && args(total);
7
8     after(ShoppinList l, double total) : change(l,total){
9
10        logger . writeLog ( "total has been updated to" + total ) ;
11
12    }
13 }
```

Listing 2.5: Final ShoppingList Class

```
1 public class ShoppingList {
2     private double total;
3
4     ShoppingList(){
5         total=0;
6     }
7     public void addProduct(Product p){
8         //business logics add a new product to the list
9     }
10    public double getTotal(){
11        return total;
12    }
13    public void setTotal(double t){
```

```
14    total=t;  
15 }  
16 }
```

### 2.7 Key Issues of AOP

Despite the advantages that Aspect orientation has brought to the software development industry, it also has its own disadvantages, such as conflicts appearing in an unexpected way.

The most common conflicts in AOP tend to be semantic conflicts. This may happen when a pointcut isn't correctly specified, or it is shared by more than one aspect. One of the most recurring conflict is when the order of execution between aspects isn't the desirable by the programmer, this may appear when one adds a new aspect to the system and didn't notice that it would interfere in a harmful way with other aspects. Since the code to be executed is weaved in the end by the compiler, this can interfere too with the expected functionality of the system, one must have in attention when this happens. The introduction mechanism that AspectJ implements is a source of conflicts too, which may introduce naming conflicts. Consider that one introduces a method in an interface with the name *modifyItem*, inside an aspect, in other aspect someone introduces a method with the same name in a class that implements the previous interface. This will lead to a conflict because the compiler doesn't know which method to call.

These conflicts will be explained in detail in the next chapter.

## Chapter 3

# Aspect Interactions and Conflicts

### 3.1 Introduction

In this chapter the state of aspect interactions and conflicts is reviewed and therefore some of the most important work is presented here in order to understand the respective key issues. The first section starts by introducing some proposals by several authors to classify the different interactions aspects or its internal mechanisms.

We can now define one of the main issues discussed on this thesis, aspect interference. We can define aspect interference as the main consequence that occurs when one tries to weave two independently working aspects into a system and as result it will happen an unexpected behavior. This is of course the real definition of conflict when two opposing parts try to accomplish its own objectives.

Studies have been made in order to detect these conflicts, so in the second section of this chapter some most relevant techniques will be described.

### 3.2 Aspect Interactions Classifications

In this section the different types of interactions aspects are presented, to better understand the conflicts that may appear in aspect oriented programming. The relations between aspects and its components are identified according to several authors that categorize aspect interactions within their own ideas and beliefs.

#### 3.2.1 Katz Classification of Aspects

The first classification for aspect interaction was proposed by Katz [[Kat04](#)] he separates aspects in three distinct categories:

- **Spectative Aspects** only observe the line of execution, gathering information about the variables of the system. The only variables that may change are its local variables;
- **Regulative Aspects** have the same functions as the spectative aspect, but also can change the control flow of the execution i.e. can restrict the conditions for activating a method call of the original system;
- **Invasive Aspects** can change the variables or the state of the original system.

### 3.2.2 Munoz, Barais and Baudry Classification

Freddy Munoz, Olivier Barais and Benoit Baudry [MBB08] classify aspects interactions from three points of view. First they classify advices, according to its behavioral interaction between them and the methods that are crosscut by them. In this category there are five types of interaction:

- **Augmentation:** The crosscut method is always executed, and the advice augments the actions of the method with new actions that won't interfere with the original behavior;
- **Replacement:** After the crosscut, the method will never be executed. Instead the advice replaces the behavior of the method with a new behavior;
- **Conditional replacement:** Following the crosscut of the method, it is not always executed. The advice checks for a condition and it will potentially replace the original behavior with a new one.
- **Multiple:** After the crosscut, the method is executed multiple times. The advice will invoke two or more times the method and it may change the intended behavior;
- **Crossing:** Following the crosscutting of the method, the advice will invoke a method that it does not cross. This means that the advice has a dependency to the class that has the invoked method.

Next they classify, advice interaction according to the way they access the objects fields and method arguments.

- **Write:** An advice can write an object field. This access may break the protection that was declared on that particular field and can change the behavior of the program;

- **Read:** When an advice reads a protected object field, this may expose critical data;
- **Argument passing:** After crosscut, the advice changes the parameters of the method it crosscuts and then it invokes the method. The method is always executed at least once.

The last category presented classifies aspects according to the modification that they introduce to the structure of a class.

- **Hierarchy:** The hierarchy of a class is modified. A good example for this would be, that an aspect adds a new parent interface to a class.
- **Field addition:** The aspect adds new fields to a class.
- **Operation addition:** The aspects add new methods to a class.

### 3.2.3 Kienzle Classification

Kienzle [KYX03] proposed two classification criteria to evaluate aspect interaction, the activation mechanism and the dependencies to classify aspect interactions.

The activation mechanism is defined when one analyzes the functionality of an aspect. One can separate this category into two kinds: the autonomous and triggered. An aspect is autonomous if it doesn't need to be called to perform its functionality, normally it does its actions continuously or periodically. The triggered aspects wait for others to activate it, and only then the aspect would execute.

The dependencies category has three kinds of dependencies: orthogonal, uni-directional and circular. The orthogonal aspect provides functionalities to the system that are absolutely independent from others' functionalities of the system. A uni-directional aspect depends on one or more functionality delivered by other aspects in the system. The uni-directional aspects are divided: into **uni-directional preserving**, which deliver new services based on other aspects, but do not change the other services, thus preserving other aspects' functionalities; and **uni-directional modifying** that replaces or modifies functionalities in the system.

Finally the circular dependencies, occur when one aspect depends on services provided by another aspect which in turn the second aspect also depends on a service offered by the first aspect.

### 3.2.4 Frans Sanen et al. Classification

Frans Sanen et al. [STJ<sup>+</sup>06] distinguishes four distinct categories to classify different aspects' interactions. This classification is based on the different manners that an aspect may interact with each other.

- **Mutual exclusion:** happens when the interaction of mutual exclusiveness is encapsulated. Take for example two aspects that execute similar algorithms, and the situation that only one aspect is needed. Negotiation will not take place because only one can be executed and the other cannot.
- **Dependency:** this category is similar to the ones presented in [KYX03], it happens when one aspect needs another aspect and therefore depends on it. This kind of dependency will not bring conflict as long as the aspect that has a dependant doesn't change or isn't erased.
- **Reinforcement:** occurs when an aspect interacts with another aspect in a positive manner, consequently reinforcing the functionality of the second aspect. For example one aspect can add new features to other aspect.
- **Conflict:** this category happens when one adds an aspect to a working system and as a result some of the aspects that were previously in the system may not work correctly.

### 3.2.5 Rinard, Sălcianu and Bugrara Classification

Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara [RSB04], have classified aspect interactions having in attention the relationship between advices and methods. They have identified two kinds of interaction:

- **Direct interaction:** occurs between advices and methods that the advice crosscuts
- **Indirect interactions:** happens when advices and methods may access the same object fields.

The direct interaction is divided in four categories. In **Augmentation** the method crosscut is always executed, the advice is orthogonal to the execution of the method. The logging aspects example is a clear augmentation type of interaction. **Narrowing** defines that after the crosscutting of the method one of the following situations occurs either the body of the method is executed or none of the body is executed. This interaction occurs whenever the advice checks safety conditions before the execution of the advice. In the **Replacement** interaction, the method will not execute at all, the advice will replace the entire body of the method. Finally, the **Combination** interaction will combine the crosscut method and the aspect in order to make a complete new behavior.

The indirect interactions define the scope which advices and methods have over the core of the application, as for example the access to objects and its fields. The **Orthogonal**



interaction is when one advice and the method access different fields, in this situation the scopes of both are orthogonal. An interaction is **Independent** when neither the advice nor the method write or read a field that the other has access to read or write. The **Observation** is defined when the advice may read one or more fields that the method writes. The Actuation interaction is the opposite, this when the advice writes one or more fields that the method reads. At last, the **Interference** occurs when one advice and one method write in the same field, this is the real interference problem in AOP that are in the origin of many conflicts.

### 3.2.6 Clifton and Leavens Classification

Clifton and Leavens [RSB04] propose a much simpler aspect interaction definition. They define that the advices may be Spectators and Assistants. Spectators are advices that do not modify the control flow of an advised method and also don't change fields, while Assistants can alter the control flow of the crosscut, method and change objects fields.

## 3.3 Conflicts Detection

Some of aspect interaction classifications presented in the previous section introduce the kinds of conflicts that are present in AOP. These conflicts are the consequence of the functionality that some mechanisms provide. Several authors have developed detection techniques in order to better understand how aspects interact with each other and possible conflict detection. This section will explain how some of these techniques are applied in AOSD.

One of the major problems with AOP is that when a new aspect is added to a system an unexpected behavior can occur. Due to the semantic nature of pointcuts, it is sometimes difficult to assure that they will crosscut the expected methods and sometimes they crosscut methods that weren't supposed to be crosscut. This is clearly a semantic conflict.

A semantic conflict is an emerging behavior that conflicts with the originally intended behavior of one or more of the involved components. This conflict arise when there are shared joinpoints in the code. This kind of conflict happens when there is no idea which aspect should execute first when we are in the presence of a shared joint point.

Consider the following example: a simple class that implements a protocol. It has two main functions *sendData* and *receiveData*. But the client wants two more requirements, encryption and logging . We are using AOP as our programming paradigm we should develop two aspects, *EncryptionAspect* to encrypt outgoing messages, decrypt ingoing

## Aspect Interactions and Conflicts

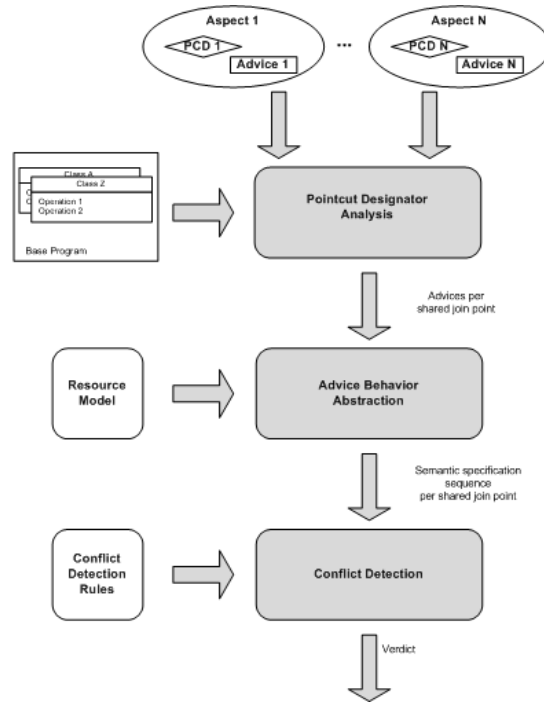


Figure 3.1: Analysis of a semantic conflict. Adapted from [DBA05].

messages and finally *LoggingAspect* to keep track when both methods are executed. The conflict arises when these aspects are applied to the same joinpoint that led to an undesired behavior. The order of execution of the two aspects should be defined to avoid this situation. But in big software projects this may not be an easy task.

The way to detect this conflict was presented by Durr et al. [DBA05] It consists in a formalization that make possible to express a behavior and conflict detection rules.

For a better understanding of this, a resource model was developed to represent the critical behavior of an advice and for the detection of semantic conflicts between advices. Figure 3.1 explains the necessary steps for this technique.

The Pointcut Designators(PCD),are the inputs for the pointcut designator analysis phase. During this stage the PCDs are evaluated according to the base program.

The next stage of this technique is the abstraction phase. From the previous phase we get a sequence of advices per shared joinpoints. The resource model is the last input for this phase. Along this stage the sequence of advices are transformed into sequences of resource-operation tuples per shared join point.

The final stage has as input the operation sequences per resource per shared joinpoint, and the conflict detection rules. In the end of this stage a verdict is determined, i.e. if a conflict is present or not, for each that was previously obtained.

Koppen and Störzer [SK04], have also proposed a technique to prevent the problem concerning pointcuts. The PointCut Delta Analysis is able to detect how many and which pointcuts have been added or eliminated. With this information, one can know exactly the steps taken before any conflict involving the pointcuts that may appear.

Their approach can be read as follows:

*Let  $P$  be the program before,  $P'$  the program after the edits. Let  $joinpoints(P)$  be a function calculating all joinpoints for the given program  $P$ ,  $advice(j, P)$  a function listing all pieces of advice at a given joinpoint  $j$  in program  $P$  (for  $j \notin joinpoints(P)$  define  $advice(j, P) = \emptyset$ ).*

Let

$$JP = joinpoints(P') \cup joinpoints(P)$$

be the set of joinpoints in both program versions.

Let

$$add(P, P') = \bigcup_{j \in JP} advice(j, P') - advice(j, P)$$

be the set of additional advice matches and

$$del(P, P') = \bigcup_{j \in JP} advice(j, P) - advice(j, P')$$

the set of lost advice matches. We are then interested in the set

$$deltaPC(P, P') = (add(P, P') \cup (del(P, P'))$$

István Nagy, Lodewijk Bergmans and Mehmet Aksit [NBA04] have proposed a solution which permits to specify the composition of the aspects. They implemented a model that can be built in the various AOP languages. To build this model they have determined two constraints to specify the composition of aspects at shared joinpoints. These constraints define the dependencies between advices. The Ordering constraints indicates the order of execution of a set of advices. The Control constraints, also affect the execution order too, but they can impose conditions on the execution of the advices.

Mehmet Aksit et al. [ARS09] have developed a very powerful technique for the detection of shared joinpoints, using graphs. This approach applies a graph transformation formalism to identify the different semantics of aspect-oriented programming languages

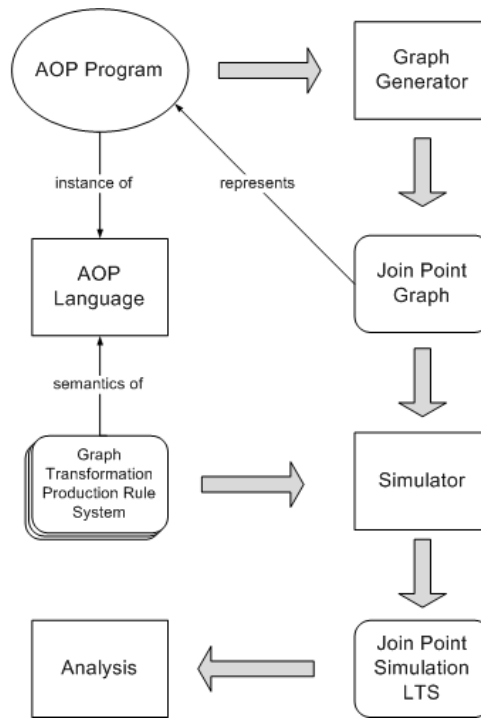


Figure 3.2: Detection of shared join points. Adapted from [ARS09].

in a specific way. With this graph model it is possible to simulate the execution of aspects, and one can observe the different advice orders and detect interferences between the aspects.

Figure 3.2 describes all the process of the technique, first the graph for each shared joinpoint is generated and then the semantics of the aspect-oriented language are specified, with the help of the graph production rules. The graph of the shared joinpoint represents a snapshot of the system, but only the fraction in which is needed for the current simulation. This is done in order to reduce the cost of the computation since it would be too expensive to simulate the entire system. The graph of the shared joinpoints and the semantics rules system are then passed to the simulator which will create a state space (LTS-labeled transition system) of the execution of the all the advices at a specific joinpoint in all possible order. With the LTS we can analyze the joinpoints, and detect any possible interference by comparing the results of all the possible orders of execution.

Havinga et al. [HNBA07] have identified conflicts which had their origin in the inter-type declaration also called introduction mechanism.

These composition conflicts happen, for example when one add a new method, variable, annotations or even change the inheritance struture. These changes can be intended

but they may also be unintended causing a program to fail, compile or even become ambiguous.

Havinga has identified three types of errors that can bring conflicts related to the introductions:

### 3.3.1 Violation of Language Rules

In listing 3.1 we have two different aspects, both introduce a new method named *saveChanges*. Aspect *PersistenceImplementation* introduces this new method in all classes that implements the interface *Persistent*. The second aspect *ObjectCache* adds a method with the same name to the class *BusinessObject*. At a first look it isn't obvious that we are in the presence of a conflict, because the type patterns used to introduce these methods are different. Though, as the *BusinessObject* class implements the *Persistent* interface, the two aspects introduce a method exactly with the same name to the class, which eventually will bring us a naming conflict.

Listing 3.1: Violation of Language Rules

```

1 public interface Persistent { ... }
2
3 public class BusinessObject implements Persistent { ... }
4
5 aspect PersistenceImplementation {
6     void Persistent.saveChanges() { db.update(...); }
7 }
8
9 aspect ObjectCache { ...
10     void BusinessObject.saveChanges() { cache.setVal(...); }
11 }
```

### 3.3.2 Introduction has unintended effects

The introduction may have unexpected effects. The example in listing 3.2 describes this situation. Aspect *Printing* introduces a new method named *getSize* in the class *AlertDialog*. But this introduction has a secondary effect, it overrides a method with the same name that the class *AlertDialog* already inherited from its parent class *DialogWindow*. This is clearly another type of conflict that can appear when one uses the introduction mechanism.

Listing 3.2: Introduction has unintended effects

```

1 public class DialogWindow {
2
3     public Rect getSize() {
4         // return window dimension..};}
5
6     public class AlertDialog extends DialogWindow {
7
8         public AlertDialog(String alertMsg) {...}}
9
10    aspect Printing {
11
12        public Rect AlertDialog.getSize() {
13            // return paper dimension..} }

```

### 3.3.3 Ambiguous aspect specification

The third type of composition conflict is caused by composition specifications referencing and modifying the same program model. The structure of the program is changed when one introduces new modifications to the base class, but it is also "queried" by the pointcut designators. It is possible that a pointcut can point to the new elements that were changed or introduced in an aspect. Therefore, introductions can influence the composition as specified by pointcuts. Listing 3.3 designates a pointcut, which selects all classes that implement the interface *PersistentRoot*. Nevertheless, classes can be modified to implement this interface by using the `declare parents` construct. With this approach, the pointcut depends on this modification of the program structure. This may be intentionally, but these dependencies can be harmful and lead to ambiguous code.

Listing 3.3: Ambiguous aspect specification

```

1 declare parent: BusinessObject implements PersistentRoot;
2
3 pointcut persistence():
4     execution(* PersistentRoot+.*(..));

```

Havinga uses a graph-based approach to model and detect, composition conflicts related to introductions. This method analyzes these conflicts by converting the structure of the source code of Java and AspectJ programs into a graph-based model. The introductions made in the aspects code are converted into graph transformation rules that can be applied to the program model built previously. The model is then analysed with another tool and composition conflicts are modelled as graph matching patterns. These patterns

are matched against the program model, in order to identify possible conflicts that exist in the program.

Havinga et al. [HNBA06] have also studied issues related to the introduction of annotations in AOP applications resulting in inter-dependencies conflicts. First, let's introduce how annotations work in Java and AspectJ. Annotations in AspectJ can help preventing the problem with fragile pointcuts. In listing 3.4 the annotation **@Persistent** class **Product**, defines that the class **Product** has attached the annotation **Persistence**.

Listing 3.4: Class Product

```
1 @Persistent class Product{
2
3   String name;
4   Double price;
5 }
```

The annotations can be applied to classes, methods and fields. Listing 3.5 shows an example of a pointcut declared with annotation. The first **\*** refers that it will match any return type of the methods, the **(@Persistent \*)**, means that the class that contains the crosscut method needs to have the annotation **Persistent**. The rest of the pointcut specifies that all methods will be matched regardless their name and number and type of parameters. Since these pointcuts are specified with annotations, we can select joinpoints based on the design information, rather than using semantic patterns, assuring more robust pointcuts.

Listing 3.5: Declaration of a pointcut

```
1 pointcut isPersistent():
2 execution(* (@Persistent *).*(..));
```

The example above is quite useful in helping preventing the fragility of pointcuts, however the problem with annotations is when they are used in inter-types declarations. The annotations can bring dependencies between introductions of annotations, i.e. the order of the declaration of each annotation is important. If the order of the introductions isn't specified, ambiguities may occur. Consider now listing 3.6, the first line states that the annotation **@TransientClass()** is applied to the class **SessionID** and all its subclasses. The second line indicates that all the fields of class marked with the **@TransientClass** annotation, should be marked with the **@TransientField** annotation.

Listing 3.6: Introduction of annotations

```

1 declare @type : SessionID+ : @TransientClass();
2
3 declare @field: (@TransientClass *) * :@TransientField();

```

The above code doesn't have any problem related with introduction of annotations, but if the order of the declaration is changed, i.e. if the declare field is introduced first, it wouldn't have any effect on code. The second introduction clearly depends on the first statement, causing a dependency problem. This case is real simple to analyze but there are some cases that this analysis isn't so easy to detect, because it can exist more than one level of dependency.

To help prevent this problem Havinga [HNBA06] proposed an algorithm that analyses the source code that contains annotations, detecting possible dependencies, and providing a correct resolution for any dependency that may exist.

Mussbacher et al. [MWA08] have proposed a new approach for the detection of interactions based on lightweight semantic annotations of aspect models.

Each aspect model is annotated with domain-specific markers and a separate goal model is used to describe how the semantic markers from other domains can influence each other. When the aspect models are composed, the resulting models are inspected to check for any semantic marker that can be a potential conflict. This is achieved by the propagation of values through the goal model to detect which goals are satisfied by the composition and the ones that are not satisfied.

This technique can be used both to highlight potential aspect conflicts and to trade off aspects.

### 3.4 Program Slicing

The idea of program slicing analysis in AspectJ programs was suggested by Balzarotti [BM04]. This technique considers that a slice of a program is a set of statements which affect a certain point in an executable program. Because AspectJ programs are woven in the Java byte codes, which are executed by the Java Virtual Machine, he considers two slicing approaches:

- Advices and methods, are considered as first class entities.
- Analyze the woven code.



The condition to check non interfering aspects proposed by Balzarotti is the following one:

*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:*

$$A1 \cap S2 = \emptyset$$

### 3.5 Regression Testing

Katz [Kat04] has suggested that regression testing can be used in order to detect harmful aspects that are added to a software system. It is a very simple idea this one. First the system is tested to see if it has any errors. If the system passes in the tests i.e. it has no errors, the new aspect can be added to the application and then the tests are run again to see if the system still works correctly. If an error occurs the test may need to be changed in order to be more explicit to the new aspect that was added, but if the system doesn't pass the tests then we may be in the presence of a conflict.

### 3.6 Aspect Interactions Charts

All of the detection techniques mentioned before, are integrated in tools that analyze AOP programs in a static or a dynamic way, so it implies that some code has already been written. Shubhanan Bakre and Tzilla Elrad [BE07] have proposed another approach. They state that the detection of aspect interactions is the responsibility of the system designers and since the current modeling languages aren't enough to capture the abstraction that aspect interactions require, they propose the use of Aspect Interactions Charts (AIC) which are build on Live Sequence Charts(LSC). These charts can be used to identify the numerous interactions between the various aspects at their join points.

### 3.7 Summary

This chapter has introduced the different classification systems of aspects interactions proposed by some authors. With these definitions it becomes easy to understand how aspects interact with each other. We can also notice that not only it is important to take in attention the aspects that interact in other aspect but also the mechanisms of the aspects are relevant to understand the origins of the conflicts that arise.

Despite the detection techniques that were introduced in the second section represent the most advanced tehcniques that we have at our diposal, these aren't quite reliable as

we should expected. This is because some of conflicts that exist in AOP are really hard to identify and some technique may even discover false negatives, i.e. some of the issues that detect may not really be conflicts.

We can conclude that the most important property of AOP can be also its Achilles kneel, the obliviousness that some authors praise so much can really bring some heavy problems to the AOP evolution.

## Chapter 4

# Catalogue of Conflicts

The objective of AOP is to improve modularity by allowing crosscutting concerns to be developed as single units of software. Despite all the advantages that AOP brings to software development it is not clean of critical issues. One of the major drawbacks of AOP is the unexpected interactions that occur between aspects that can be quite frequently than those that happen in normal OOP development.

The main objective of this master thesis was to write a catalogue containing a description and examples of each type of conflicting interaction, in order to better understand how these interactions happen and how they can be detected. This catalogue will be extremely helpful for future research work or to develop techniques and tools for conflict detection and resolution.

### 4.1 Catalogue Structure

The first thing to decide before composing the catalogue is to define the fields that should compose each catalogue entry. To get the much information as possible about a conflict the catalogue was composed having in attention six fields. The following list explains the function and what they mean of each field in the catalogue.

- Conflict Name: The name of the conflict;
- Description: A brief description about the conflict;
- Context: The context in which the conflict may arise;
- Category: The type of conflict;
- Example: A small example of code to show how the conflict typically appears.

- **Resolution or Preventive Measures:** Because not all conflicts have a solution some preventives actions are suggested in order to avoid further problems.

## 4.2 Conflicts Categories

To identify the conflicts that appear in AOP, it was important to study which were the causes of such problems. The best way to start the study on this issue was to understand how aspects could interact with each other. This was done by investigating how researchers classified interactions between aspects. Some of these classifications systems proved to be very useful when we started to write the catalog.

In some of the articles that were studied already contained some information about the conflicts. But the lack of some details about them introduced some difficulty to understand them. The other method used to identify the conflicts was the testing of some source code that could have conflicts. There are some examples in the catalogue that do not refer to any specific example, but these conflicts are possible to occur.

This catalogue aims to give a good resource about the conflicts. Other researchers may find this as a good guide when studying the same concepts that are documented.

We have distinguished three types of category to classify the conflicts. They are on the following list:

- **Semantic Conflict:** This conflict occurs because of the semantic nature of designating pointcuts, and joinpoints. Normally these are the hardest conflicts to detect.
- **Behavioral Conflict:** This category is called behavioral because these conflicts only change the behavior of the system.
- **Introduction Conflict:** The Introduction conflicts have their origin when we add new members to classes.

## 4.3 Conflicts Documented

We have identified eight conflicts. Some may be very similar but are in fact a quite different when we analyze the source code example. The following list does a small presentation of them:

- **Fragile Poincuts:** This conflict occurs when one pointcut was working correctly, before a change of the original code. After the changing the pointcut doesn't work anymore.
- **Shared Joinpoint:** Appears because it wasn't defined the order of execution of aspects.

- **Shared Variables:** This conflict occurs when two aspects have read and write access to the same variable.
- **Dependent Aspects:** This situation occurs when one aspect depends on some condition of another aspect.
- **Advice Interference:** The advice around defined in one aspect replaces the execution of the method that it crosscuts. Because this method has a call to another method that is crosscut by another aspect, the control flow of the program is changed.
- **Incorrect Method Call:** An aspect introduces a new method member in a class. That already inherits from its parent class a method with the same name of the new introduced method.
- **Dependent Annotations:** Annotations made in classes may be dependent so its declaration order is important to define.
- **Incorrect Aspect Precedence:** This isn't a real conflict like the previous ones. It's more a like a conflict that occurs in the requirements stage of the life cycle of software. Because the declare statement of AspectJ doesn't support conditional execution it may conflict with some requirement of a software project.

In the next sections all the documented conflicts are explained in detail according to each field that we defined to describe the conflicts.

## 4.4 Conflict Fragile Pointcuts

### Description

This conflict occurs when one pointcut was working correctly, before a change of the original code. After the changing the pointcut doesn't work anymore.

### Context

Since software is constantly evolving, naturally, new implementations will be made to the software. These changes may be the addition of new pointcuts or deleting old ones, changing the original code etc.

Because pointcuts are defined at a semantic level it's hard to guarantee that they intercept the correct methods, when using wildcards. The programmer has to be extremely careful when coding the pointcuts.

### Category

Semantic Conflict

### Example

This kind of problem, isn't a conflict per se. The difficulty here is that it will bring conflicts that didn't exist before any code adjustment or due a bad coded pointcut. For instance take a look on listing 4.1, one can conclude that this point is too weak and will crosscut many methods. This pointcut is supposed to crosscut methods that pay the salaries of the collaborators of a company, but what if someone added a new method called *payBills()*, that would pay the bills that the company had. This pointcut would surely crosscut this method.

Listing 4.1: Pointcut payColaborators

```
1 public pointcut payColaborators():  
2   execution(* pay*(..));
```

### **Resolution or Preventive Measures**

For this type of conflict the only resolution possible would be that programmers need to be very careful when coding pointcuts. They should take in attention the requirements of each project that they are working on. Therefore one must have in attention the strength of the pointcut, i.e., the pointcut can be too strong and capture only a few desired methods or too weak and will capture way too many methods.

## 4.5 Conflict Shared Joinpoint

### Description

Sometimes multiple aspects apply to the same joinpoint.

### Context

As part of aspect oriented programming, sometimes multiple aspects may define the same joinpoint. Because this paradigm involves some modularity this means that some of the programmers won't know that other programmers involved in the project may use the same join point.

This may raise some issues like which order should the aspects execute, or even if there is any dependency between them.

### Category

Semantic Conflict

### Example

The following examples give us an idea how important is the order that aspects execute. As a first requirement for this situation we need a logging system to monitor the changes of salaries of company's employees. Aspect *MonitorSalary* in Figure 4.1 is implemented in order to perform this requirement. When the method *increaseSalary* executes it will print a message with the information about the employee and its level.

Now the second requirement is to add persistence to this information about the changes of the salaries. If the salary change the aspect *DBEmployeePersistence* will update the database with this information.

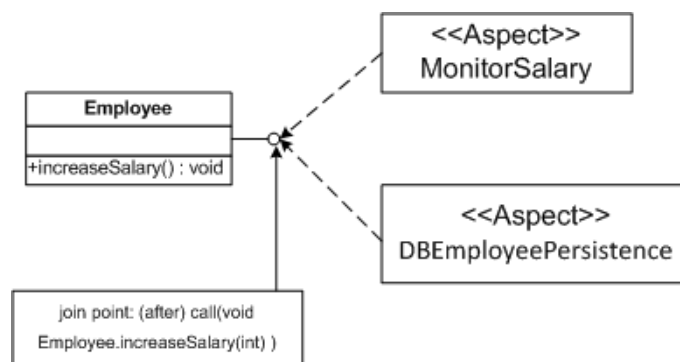


Figure 4.1: Two aspects are superimposed in the same joinpoint.



As we can see these two aspects are now superimposed in the same joinpoint. The two aspects can work correctly if isolated from each other, but when combined an unexpected behavior can occur. The problem here is that the database needs to be updated just after the change occurs in the object, and the advice on *DBEmployeePersistence* aspect needs to be executed before the advice in *MonitorSalary* aspect.

### Resolution or Preventive Measures

There is an easy solution for this problem. AspectJ provides the mechanism to fix this scenario. We can declare the execution order of the aspects. By specifying the following line of code of listing 4.2, we are declaring that the advice in aspect *DBEmployeePersistence* will execute before the advice in aspect *MonitorSalary*.

Listing 4.2: Define precedence between two aspects

```
1 declare precedence: DBEmployeePersistence, MonitorSalary;
```

## 4.6 Conflict Shared Variables

### Description

This conflict occurs when two aspects have read and write access to the same variable.

### Context

Since AOP is all about modularization and separation of crosscutting concerns, its normal that the most important feature in AOP, the obliviousness, will be accomplished. Because of that some problems may occur when some aspects share the same variables. This will be an indirect interaction between the aspects that share the same variable. For example, imagine a particular case where one aspect will need to get a specific value from a variable to execute an advice, if that value was previously changed by some other aspect, it will definitely interfere with the normal execution of the first aspect.

### Category

Behavioral Conflict

### Example

Consider the following example in Listing 4.3 where the class *Object* has one method called *something(int i)*. This method only prints a message in the console. It also has a integer field called *var*, that is initiated at zero. We have two aspects in this example. The aspect *DoSomething* has a pointcut that crosscuts the method *something(int i)*, if the value of *var* is zero (line 18) and then updates the *var* field to two. The second aspect *DoSomethingElse* has the same pointcut that crosscuts the method *something(int i)*, if the value of *var* is zero (line 30) and finally prints out a message to the console.

At a first view, there is nothing wrong with the class and the two aspects. Each aspect works without problems when executing separately. When the aspect *DoSomething* executes isolated from the other aspect it work correctly. The same thing happens with aspect *DoSomethingElse*, working perfectly without any flaws.

The conflict arises when we add the two aspects to the system. Since the aspect *DoSomething* modifies the value of *var* to one, the aspect *DoSomethingElse* will never execute, its function of printing the message to the console.

Listing 4.3: Class *Object* and aspects *DoSomething* and *DoSomethingElse*

```

1 public class Object {
2     public int var=0;
3     public static void main(String[] args){
4         Object o = new Object();
5         o.something(4);

```

## Catalogue of Conflicts

```
6  }
7  public int something(int i) {
8
9      System.out.println("something executed.");
10     return 0;
11 }
12 }
13
14 public aspect DoSomething {
15     public pointcut somethingCalled(Object o):
16         execution (int something(int)) && this(o);
17     after(Object o): somethingCalled(o){
18         if(o.var == 0)
19         {
20             o.var =2;
21             System.out.println("after DoSomething: "+o.var);
22
23         }
24     }
25 }
26 public aspect DoSomethingElse {
27     public pointcut somethingCalled(Object o):
28         execution (int something(int)) && this(o);
29     after(Object o): somethingCalled(o){
30         if(o.var == 0)
31         {
32             System.out.println("after DoSomethingElse: " +o.var);
33         }
34     }
35 }
```

## Resolution or Preventive Measures

A simple solution to this problem would be adding one extra field to the class *Object*.

This new field would keep the old value of the field *var*. Aspect *DoSomething* will store the value of *var* in the variable *oldVar*. With that new field aspect *DoSomethingElse* will compare the value of the variable, that is supposed to be the value of *var* and therefore will print its message. In Listing 4.4 we can see the changes in the code, first we add the field *oldVar* to the class *Object*; and in the advice *somethingCalled* in aspect *DoSomethingElse* we add the condition if the *oldVar* is equal to zero.

Listing 4.4: Solution

```
1 public class Object {
2     public int var=0;
3     public int oldVar;
4     public static void main(String[] args){
5         Objecto o = new Objecto();
```

## Catalogue of Conflicts

```
6    o.something(4);
7  }
8  public int something(int i) {
9    System.out.println("something executed.");
10   return 0;
11  }
12 }
13 public aspect DoSomething {
14   public pointcut somethingCalled(Object o):
15     execution (int something(int)) && this(o);
16   after(Object o): somethingCalled(o){
17     if(o.var == 0)
18     {
19       o.oldVar=var;
20       o.var =2;
21
22       System.out.println("after DoSomething: "+o.var);
23     }
24   }
25 }
26
27 public aspect DoSomethingElse {
28   public pointcut somethingCalled(Object o):
29     execution (int something(int)) && this(o);
30   after(Object o): somethingCalled(o){
31     if(o.var == 0 || o.oldVar == 0)
32     {
33       System.out.println("DoSomethingElse after modify: " +o.var);
34     }
35   }
36 }
```

Another possible conflict similar to this can also emerge from a transitive interaction. Imagine that *AspectA* modifies a variable  $x$ , and this variable is passed to a method  $f$ , that uses  $x$  to define a variable  $y$ . The value of  $y$  will be used in another *AspectB* to decide this aspect behavior. Figure 4.2 illustrates this situation.

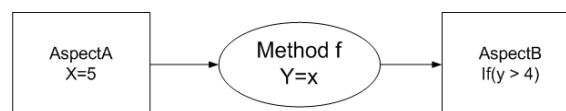


Figure 4.2: Transitive interaction, the value of  $x$  is passed to  $y$

## 4.7 Conflict Dependent Aspects

### Description

This situation occurs when one aspect depends on some condition of another aspect.

### Context

A dependent interaction between aspects occurs when an aspect depends on a condition that is established by another aspect. Imagine an aspect that is woven into a system, its execution will depend on one or more conditions that are associated with this aspect and if one of those conditions will never be met, the aspect will not execute. A condition can be defined as a boolean expression that evaluates a context in which the aspect is deployed.

### Category

Behavioral Conflict

### Example

Let's continue to explore the previous example in the conflict shared joinpoint, where we had to update a database whenever the salary of an employee changed.

Now we need to add a new functionality. We need to check if the salary of the employee is higher than his manager salary. Aspect *CheckRaise* in Listing 4.5 implements this functionality. If the raise of salary of the employee is higher than his manager salary it will print an error message and the salary of the employee will change to its original value. We need to implement some changes in aspect *DBEmployeePersistence* in order to decide about the execution of this aspect changes are shown in listing 4.6.

The problem here is that aspect *CheckRaise* affects the composition, if this aspect fails the *DBEmployeePersistence* aspect shouldn't execute, because the data will not change. The execution of aspect *DBEmployeePersistence* depends on the result of aspect *CheckRaise*.

Listing 4.5: Aspect checkRaise

```

1 public aspect CheckRaise pertarget(target(Employee) ){
2   private boolean _isValid;
3
4   public boolean isValid(){
5     return _isValid;
6   }
7
8   before(Employee person, int level):
9     MonitorSalary.salaryChange(person, level){
10    _isValid = true;

```

## Catalogue of Conflicts

```
11     } // workaround for conditional execution
12
13     after(Employee person, int level):
14         MonitorSalary.salaryChange(person, level) {
15             Manager m=person.getManager();
16             if ((m!=null) && (m.getSalary() <= person.getSalary())) {
17                 //Warning message
18                 System.out.println("Raise rejected");
19                 //Undo
20                 person.decreaseSalary(level);
21                 //workaround for conditional execution
22                 _isValid = false;
23             }
24         }
25     }
```

Listing 4.6: Aspect DBEmployeePersistence

```
1 public aspect DBEmployeePersistence {pertarget (target(PersistentObject)) {
2
3     private boolean _isUpdated;
4
5     public boolean isUpdated(){
6         return _isUpdated;
7     }
8
9     // workaround for conditional execution
10    after(PersistentObject po): stateChange(po) {
11        if (CheckRaise.aspectOf((Object)po).isValid()){
12            System.out.println("Updating DB...");
13            po.update(po.getConnection());
14        }
15    }
16 }
```

## Resolution or Preventive Measures

Its not easy to implement a conditional execution of aspects, because most of AOP do not provide useful mechanisms for this scenario. In AspectJ we can use workarounds, like maintaining boolean members in aspects but an effective composition cannot be achieved. A possible solution would be to introduce extra advices to maintain the boolean variables and more if-statements inside the aspects in order to handle these variables.

Another possible solution for this conflict proposed by Frans Sanen et al. [[STJ08](#)] , where they state that its possible to compose these dependencies using default logics.

## 4.8 Conflict Advice Interference

### Description

Some advices may interfere with the correct flow of the application.

### Context

Advice code may execute in several ways inside an aspect. They may introduce new functionalities, execute the method multiple times, impose conditions to the execution of methods, change the control flow of the program, or even replace the entire execution of the crosscut method. Some of the examples stated above may bring some conflicts to the application due to the direct interaction that occurs between advices and methods. Since the aspect will probably work correctly isolated, we cannot guarantee that it will function properly when some other aspect are introduced to the system.

### Category

Behavioral Conflict

### Example

Consider the following example in Figure 4.3 where a class defines two methods *foo* and *bar*, and the *foo* method is invoked inside the *bar* method.

Now consider two separated aspects, *AspectA* declares a pointcut on method *foo* and has a before advice associated. The second aspect, *AspectB*, has a pointcut to capture method *bar* but in this case there is an around advice to advise the method execution. The problem here occurs when there is no call to the method *proceed()* inside the around advice as we can see in Figure 4.3. This situation will replace the entire execution of the method *bar* and consequently the method *foo* that is invoked inside the method *bar* will not execute.

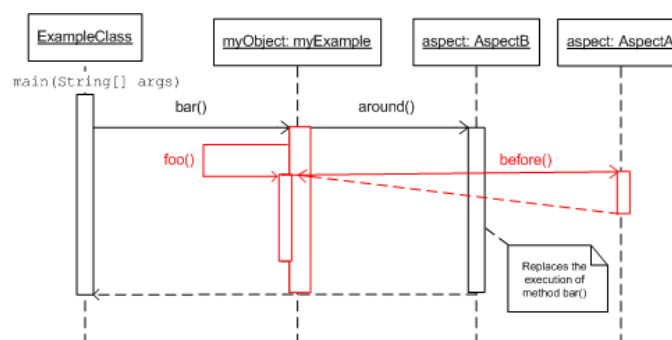


Figure 4.3: Advice around replaces the method execution

### **Resolution or Preventive Measures**

For this case there isn't much we can do. This is clearly an anti pattern we are facing here. If the programmer really wants to capture the method *foo* execution in another aspect he cannot declare an around advice on method *bar*. If this conflict arises in any software project there isn't a really viable solution for this problem.



## 4.9 Conflict Incorrect Method Call

### Description

A wrong method is called, despite it has the correct name.

### Context

The mechanism of inter-type declaration is very powerful allowing one to add new methods or fields to the classes. One can also modify the hierarchy of the base class.

This situation may occur when one have a class that inherits the methods and fields of a mother class. If someone else implements an aspect that introduces a method that has the same name that another method that is coded in the parent class and has a different function, this same method will be overridden. Therefore the behavior of the system will not be the expected by the programmer.

### Category

Introduction Conflict

### Example

In Figure 4.4 we can see that the class *Manager* inherits *Employee* class methods. At the beginning of the development everything is working accordingly to the requirements.

Since software evolves, someone asks for new functionalities, thus leaving us to add new modules to the application. In this example we add the aspect Introduction that declares a new method inside the class *Manager*. As we can see this new method overrides one of methods of the class *Employee*, resulting in a different action of the application. This new behavior may be the intended by the programmer, but it can be also unexpected because of the obliviousness that AOP offers.

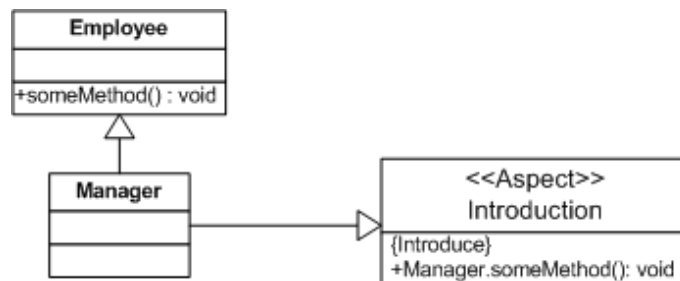


Figure 4.4: Introduction of a new method

### **Resolution or Preventive Measures**

For this type of conflict there is no way around to solve it in an elegant approach. The programmer itself has the responsibility to carefully study the system on which he is implementing new functionalities. This conflict is very annoying since the aimed obliviousness of AOP will not be met in this solution.

We can change the method names that were introduced by the aspect, in order to avoid the naming conflict.

## 4.10 Conflict Dependent Annotations

### Description

Some annotations declared in aspects are dependent on the order which they are introduced in aspects.

### Context

Annotations are a useful tool in object-oriented programming. Class methods fields can be annotated, but don't have any direct effect on the execution of the code they annotate. AspectJ provides annotations too. We can declare class annotations in an aspect. For example, one can declare annotations on types and fields of classes.

### Category

Introduction Conflict

### Example

Consider the following example of introduction of annotations in a class in AspectJ in [HNBA06]. Suppose we want to annotate the class *SessionID* and all its subclasses with the annotation *@TransientClass* we could do as in listing 4.7.

Listing 4.7: Annotation *@TransientClass*

```
1 declare @type : SessionID+ : @TransientClass();
```

Now we introduce another annotation in this one we want to annotate all the fields of the classes annotated with the *@TransientClass* annotation with the following annotation *@TransientField*. This can be done like the code in listing 4.8.

Listing 4.8: Annotation *@TransientField*

```
1 declare @field: (@TransientClass *) * :@TransientField();
```

If we declared these introductions by this order, first annotate the classes and then its fields, there wouldn't be any problem at all because the weaver will annotate first the class and its subclasses and finally it would introduce the annotations on their fields. But if we change the order of the declarations nothing should happen, because the annotation of the class will be done after the annotation of the fields.

This is a very small example and therefore its easy to detect this problem just by looking in the code, but imagine that this was done in a more complex software system, it would be extremely difficult to detect this conflict.

### **Resolution or Preventive Measures**

Like the previous conflict with introductions on the previous conflict, we cannot do much to solve this conflict. The only way to get a solution for this kind of conflicts related with introductions is to identify them on the design phase of the software lifecycle. Programmers should keep in mind that things should be prepared and studied in early stages of any software project development. Wilke Havinga et al. [[HNBA06](#)] have proposed a tool that can assist software developers in composing these types of introductions and detect possible dependencies between them.

## 4.11 Conflict Incorrect Aspect Precedence

### Description

This conflict happens when one needs that a specific order between aspects must be maintained accordingly to different conditions.

### Context

This type of conflict is originated because AspectJ doesn't have the ability to declare conditions of execution in its "declare precedence" statement. Sometimes the order defined by using the *declare precedence* isn't always valid and in some situations the order of execution needs to change due some required condition.

### Category

Behavioral Conflict

### Example

The example in listing 4.9 is impossible to implement in AspectJ, as the *declare precedence* doesn't support verification of conditions of execution between aspects.

Listing 4.9: Example

```

1 public aspect A{
2
3 declare precedence: A, B if (cond)
4
5 pointcut A1(): call(void CX.met());
6
7 before(): A1()
8 { ..... }
9
10 }
11 public aspect B{
12
13 declare precedence: B,A if (!cond)
14
15 pointcut B1(): call(void CX.met());
16
17 before(): B1()
18 { ..... }
19 }

```

## Resolution or Preventive Measures

This method was proposed by Sandra I. Casas et al.[CPSM07]. They have developed an extension for AspectJ named MEDIATOR. With this one can define associations between aspects and classes. Associations can be defined in a separated way, instead being tied to aspects, they link aspects with classes. In listing 4.10 we can see how an association establish a relation between the aspect *Logging* and the class *Account*. It states that every time the method *setBalance(float amount)*, that belongs to the class *Account*, is invoked the *loggedOperations()* method from aspect *Logging* is executed immediately afterwards.

Listing 4.10: Association LogAcc

```
1 association LogAcc {
2   call void Logging.loggedOperations();
3   after void Account.setBalance(float amount);
4 }
```

When the conflict is detected, one must define a rule to determine the order of execution, and which they must obey. The format of the rule is defined in the following text:

Rule: Id\_rule

Condition:  $(A_1, A_2, \dots, A_n)$

Action:  $R(A_1, A_2, \dots, A_n)$

The resolution of conflicts can be divided in several categories, Figure 4.5 summarizes the most common (A1,A2 and A3 represent associations).

Listing 4.11, help us understand how these rules can be applied. There are two associations *LoogAcc* and *StatisAcc*, two aspects *Logging* and *Statistic*, and finally one class *Account*. The rule R is used to try to solve any possible conflict. There is conflict between *LoogAcc* and *StatisAcc*, the rule R is then defined to solve this conflict. The category that this rule belongs is the combined optional-order.

Listing 4.11: Example of a solution

```
1 association LoggAcc{
2   call void Logging.loogedOperations();
3   after void Account.debit();
4 }
5
6 association StatisAcc{
7   call void Statistic.register();
8   after void Account.debit();
9 }
10
```

## Catalogue of Conflicts

```

11
12 Rule R
13
14 Condition: LoggAcc, StatisAcc;
15 Action:
16   if (n)
17       order (LoggAcc, StatisAcc);
18   else
19       order (StatisAcc, LoggAcc);

```

Combination	Example
Order-Nullity	order (A1, A2), anulled (A3);
Optional-Order	if (cond) order (A1, A2); else order (A2, A1);
Optional-Order- Exclusion	if (cond) order (A1, A2); else excluded (A2, A1);
Optional – Order - Nullity	if (cond) order (A1, A2, A3); else order (A2, A1); anulled (A3);
Optional - Order - Nullity - Exclusion	if (cond) excluded (A1, A2, A3); else order (A2, A1); anulled (A3);

Figure 4.5: Conflict Resolution Categories. Extracted from [CPSM07].

## 4.12 Summary

We can identify three main causes of the conflicts that occur when aspects interact with each other. One of the most common causes of conflicts between aspects is when two or more aspects share a variable or a data object. The aspects may work correctly when isolated but when one try to combine these aspects the system may act in an unexpected way.

The second cause is when an aspect changes the control flow of a program, for example imagine that one aspect has one around advice without the proceed statement. Now this around advice advises a method that has a call to another method inside its block statements. If this second method is supposed to be crosscut by any other aspect, this will not occur since the around advice declared in the first aspect will replace the execution of the method.

Finally, one of the mechanism of AOP, the introduction or inter-type declaration, that enables us to add new methods or variables or even change the inheritance of a class, can bring some conflicts to AOP programs. Since all software evolves, new features may be needed to be implemented, eventually some programmer may introduce some method with the same of any other method declared before. Therefore the method will be overridden. This may be an intended decision made by the programmer, but of course if the he wants to maintain the obliviousness property of AOP the programmer may be unaware of the existence of another method with the same name.

Because aspects will interact with each other, it's crucial that these interactions do not interfere with the expect execution of the program. With the conflict origins identified, the programmers may be more aware of the dangers of some of the most common techniques in AOP.

The next chapter will describe the development of a small prototype, that detects some AOP conflicts.



## Chapter 5

# Prototype

In the previous section several conflicts related with the development of AOP applications were catalogued in order to future programmers to be able to study which problems may arise with the integration of new aspects to a software system.

In this chapter it is described the development of a plug-in for the Eclipse IDE to detect some of the conflicts that are present in the catalogue. The plug-in was built on top of the AspectJ Development Tools (AJDT) that is a set of plug-ins supporting the development of AspectJ applications within the Eclipse IDE.

### 5.1 Eclipse Architecture

The Eclipse IDE [iS09] is a powerful tool to develop software systems. Eclipse isn't an enormous single Java application, it provides the functionality of a normal plug-in loader. Eclipse is composed by several plug-ins. A plug-in is just another Java program that extends the functionality of Eclipse.

For example, originally Eclipse was built to develop Java software but a plug-in to develop C++ is also available thus adding new functionalities. Plug-ins can also consume services offered by other plug-ins or extend its own functionality to be consumed by other plug-ins. They can be loaded dynamically by the IDE at run time whenever the user demands its functionalities.

Since Eclipse is an open platform it is designed to be extensible by third parties. The core is the eclipse software development kit (SDK) that can be used to build a diverse set of tools and products. These products can be extended by other products and so on. For example, one can extend a simple text editor to create source code viewer.

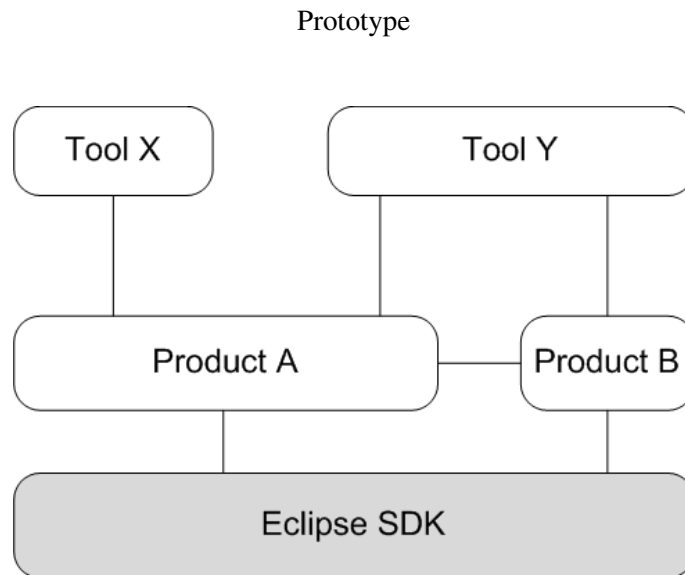


Figure 5.1: Eclipse extensibility, products can extend other products functionalities. Adapted from [iS09].

Figure 5.1 give us a visual glimpse of the eclipse extensibility where we can see how products relates with each other and with the Eclipse SDK.

The Eclipse SDK is divided in four layers Figure (5.2). The first layer is RCP and provides the architecture and the framework to build a rich client application. The IDE layer provides tools to various forms of tooling. The JDT is the complete Java platform and the PDE layer offers all the tools necessary for the development of plug-ins and RCP applications.

## 5.2 Conflict Detector Plug-in

The plug-in developed in this master thesis, is a small prototype that can detect some AOP conflicts. The plug-in was built using some packages that the Eclipse IDE offers, and by extending the AJDT the plug-in that enables the development of aspect-oriented software using AspectJ in Eclipse.

Not all the conflicts that are present in the catalogue, are detectable by the plug-in, but only three types of conflicts are detected by the tool.

The first conflict is the problem with shared join points, which happens when there are aspects with equal join points that are advised by the same type of advice are there isn't any declared order of execution.

The second type of conflict that is detected is when one aspect changes the control flow of a program i.e. it happens when one aspect has an around advice, without the proceed statement, that advises one method that as call to another method that is advised

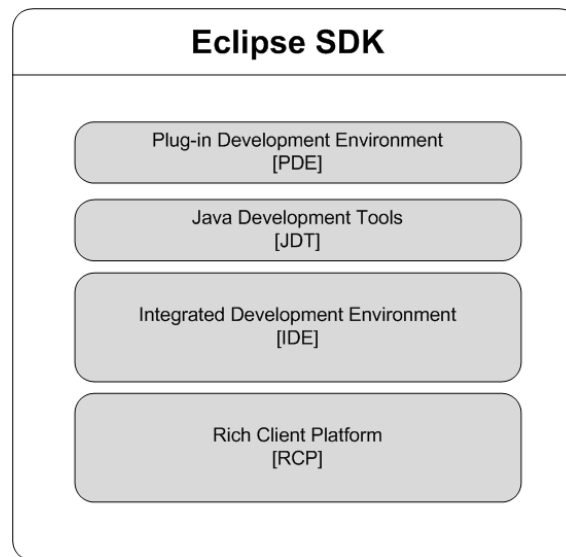


Figure 5.2: Eclipse SDK layers. Adapted from [iS09].

some other aspect. This situation will ignore the call to the method that is advised in the second aspect.

The last type of conflict that is detect in when one introduces a new method in a class, that is a child class of another class that declares another method with the same name. This may be intended but due to the obliviousness property of AOP applications it's always good to check the introductions made so far.

### 5.2.1 Design and Implementation

This section will explain some of the JDT classes that were used in the implementation of the plug-in as well as the important design decisions made.

The JDT offers several types of useful packages and classes. For this prototype it was used the *ASTParser* class. This class can build the Abstract Syntax Tree (AST) of a Java program, which can process entire Java files as well as small portions of code snippets. Listing 5.1 shows the implementation of the function that parses the project. With this model of AST of the Java program, one can retrieve crucial information about its source code. This will get more detailed information on Java classes. For aspects source there too a package that deals specifically with AspectJ code and will be explained further in this section.

Listing 5.1: Method CompilationUnit

```

1 private static CompilationUnit parse(CompilationUnit unit) {
2     ASTParser parser = ASTParser.newParser(AST.JLS3);
3     parser.setKind(ASTParser.K_COMPILATION_UNIT);
4     parser.setSource(unit);

```

## Prototype

```
5  parser.setResolveBindings(true);
6  return (CompilationUnit) parser.createAST(null);
7 }
```

But in order to get the class name, list of the methods, parent class name and other basic information, we need to use the Visitor Pattern [GHJV03]. This is done by extending *ASTVisitor* class and use one of its methods to retrieve the information needed for future analysis. Listing 5.2 shows the implemented class.

Listing 5.2: ClassVisitor

```
1
2 public class ClassVisitor extends ASTVisitor {
3     MethodDeclaration[] methods ;
4     String name;
5     String superClass= null;
6
7     @Override
8     public boolean visit(TypeDeclaration node) {
9
10        methods=node.getMethods();
11        name= node.getName().toString();
12        Type superType = node.getSuperclassType();
13        if (superType != null)
14            superClass=superType.toString();
15
16        return false;
17    }
18 }
```

To parse a complete AspectJ file the same class *ASTParser* can build up the AST of any source code file, but the visitor class is the one that is more specific to AspectJ files. The *AjASTVisitor* can retrieve all the necessary information about an aspect, like its name, advices, pointcuts and a list of methods, and more in particular the inter type method declaration that are important for comparison. Listing 5.3 presents the implemented solution for this plug-in.

Listing 5.3: AspectVisitor

```
1 public class AspectVisitor extends AjASTVisitor {
2     private AspectInfo aspectInfo;
3     public AspectInfo getAspectInfo() {
4         return aspectInfo;
5     }
6     @SuppressWarnings("unchecked")
7     public boolean visit(TypeDeclaration node) {
8         if ((AjTypeDeclaration) node).isAspect() {
9             AspectDeclaration a = (AspectDeclaration) node;
10            List<AdviceDeclaration> listAdvices = a.getAdvice();
11            PointcutDeclaration[] listPointcuts = a.getPointcuts();
```

## Prototype

```
12     MethodDeclaration[] listMethods = a.getMethods();
13     aspectInfo = new AspectInfo(listAdvices, listaPointcuts,
14         listMethods, a.getName().toString());
15 }
16 return false;
17 }
18 }
```

One final step in the analysis of the source code is getting the crosscutting information. This is done by building the model of the program using the class *AJProjectModelFactory*. This model contains all the relationships of each element of the source code. The crosscutting information is used to find out which methods are advised and which advices advise them. The only flaw that this class has is that it doesn't associate the advice with the pointcut, one member of the AJDT development team informed that this information should be implemented in a near future.

All the information that is retrieved from the source files is stored in array lists for further analysis to find out if there are any of the conflicts that the plug-in is able to detect.

We must refer to the three major methods in the plug-in. These methods are the who have the function of analyzing the arrays of information about the project. Each method is explained in the following list:

- **getSharedJoinPoints:** This method, searches for shared joinpoints in the project. It will look in the array that contains the relationships between advices and methods. If it discovers that an aspect contains the same type of advice as of another aspect and that this advice crosscut the same method and there is no information about the order of execution of the two aspects, then we are in the presence of a conflict.
- **lookForAdviceAround:** The method will look in for any around advices in the aspects. If it finds any around advice, it will look in its block statements if there is a call on the proceed method. If there is no such call, then this method will call on another auxiliary method that will search in the body of the crosscut method if there is any call of a method that is advised by any other aspect. If it finds a call to a method that is also advised by another aspect, then it has found a conflict of the type Advice Interference.
- **checkIntroductions:** This method searches for conflicts of the type Incorrect Method Call. First it searches on an array that contains information about the aspects. If it finds in this array that an aspect introduces a new method on a class, it will look in if the class is a subclass. If the class is indeed a subclass, then it will search in its parent class if there is any method with the same as the new added method. If no method is found then there is no conflict, otherwise a conflict is detected.

### 5.3 Plug-in Demonstration

This section will show a small demonstration of the plug-in. This will demonstrate how to select the plug-in from the Eclipse interface and how the conflicts are viewed in the view interface.

The first thing to do is to build the project because we need the built model of the application with the crosscutting information. This can be done by two ways. The first is by choosing in the Eclipse menu *Project-> Build Project* Figure 5.3.

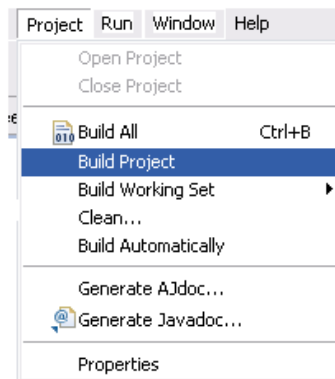


Figure 5.3: Build project on menu bar.

The second option is by right clicking on an advice marker in the Eclipse IDE a popup menu will appear and then choose *AspectJ References-> Build project to generate references* like in Figure 5.4 shows.

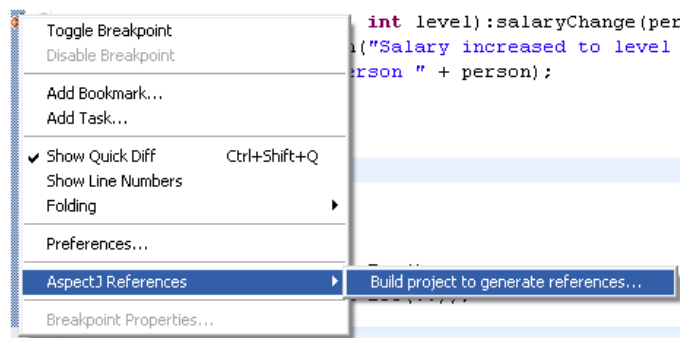


Figure 5.4: Build project on popup menu.

## Prototype

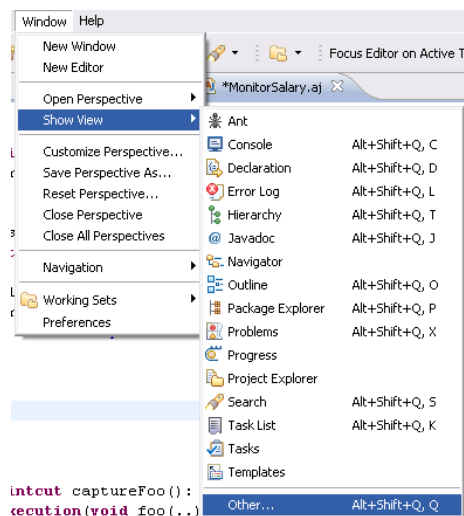


Figure 5.5: Open views menu.

The plug-in is an Eclipse view that will show a table containing two fields, the first filed Type will indicate the type of conflict and the field Conflict will give a small description where the conflict occurs. To open the view we have to go to the Eclipse menu and choose *Window-> Show View-> Other* Figure 5.5 . The next the is to choose the view from the category "AOP Conflicts" Figure 5.6.

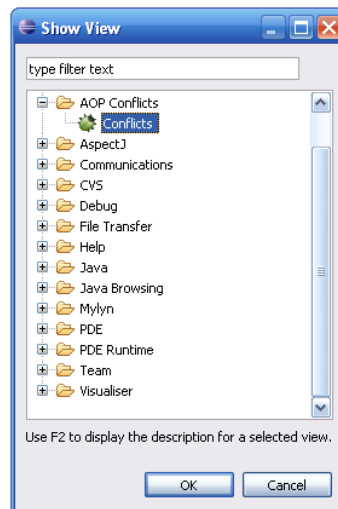
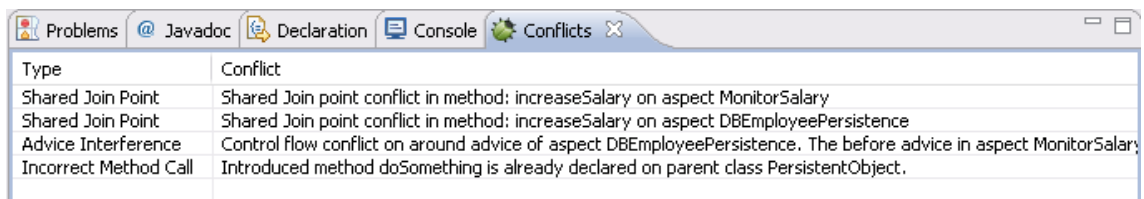


Figure 5.6: Open conflict detector view.

## 5.4 Execution Results

In this example the project contains the three conflicts, the shared join point conflict, advice interference conflict and the incorrect method call conflict. Figure 5.7 shows how conflicts are seen in the plug-in. The example consists in a class *Employee*, its parent class *PersistentObject* and two aspects that are applied to the *Employee*, aspect *MonitorSalary* and aspect *DBEmployeePersistence*.



Type	Conflict
Shared Join Point	Shared Join point conflict in method: increaseSalary on aspect MonitorSalary
Shared Join Point	Shared Join point conflict in method: increaseSalary on aspect DBEmployeePersistence
Advice Interference	Control flow conflict on around advice of aspect DBEmployeePersistence. The before advice in aspect MonitorSalary
Incorrect Method Call	Introduced method doSomething is already declared on parent class PersistentObject.

Figure 5.7: Conflict detector view.

The first two lines points to the same conflict, because the conflict is a shared join point, it identifies in which aspects there are the conflicting join points. In this case we have a conflict in aspect *MonitorSalary* and in aspect *DBEmployeePersistence*.

The second conflict is of the type advice interference. The around advice in aspect *DBEmployeePersistence* does not invoke the method proceed. Because method bar that is the method that it's advised by this around advice, has a call to another method that is advised another aspect the second method will not execute and therefore will not be advised.

The third conflict that is detected is the incorrect method call conflict. Class *Employee* extends class *PersistentObject*, and therefore it inherits all his parent class methods. But in this example aspect *MonitorSalary* introduces a method in class *Employee* with the same name of a method that is declared on the parent class of *Employee*. Now this can be intended in this small project, but in a huge company project an due the AOP properties that programmers achieve with this paradigm like the modularity, some programmers might use the same names that others programmers use in their methods.



## 5.5 Summary

This prototype isn't the main objective of this thesis, so it had a small window for development and for that not all functionalities are implemented. The main goal with this prototype was the beginning of the development of a tool to detect more conflicts than those that it detects for now. With this prototype other researchers have a starting point to begin new studies on AOP interactions and the detection of these interactions.

Some work has still to be done around the prototype to become more feasible. For example, the plug-in doesn't build the project in order to get the crosscutting information that it need to check the relations between advices and methods and this is one of the possible new features that one can add to the prototype. Because of some limitations on the AJDT source code some features have still to be implemented, because when the prototype gets the relations between advices and methods it does not inform which pointcut is referred in the advices.

The prototype accomplishes the objectives that were proposed for it. It detects at least three conflicts concerning the AOP interactions and gives a good idea how a tool like this one can become one day.

Prototype

## **Chapter 6**

# **Conclusions and Future Work**

This chapter will summarize all the contributions made in this thesis. It will be explained the satisfaction of the objectives and the results that we obtained in this dissertation.

Following that, we will present some suggestions on future work that can be done to improve the catalog as well for the prototype.

### **6.1 Objectives Satisfaction**

The main objective for this thesis was the writing of a catalog describing the AOP conflicts that one can find in software applications of this kind. AOP conflicts are the result of interferences that happen between aspects. These interferences arise are when one aspect changes the behavior of another aspect or even prevent the execution of other aspect. These kinds of problems have been a serious obstacle in the evolution of AOP, because if a programming paradigm has so many unsolved issues its clear that no software company will adopt it.

However there is a lot of work done in this area. Several authors have studied how aspects interact with each other, and have classified them according to their own personal criteria and thoughts. Some of the authors have even contributed with interference detection techniques to identify them. Despite the good ideas behind the techniques, some cannot differentiate an intended interaction from an unintended one and that one major drawback when we are looking for unexpected interactions.

As said earlier that other researchers in the field have studied how aspects interact, it was imperative to investigate as much as possible all the different classifications system that exist. This was required to understand how aspects could interfere with each other. The classifications of the interactions may vary according to the author's point of view. Some classify the interactions purely the way how aspects influence the behavior of other

aspects. Some authors classify the interactions according to the read and write access that aspects may have on object fields. Other classification system classify interactions according to the action that advices take on the crosscut methods.

### 6.1.1 Catalogue

The catalogue has identified eight types of conflicts that usually occur in AOP programs. To give a good overview on how conflicts could appear we have select six fields to document each conflict identified. The first field is the name of the field, it has the purpose of identifying the conflict the description field will give a short summary on what is this conflict about. The context field will explain in which conditions a conflict of that type will occur i.e. which recurring scenarios that usually lead to a conflict. A small code example is also given as a description of the conflict, this helps the reader to really see on the source code how the conflict looks. Finally the last field tries to suggest how the conflict can be fixed or even if it is possible introduce a preventive measure to avoid the conflict.

One of the main conclusions that we get after the reading of the catalogue is that there are three major causes that origin conflicts. Most of the conflicts have their origin when aspects share join points, other conflicts appear when two aspects have read and write access to the same object fields. The other cause happens when one changes the structure of a class by introducing new methods or by changing its inheritance.

### 6.1.2 Prototype

The other objective for this thesis was the development of a prototype to detect the conflicts in AOP projects. The prototype is an Eclipse plug-in that uses some of the Eclipse PDE packages as some from AJDT the AspectJ extension for Eclipse. The plug-in does a static analysis of the code, in order to collect information about classes and aspects.

With the information gathered during the analysis it was possible to detect three types of conflicts. The conflicts that the plug-in detects are the shared join point conflict, the advice interference conflict and the incorrect method call conflict.

## 6.2 Future Work

As all projects of investigation there is always room for more improvements. We do not think that all the conflicts that were documented in the catalog are the only ones that exist. There are of course more conflicts out there that can be very similar to the ones in the catalogue. There are many ways in combining several aspects functionalities that can interfere with each.

A knowledge database containing these conflicts could be developed. This database could be used by several AOP developers, as a reference guide to study when some issue arises in their projects. The database itself could work as a wiki where all could contribute in order to gather as much information as possible about conflicts in AOP.

The prototype, as said before, detects three of the eight conflicts documented in the catalogue. This of course isn't enough for debugging real software projects. As an improvement of the prototype we could consider the detection of other conflicts.

Due to some limitation on the AJDT some functions are not quite what we should expect. When the prototype gets all the crosscutting information from an AOP project it is unable to associate each advice to its corresponding pointcut, this was confirmed by an AJDT development team member. Without this information we cannot indicate which pointcut is the conflicting one. But this will be implemented soon as they stated.

### **6.3 Concluding Remarks**

This dissertation studied the unexpected interactions between aspects. We documented some of the most recurrent issues that arise in AOP applications. With the catalogue we expect to give to the community a guide to help prevent these unwanted scenarios.

The prototype is not a complete tool so its need to improve in some points but we think it is a good starting point to a more mature and robust testing tool for software projects.

## Conclusions and Future Work

# References

- [AG95] Alfred V. Aho and Nancy D. Griffeth. Feature interactions in the global information infrastructure. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 2–5, 1995.
- [ARS09] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of the 8th ACM international conference on Aspect-oriented Software Development*, pages 39–50, 2009.
- [Bae06] Jon Swane Baekken. A fault model for pointcuts and advice in aspectj pprograms. Master’s thesis, Washington State University, 2006.
- [BE07] Shubhanan Bakre and Tzilla Elrad. Scenario based resolution of aspect interactions with aspect interaction charts. In *Proceedings of the 10th international workshop on Aspect-Oriented Modeling*, pages 1–6, 2007.
- [BM04] Davide Balzarotti and Mattia Monga. Using program slicing to analyze aspect oriented composition. In *Proceedings of the Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, 2004.
- [BM07] Jorge Barreiros and Ana Moreira. Aspect interaction management with meta-aspects and advice cardinality. In *Proceedings of the European Conference on Object Oriented Programming, 2007 - Aspects, Dependencies, and Interactions Workshop*, 2007.
- [CL02] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect Languages*, pages 33–44, 2002.
- [CPSM07] Sandra I. Casas, J. Baltasar García Perez-Schofield, and Claudia A. Marcos. Associations in conflict. *Journal of Computer Science*, pages 27–36., 2007.
- [CRK<sup>+</sup>08] Roberta Coelho, Awais Rashid, Uira Kulesza, Arndt von Staa, Carlos Lucena, and James Noble. Exception handling bug patterns in aspect oriented programs. *PLOP 2008*, 2008.
- [DBA05] Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about semantic conflicts between aspects. In *EIWAS 05: The 2nd European Interactive Workshop on Aspects in Software*, pages 10–18, 2005.

## REFERENCES

- [DBA07a] Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about behavioral conflicts between aspects. Technical Report TR-CTIT-07-15, University of Twente, 2007.
- [DBA07b] Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit. Static and dynamic detection of behavioral conflicts between aspects. In *Proceedings of the Seventh International Workshop on Runtime Verification*, pages 38–50, 2007.
- [DD02] Maja DHondt and Theo DHondt. The tyranny of the dominant model decomposition, 2002.
- [DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT*, pages 173 –188, 2002.
- [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*, pages 141 – 150, 2004.
- [FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35, 2000.
- [Fil01] Robert Filman. What is aspect-oriented programming, revisited. In *Workshop on Advanced Separation of Concerns*, 2001.
- [FSJ08] Bruno De Fraine, Mario Südholt, and Viviane Jonckers. Strongaspectj: Flexible and safe pointcut/advice bindings. In *Proceedings of AOSD’08*, 2008.
- [FT06] Paolo Falcarin and Marco Torchiano. Automated reasoning on aspects interactions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 313 –316, 2006.
- [GHJV03] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2003.
- [GSS09] Rüdiger Kapitza Guido Söldner and Sven Schober. Aoci: Ontology-based pointcuts. In *Proceedings of the 8th workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 25–30, 2009.
- [Gui09] The AspectJ Programming Guide.  
<http://www.eclipse.org/aspectj/doc/released/progguide/index.html>, Accessed on February of 2009.
- [HLH08] Chengwan He, Zheng Lia, and Keqing He. Using conceptual model and reflection mechanism to resolve the structural conflict in aop application. *International Conference on Computer Science and Software Engineering*, pages 77–80, 2008.



## REFERENCES

- [HNBA06] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Detecting and resolving ambiguities caused by interdependent introductions. In *Proceedings of the 5th international conference on Aspect-oriented Software Development*, pages 214–225, 2006.
- [HNBA07] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 85–95, 2007.
- [iS09] Eclipse Plug in Site. <http://www.eclipsepluginsite.com/>, Accessed on June of 2009.
- [Kat04] Shmuel Katz. Diagnosis of harmful aspects using regression verification. 2004.
- [KDEG06] Jörg Kienzle, Ekwa Duala-Eboko, and Samuel G lineau. Aspectoptima: A case study on aspect dependencies and interactions. *AOSD*, 2006.
- [KFG04] Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In *Proceedings of the 12th ACM SIGSOFT*, pages 137–146, 2004.
- [KHH<sup>+</sup>01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, 2001.
- [KK08] Emilia Katz and Shmuel Katz. Incremental analysis of interference among aspects. In *Proceedings of the 7th workshop on Foundations of Aspect-Oriented Languages*, pages 29–38, 2008.
- [KLM<sup>+</sup>97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. 1997.
- [KM05] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. *ECOOP 2005 - Object-Oriented Programming*, pages 195–213, 2005.
- [Kni09] G nter Kniesel. Detection and resolution of weaving interactions. *Transactions on Aspect-Oriented Software Development*, 2009.
- [KYX03] J rg Kienzle, Yang Yu, and Jie Xiong. On composition and reuse of aspects. In *Foundations of Aspect Languages*, 2003.
- [Lad03] Ramnivas Laddad. *AspectJ in Action Practical Aspect-Oriented Programming*. Manning, 2003.
- [MBB08] Freddy Munoz, Benoit Baudry, and Olivier Barais. A classification of invasive patterns in aop. *INRIA*, 2008.

## REFERENCES

- [McC05] Edward J. McCormick. Pointcuts by example. Master’s thesis, The University Of British Columbia, 2005.
- [Mil04] Russ Miles. *AspectJ Cookbook*. Oreilly & Associates Inc, 2004.
- [Mon05] Miguel Jorge Tavares Pessoa Monteiro. *Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts*. PhD thesis, Universidade do Minho, 2005.
- [MWA08] Gunter Mussbacher, Jon Whittle, and Daniel Amyot. Towards semantic-based aspect interaction detection. *1st International Workshop on Non-functional System Properties in Domain Specific Modeling Languages*, 2008.
- [NBA04] István Nagy, Lodewijk Bergmans, and Mehmet Aksit. Declarative aspect composition. In *2nd Software Engineering Properties of Languages and Aspect Technologies*, 2004.
- [NBA05] Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing aspects at shared join points. *NetObjectDays*, 2005.
- [PDS05] Renaud Pawlak, Laurence Duchien, , and Lionel Seinturier. Compar: Ensuring safe around advice composition. *Formal Methods for Open Object-Based Distributed Systems*, 3535:163–178, 2005.
- [PHPP06] Eduardo Kessler Piveta, Marcelo Hecht, Marcelo Soares Pimenta, and Roberto Tom Price. Detecting bad smells in aspectj. *Journal of Universal Computer Science*, 12:811–827, 2006.
- [PTC00] J. Andres Diaz Pace, F. Trilnik, and Marcelo R. Campo. How to handle interacting concerns? In *Workshop on Advanced for Separation of Concerns in OO Systems, OOPSLA 2000*, 2000.
- [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*, 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*, 2008.
- [RSB04] Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara. A classification system and analysis for aspectoriented programs. In *Proceedings of the 12th Symposium on the Foundations of Software Engineering*, pages 147–158, 2004.
- [SB06] Sergio Soares and Paulo Borba. Aspectj programação orientada a aspectos em java. *NovaTec*, 2006.
- [SdM03] Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proceedings of the 2nd international conference on Aspect-Oriented Software Development*, pages 30 –39, 2003.

## REFERENCES

- [SK04] Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem. *European Interactive Workshop on Aspects in Software*, 2004.
- [SLL03] Therapon Skotiniotis, Karl Lieberherr, and David H. Lorenz. Aspect instances and their interactions. In *Proceedings of International Conference on Aspect-Oriented Software Development*, 2003.
- [SSF06] Maximilian Störzer, Robin Sterr, and Florian Forster. Detecting precedence-related advice interference. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 317–322, 2006.
- [STJ<sup>+</sup>06] Frans Sanen, Eddy Truyen, Wouter Joosen, Andrew Jackson, Andronikos Nedos, Siobhán Clarke, Neil Loughran, and Awais Rashid. Classifying and documenting aspect interactions. In *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 23–26, 2006.
- [STJ08] Frans Sanen, Eddy Truyen, and Wouter Joosen. Modeling context-dependent aspect interference using default logics. *ECOOP Workshop on Reflection*, 2008.
- [SZZ<sup>+</sup>08] Haihao Shen, Sai Zhang, Jianjun Zhao, Jianhong Fang, and Shiyuan Yao. Xfindbugs: extended findbugs for aspectj. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT*, pages 70–76, 2008.
- [WTR07] Nathan Weston, Francois Taiani, and Awais Rashid. Interaction analysis for fault-tolerance in aspect-oriented programming. *MeMoT 2007*, 2007.
- [ZZ07] Sai Zhang and Jianjun Zhao. On identifying bug patterns in aspect-oriented programs. *COMPSAC’07*, 1:431–438, 2007.