On the Design of Aspect-Oriented Composition Models for Software Evolution

István Nagy

On the Design of Aspect-Oriented Composition Models for Software Evolution

István Nagy

Dissertation Committee

Chairman and secretary:

Prof. Dr. Ir. A.J. Mouthaan	University of Twente, The Netherlands

Promotor:

Prof. Dr. Ir. M. Akşit	University of Twente, The Netherlands

Assistant-promotor:

Dr. Ir. L.M.J. Bergmans	University of Twente, The Netherlands

Members:

Prof. Dr. P.H. Hartel	University of Twente, The Netherlands
Prof. Dr. Th. D'Hondt	Vrije Universiteit Brussel, Belgium
Prof. Dr.-Ing. M. Mezini	University of Technology Darmstadt, Germany
Dr. Ir. A. Rensink	University of Twente, The Netherlands
Dr. M. Südholt	École des Mines de Nantes-INRIA, France

István Nagy
On the Design of Aspect-Oriented Composition Models for Software Evolution
PhD Thesis, University of Twente, 2006
ISBN 90-365-2368-0

The work in this thesis has been carried out within the context of the Centre for Telematics and Information Technology (CTIT).

CTIT Ph.D.-thesis Series, no. 06-88
CTIT, P.O. Box 217, 7500 AE, Enschede, The Netherlands
ISSN: 1381-3617

Printed by Febodruk BV, Enschede, The Netherlands.
Cover design by István Nagy

On the Design of Aspect-Oriented Composition Models for Software Evolution

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 8 juni 2006 om 15.00 uur

door

István Nagy

geboren op 30 juli 1977
te Karcag, Hongarije

Dit proefschrift is goedgekeurd door:

Prof. Dr. Ir. M. Akşit, promotor
Dr. Ir. L.M.J. Bergmans, assistent-promotor

“Every journey brings its own surprises: a challenge, a sudden detour, a new set of friends along the way, perhaps even a destination different from the one you intended.”

- Loreena McKennitt

To the loving people with whom I was along on this journey

Acknowledgements

This thesis is the result of 4 years of work and experience in my life. This was an important journey whereby I was accompanied and supported by many people. Below I would like to present my words of gratitude and appreciation for all of them.

First, I thank my promoter, Mehmet Akşit, for giving an opportunity to learn and do research in his group. He provided me fruitful and productive working conditions, and numerous contacts in the international research community. It was very important to me that he kept a critical eye on my research; his support, guidance and feedback have all been of great value.

My assistant-promoter, Lodewijk Bergmans, was involved in my work from the very beginning. I could hardly forget the countless discussions with him, including those ones when he was challenging my research ideas. I would like to thank him for his ever-lasting patience to deal with me, and teaching me how to develop and present my work. Lodewijk has been much more than just a supervisor for me. He has been a true friend with whom we could share the difficult moments as well as the joy of not only the research, but many other activities. I am happy that I have come to get know him in my life. I also thank the support of his wife, Ingrid, for making good meals and taking care of us before the important submission deadlines.

I thank Pieter Hartel, Theo D'Hondt, Mira Mezini, Arend Rensink and Mario Südholt for assessing my manuscript as members of my graduation committee; thanks for the great effort in reading the earlier draft and providing constructive and valuable comments on my work.

My colleagues in the Software Engineering Group have always created a friendly and pleasant working environment. Of the members of my group I would like to mention a few in particular. I wish to thank Klaas van den Berg and Bedir Tekinerdogan for their useful feedback on my work. In addition, I could always bother Klaas about bureaucratical issues that nobody else could

know in my group; thanks for his patience. Special thanks to a former room-mate of mine, Joost Noppen. Despite having very different research topics, Joost and I always had fruitful discussions on each others work. Besides, he helped me to handle many problems caused by my deficiencies on the Dutch language. I had enjoyable time and discussions with my new PhD fellows, Gürcan, Pascal and Wilke too. Thanks for the good companion and meals that we have made together during the 'Gang of Five plus One' dinners.

I am grateful to the secretaries of my group, Ellen and Joke for their infinite patience towards me and taking care of all the administrative work.

I also wish to thank my undergraduate supervisor, István Juhász who helped me in taking the first important steps on the field of software engineering and in the world of aspect-oriented programming.

I consider myself lucky to have been able to make new friends along the way. I would like to say a big thanks to two very special friends of mine, Gürcan and Marloes who were always there for me, shared all good and difficult moments, and helped me to overcome many difficulties. I also thank Mariëlle, Pieter and Tomas for the enjoyable dinners, discussions and the fun we had together as well as their help in many issues.

I had pleasant times with many Hungarian friends as well. I thank Róbert, Tímea, Szabi, Dela, Roland, Kati, Szilárd, Mária, Andris és Saci for all the great time we spent together.

Furthermore, I thank all of my friends who made my stay in the Netherlands pleasant and helped me in one way or another.

Last but not least, I would like to thank my family. My sister, brother and parents supported me from far away throughout the years. *Köszönöm!*

István Nagy,
May 15, 2006
Enschede, The Netherlands

Abstract

Aspect-oriented programming is an emerging approach in software development, which provides new possibilities for separation of concerns. Aspect-oriented languages offer abstractions for the implementation of concerns whose modularization cannot be achieved by using traditional programming languages. Such concerns are generally termed as crosscutting concerns. It is generally agreed that separating the right concerns from each other enhances software quality factors such as reusability and adaptability. The separated concerns in software must be composed together so that software behaves according to its requirements in a coherent way. We refer to language mechanisms that separate and compose concerns as 'composition mechanisms'. This thesis evaluates the software composition mechanisms of current aspect-oriented languages from the perspective of software quality factors such as evolvability, comprehensibility, predictability and adaptability. Based on this study, the thesis proposes novel extensions to current aspect-oriented languages so that programs written in these languages exhibit better quality.

A considerable number of aspect-oriented languages has been introduced for modularizing crosscutting concerns. Naturally, these languages share a number of common concepts and have distinctive features as well. For this reason, we propose a reference model that aims to capture the common and distinctive concepts of aspect-oriented languages. This reference model provides a basis to understand the important characteristics of the state-of-the-art AOP languages and helps us to compare the AOP languages with each other. Furthermore, it exposes the issues that have to be considered when a new aspect-oriented language needs to be developed.

In this thesis, we analyse the four main aspect-oriented concepts of the reference model, namely join point, pointcut, advice and aspect, and identify problems related to their use in various AOP languages. Based on this analysis, we propose extensions of the existing concepts and/or design new ones to address the identified problems.

In current aspect-oriented languages, pointcuts select join points of a program based on lexical information such as explicit names of program elements. However, this reduces the adaptability of software, since it involves too much information that is hard-coded, and often implementation-specific. We claim that this problem can be reduced by referring to program elements through their semantic properties. A semantic property describes for example the behavior of a program element or its intended meaning. We formulate requirements for the proper application of semantic properties in aspect-oriented programming. We discuss how to use semantic properties for the superimposition of aspects, and how to apply superimposition to bind semantic properties to program elements. To achieve this, we propose language constructs that support semantic composition: the ability to compose aspects with the elements of the base program that satisfy certain semantic properties.

The current advice-pointcut binding constructs of AOP languages maintain explicit dependencies to advices and aspects. This results in weaving specifications that are less evolvable and need more maintenance during the development of a system. We show that this issue can be addressed by providing associative access to advices and aspects instead of using explicit dependencies in the weaving specification. To this aim, we propose to use a designating (query) language in advice-pointcut bindings that allows for referring aspect/advices through their (syntactic and semantic) properties. We also present how semantic properties can be applied to provide reusable (adaptable) aspect abstractions.

Aspect-oriented languages provide means to superimpose aspectual behavior – in terms of advices - on a given set of join points. It is possible that not just a single, but several advices need to execute at the same join point. Such "shared" join points may give rise to issues such as determining the exact execution order and the other possible dependencies among the aspects. We present a detailed analysis of the problem, and identify a set of requirements upon mechanisms for composing aspects at shared join points. To address the identified issues, we propose a general and declarative model for defining constraints upon the possible compositions of aspects at a shared join point. By using an extended notion of join points, we show how concrete aspect-oriented programming languages can adopt the proposed model.

The thesis also presents how the proposed extensions and new constructs are adopted by the aspect-oriented language Compose*. To evaluate the proposed constructs, we provide qualitative analyses with respect to various software engineering properties, such as evolvability, modularity, predictability and adaptability.

Table of Contents

Chapter 1

Introduction

1.1 Synopsis

Aspect-oriented programming is an emerging approach in software development, which provides new possibilities for separation of concerns. We adhere the commonly used definition of concern from IEEE 1471: *a concern is an interest which pertains to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.* Aspect-oriented languages offer abstractions for the implementation of concerns whose modularization cannot be achieved by using traditional programming languages. Such concerns are generally termed as crosscutting concerns. It is generally agreed that separating the right concerns from each other enhances software quality factors such as reusability and adaptability. Obviously, the separated concerns in software must be composed together so that software behaves according to the requirements in a coherent way. We will briefly refer to language mechanisms that separate and compose concerns as the composability features of the language. This thesis evaluates the software composition mechanisms of current aspect-oriented languages from the perspective of software quality factors such as evolvability, comprehensibility, predictability and adaptability. Based on this study, the thesis proposes novel extensions to current aspect-oriented languages so that programs written in these languages exhibit better quality.

This chapter is organized as follows: section 1.2 provides background on software quality factors and their relevance to the design of programming languages. Section 1.3 outlines the problem of crosscutting concerns by a simple example. Sections 1.4 and 1.5 show two possible solutions to modelling

crosscutting concerns are presented: proxy-based and aspect-oriented language based. This section evaluates these two alternatives. Although aspect-oriented languages provide better modularization for crosscutting concerns, they may fall in short in case requirements evolve in unpredictable ways. Section 1.6 elaborates on such composability problems. Section 1.7 summarizes the contributions to overcome these problems and provides an overview about the structure of this thesis.

1.2 Software Quality Factors

Software developers squarely agree that providing high-quality software is an important goal. In software engineering, quality can be expressed by so called 'ilities', such as *usability*. *reliability*, *adaptability*, for instance. (An extensive set of these ilities is listed (and discussed) in [13], in chapter 19.) These ilities represent important quality aspects of software.

Nowadays software is developed by sophisticated tools, environments and high-level programming languages. The adopted programming language and the related tools directly affect the quality of software. This thesis evaluates current aspect-oriented languages with respect to the quality values evolvability, comprehensibility, predictability, and adaptability[1]. After defining these quality factors, we explain how aspect-oriented languages may impact these quality factors in practice.

1.2.1 Evolvability

Evolvability is an important software quality factor that indicates the ability of developing programs *incrementally*. In general, evolvability facilitates extending an application towards new requirements mostly by reusing previously written modules without modifying them.

The inheritance construct adopted by object-oriented languages is an example of a language construct that contributes positively to evolvability. Inheritance is basically a composition abstraction that allows for incrementally extending (and refining) a definition of class with a new one.

1. Of course, one may define many more ilities; we have chosen the above mentioned ilities because they are related to composition mechanisms of programming languages, which is our focus of interest.

1.2.2 Predictability

Predictability ensures programmers that certain properties are held during the development and execution of a program. In general, there is a wide range of features that supports predictability in programming languages. For example, in Java, if the keyword final is used, the definition of a class cannot be extended and/or overridden using inheritance. Although this hampers evolvability, it may positively contribute predictability since a class which is defined as final cannot be extended and modified in unpredictable ways by its subclasses. Other language constructs that enhance predictability are for example type and interface declarations. These language mechanisms obviously require adequate compiler and/or run-time support.

1.2.3 Comprehensibility

We define comprehensibility as the ability to understand the meaning of a program by just looking at its source code. Comprehensibility can be influenced by the programming style and the constructs of the adopted language, such as how the program units are modularized, how the units refer to each other, and where the references in the units are specified.

For example, if utilized appropriately, the notion of classes in object-oriented programming enhances comprehensibility of software because classes have well-defined syntactic structure and provide linguistic constructs for the separation of interface from implementation.

1.2.4 Adaptability

Certain language constructs may ease implementing software modules that can adapt to changing context. (Adaptability is often related to configurability.) Various language constructs that support parametrization can positively contribute to the adaptability of programs.

For example, in object-oriented programming languages, the operations that are offered by a class may be adapted by initializing them with different arguments. Generic types are another example for adapting software. For example, template classes in C++ enable instantiation of classes with different types of local variables.

1.2.5 Trade-offs and Relations between Software Quality Factors

As indicated in the previous section, the final keyword (i.e. language abstraction) has influence both on evolvability and predictability. Language constructs usually contribute to more than just one quality factor. For instance, as discussed in the previous section, the keyword final has influence both on evolvability and predictability of programs. Moreover, certain software engineering properties often have 'synonym' or 'antonym' type of relationships with each other. For example, the language constructs that provide better adaptability often decrease the predictability of the programs, and the other way around, better predictability can often result in less evolvability of programs. Also, predictability is often considered as a feature of comprehensibility in [4].

Preferably, the designers of a language should deliberately consider the impact of introducing a new language construct on software quality factors. Obviously, this was the case when the keyword final was introduced. The purpose of final is to provide better predictability by limiting evolvability. However, a language construct may contribute (both in a negative and positive manner) to more software quality factors than the ones that were involved in its design.

We therefore claim that the trade-off analysis of various language constructs from the perspective of multiple software quality factors plays an important role in evaluating and comparing programming languages with each other

For further discussions on the relation between language mechanisms and software quality factors, the interested reader can refer to [4].

1.3 Crosscutting Concerns

Programming languages are the primary means in the implementation phase of software development processes: they act as a specification language that is used to describe the *concerns* of the software system being developed. Many different programming languages have been introduced in the past. Among these, the object-oriented languages retain their popularity. In these languages, the notion of "object" is a primary means to represent concerns. Every object has an explicit identity with a set of associated operations. Objects may encapsulate other objects. The internal structure of an object is abstracted through the set of operations exposed at its interface. Various publications in the literature

[3, 1] have shown that there are certain concerns that cannot be adequately represented as a single object. We take the familiar example concern of tracing of changes of figure elements [3], and show its implementation in Java. Assume, for instance, that we have two main concerns in the implementation of the graphical elements shown in Figure 1.1. First concern is to represent the graphical elements as explicit modules. The second concern is to trace changes of the coordinate values of the graphical elements.

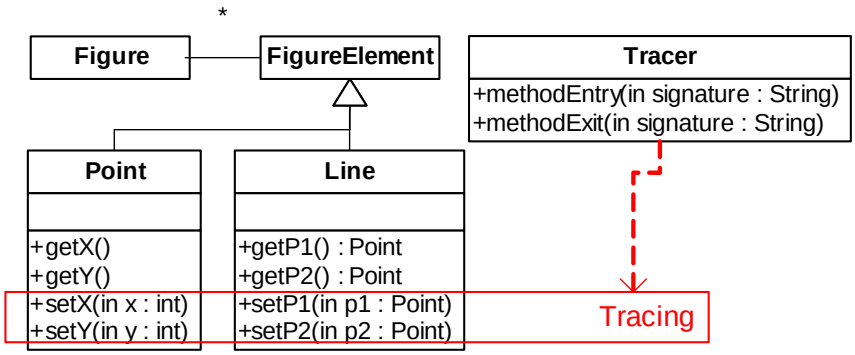

Figure 1.1 *An example crosscutting concern: Tracing*

In this figure, classes Figure, FigureElement, Point and Line are program modules that represent the graphical elements explicitly; thereby, they fulfil the first concern. Since every graphical element is represented as a class, they can be reused and extended individually using the inheritance and aggregation mechanisms of object-oriented languages. In addition, implementing every graphical element explicitly as a single language module possibly contributes to the comprehensibility of the program. Similarly, for the purpose of reusability, extendibility and comprehensibility, the concern of tracing is implemented by class Tracer. However, the implementation of this concern is not cleanly separated from the implementation of graphical elements. The methods that are used for setting the coordinate values in graphical elements must somehow implement part of the tracing functionality; after every set operation, the tracer

object must be informed accordingly. Here, the implementation of the tracing

```
0) class Point extends FigureElement{
1)    private int x,y;
2)    ...
3)    public setX(int x){
4)       trace.methodEntry("Point.setX, x="+x);
5)       this.x=x;
6)       trace.methodExit("Point.setX");
7)    }
8)    public setY(int y){
9)       trace.methodEntry("Point.setY, y="+y);
10)      this.y=y;
11)      trace.methodExit("Point.setY");
12)   }
13)}
```

Listing 1.1 *Crosscutting Implementation of Tracing*

concern is in fact scattered over classes Point, Line and Tracer. In classes Point and Line, the implementation of the graphical concern is also partially tangled with the implementation of the tracing concern. If a concern is scattered and/or tangled, it is called a crosscutting concern. Let us now look at this problem in more detail. As shown in lines 4 and 9 in Listing 1.1, invoking the methods setX and setY will cause calls on the method methodEntry of trace, which is an instance of class Tracer. This is used by trace to register that a call is made either on the method setX or setY. Similarly, in lines 6 and 11, after setting the x or y value of a point, the method methodExit is called on trace. This will be used by trace to register that either the x or y value of a point has been set. Here, the arguments of the calls methodEntry and methodExit are used to pass the necessary information to trace.

In this listing, parts of the implementation of the concerns “representing the graphical elements” and “tracing changes of the coordinate values of the graphical elements” are in-lined in the implementation of the methods setX and setY. This is generally referred to as code-tangling. As a result, class Point partially implements two concerns at the same time, which should be preferably separated. Here, code-tangling in fact occurs in the methods setX and setY of classes Point and Line, since the implementation of the both concerns are scattered to multiple classes and methods.

Unfortunately, not being able to represent crosscutting concerns explicitly may negatively influence the software quality factors discussed in this chapter. In the following subsections, by using a state-of-the-art, object-oriented software development platform, we will investigate various means in implementing the example shown in Figure 1.1.

1.4 A First Attempt: Dynamic Proxies

It is important to mention that languages such as Java and C# are in fact supported by tailored software development platforms, which are typically equipped with virtual machines that control the execution of programs. Through Application Programming Interfaces (APIs), these platforms may provide various services such as *introspection* and *dynamic class loading*, so that programs can observe and manipulate their execution context effectively. In fact, by using these so-called meta services, it is possible to modularize the crosscutting concerns shown in Figure 1.1. In Listing 1.2, for example, we illustrate how the tracing can be implemented in a standalone module.

The bold lines of Listing 1.2 are used to indicate the statements related to the meta-services of the Java programming environment. Here, class `Tracing` implements the interface `InvocationHandler`. Instances of the interface `InvocationHandler` are responsible for intercepting and dispatching all method calls made to a proxy[2] regular object. The method `invoke(Object proxy, Method method, Object[] args)` of `InvocationHandler` gets executed whenever a method is executed on the proxy. The first argument is the proxy instance that has been invoked, the second argument is the target method, and the third argument

2. A dynamic proxy is a class that implements a list of interfaces, which one specifies at runtime at the time of creation of the proxy. A proxy behaves like any other class that implements the supplied interfaces.

(args) are the runtime parameters that the caller supplies. This method realizes the logic of tracing: first, the entry of the method execution is printed.

```
0) public class Tracing implements InvocationHandler {
1)   protected Object delegate;
2)
3)   public Tracing(Object delegate) {
4)     this.delegate = delegate;
5)   }
6)
7)   public Object invoke(Object proxy, Method method,
8)                        Object[] args) throws Throwable {
9)     try {
10)      System.out.println("Entering the method " + method);
11)      /* ... for loop for printing out the args ... */
12)      Object result = method.invoke(delegate, args);
13)      return result;
14)    } catch (InvocationTargetException e) {
15)      throw e.getTargetException();
16)    } finally {
17)      System.out.println("Exiting the method " + method );
18)    }
19)  }
20)
```

Listing 1.2 *A standalone module (class) in Java that represents Tracing*

In line 12, the original (intercepted) method is executed and its return value is stored in temporary variable. This value is returned in the next statement. In lines 17, the exit of the method execution is printed in the finally clause. The example has been adopted from [2], the reader can find further details about using these services here.

Although the implementation in Listing 1.2 is able to represent the tracing concern as a standalone module, it has two important disadvantages. First, the implementation depends on the API. If the API changes, the implementation may not work as desired[3]. Second, in order to understand the listing, the

3. For backward compatibility, the original methods of an API are always kept in Java, and marked as deprecated so that new implementation can, but should not use the deprecated functions.

programmer must have knowledge on the API of the Java environment. This obviously reduces the comprehensibility of the program.

In Listing 1.3, we will now show how the tracing concern as implemented by class Tracing in Listing 1.2 can be composed with the implementation of the graphical element Point. First, the class Point has to be 'prepared' for the presence of proxy classes. For this reason, an interface (IPoint) is created that contains the signature of every method that can be intercepted by a proxy (see lines 0 to 2). In addition, we need to indicate that class Point implements the interface IPoint (line 5) and declare the exception clause to every method where it is necessary (see for example line 7).The program shown in Listing 1.3 works as follows: first, an instance of IPoint created by the instantiation of the class Point. In line 18, we create an instance Tracing, which is referred to as an instance of InvocationHandler. In line 21, we create a proxy instance of Point and pass in the previously instantiated invocation handler. Finally, we test the proxy instance by calling their method setX on it in line 26.

The implementation shown in Listing 1.3 has two drawbacks. First, in order to compose the tracing concern as implemented in Listing 1.2, the implementation of class Point must be modified considerably. This means that the tracing concern cannot be added to class Point in an incremental way. Second, the solution given in this section requires the availability of the source code of class Point If the source is not available, the same modifications could be also performed on the bytecode level with the help of a proper tool support. Obviously, if the tracing requirement was anticipated before, the implementation of class Point could be prepared accordingly. Unfortunately in reality, not all (future) requirements can be anticipated. These problems get amplified when the other graphical elements are considered as well. This means that for each class which implements a graphical element, a generic interface (lines 0-2 in Listing 1.3) and a binding specification (lines 14-25 in Listing 1.3) must be added. For this purpose, the approach by using dynamic proxies is not considered satisfactory. In general, programming languages can solve the above mentioned problems by providing one or more language constructs to support the proper modularization of concerns. In fact, aspect-oriented programming languages (AOPs) have been developed to model crosscutting concerns explicitly.

```
0) public interface IPoint {
1)   public void setX(int x) throws Exception;
2)   ...
3) }
4)
5) public class Point extends FigureElement implements IPoint {
6)   ...
7)   public void setX(int x) throws Exception {
8)     this.x=x;
9)   }
10)}
11)
12)public class Test{
13)  public static void main(String argv[]){
14)    // Create an instance of Point
15)    IPoint pi = new Point();
16)
17)    // Create InvokeHandler for Tracing
18)    InvocationHandler handler = new Tracing(pi);
19)
20)     //Create a dynamic proxy and pass in Tracing.
21)    IPoint p =
22)      (Point) Proxy.newProxyInstance(
23)        pi.getClass().getClassLoader(),
24)        pi.getClass().getInterfaces(),
25)        handler);
26)    p.setX(5);
27)  }
28)}
```

Listing 1.3 *Applying Tracing on Point*

In the following section, by the help of the aspect-oriented language AspectJ[4] [7], we give a brief introduction to the main concepts of AOP languages. For this purpose, the example problem introduced in Figure 1.1 is used.

4. To be precise, AspectJ is an aspect-oriented language extension to Java.

1.5 A Short Introduction to Aspect-Oriented Languages

This section gives a short introduction to the essential concepts of aspect-oriented languages for the novice reader. We discuss the most important abstractions based on the source code of the example presented in Listing 1.4.

```
0)  public aspect Tracing{
1)    pointcut stateChanges():
2)       call(void Point.setX(int)) ||
3)       call(void Point.setY(int));
4)
5)    Object around(): stateChanges(){
6)       String m_name =
7)          thisJoinPoint.getSignature().getName();
8)
9)       System.out.println("Entering the method " + m_name);
10)      Object temp = proceed();
11)      System.out.println("Exiting the method " + m_name);
12)
13)      return temp;
14)   }
15)}
```

Listing 1.4 *Implementation of the crosscutting concern tracing in AspectJ*

To represent crosscutting concerns as separate modules, AspectJ introduces a novel language construct called *aspect*. Aspects, like ordinary Java classes, can be defined using methods, fields and static initializators. In addition to these, *pointcut* and *advice* are the two additional language constructs that are specifically introduced to define aspects. A pointcut is used to designate certain events in the execution of a program. These events are called join points. Although the definition of join points may differ from one language to another, the events like passing a message, reading a field value of an object, raising an exception are typically considered as join points. Consider, for example lines 1, 2 and 3 of Listing 1.4, which declares the pointcut stateChanges that is used to designate call events on the methods setX and setY of class Point. Advices represent the crosscutting code that is executed at the designated join points. In AspectJ, there are three types of advices: *before*, *after* and *around*[5]. Given a sequence of events, a before advice is executed just before the execution of its

5. There are two additional *after* advices: *after returning* and *after throwing*. We will discuss these in Chapter 2.

designated join point. As the name implies, the after advice is executed just after the termination of its designated join point. An around advice is executed instead of its designated join point. In Listing 1.4, between lines 5 and 14, an around advise is defined. In line 5, the term Object indicates that the execution of this advice will return an instance of type Object. This advice is bound to the join point stateChange, which is defined between lines 1 and 3. This means that the code of this advice is executed when the methods setX or setY of class Point are called. In AOP terminology, class Point is called as a base class. In line 7, the pseudo variable thisJoinPoint is used to denote an object which represents the designated join point. This object provides reflective information about the join point. By calling subsequently on the methods getSignature and getName of this object, it is possible to obtain the name of the method which has been called. The name of the method is then printed using the statement in line 9. In AspectJ, the keyword proceed in line 10 is used to execute the designated join point, which is either calling on setX or setY in this case. The result of this execution is kept in the variable temp of type Object. In line 11, it is reported that the execution of the called method has been completed. Finally, in line 13, the value stored in temp is returned as a result of the execution.

There are three observable differences between the two implementations given in Listing 1.2 and Listing 1.4. First, the AspectJ implementation is shorter and easier to comprehend. But most importantly, in the AspectJ implementation, the tracing concern can be composed with the methods setX and setY of class Point without preparing class Point for this purpose. It is therefore claimed that in the implementation of such crosscutting concerns, the AOP languages provide true incremental composition. Finally, although not illustrated here explicitly, the aspect specification shown in Listing 1.4 can be easily extended to designate any graphical element class.

1.6 Problem Statement

In the previous sections, we have discussed various language constructs that can be used to effectively express the separation and composition of program modules that would otherwise crosscut each other in case non-aspect oriented languages would have been used. Aspect-oriented languages adopt various and different composition possibilities. Through this diversity[6], each languages may emphasize certain software quality factors. Programs written in different

aspect-oriented languages, may therefore, display different quality factors even though they implement the same behavior.

The goal of this thesis is to enhance the current composition mechanisms of aspect-oriented languages and if necessary introduce new ones so that the programs written using these new composition mechanisms will manifest better software qualities, in terms of evolvability, comprehensibility, predictability and adaptability. The problems of composition mechanisms of current aspect-oriented languages are termed as: *fragility of pointcuts*, *tightly coupled advice-pointcut bindings* and *aspect composition at shared join points*, which will be briefly summarized in the following sections.

Fragility of Pointcuts

The process of software development generally consists of refinement of conceptual knowledge towards an executable program. During this process, "ideas" or design artifacts are mapped onto implementation artifacts. Typically, the actual implementation contains the artifacts that are necessary for execution. Consequently, certain conceptual knowledge that expresses the intentions of a design may not be explicitly represented in the final program. This information-loss has an important consequence on the *pointcut* expressions of aspect-oriented languages. Although *design information* is not necessary for correct execution, it is generally required within pointcut expressions to avoid fragility of pointcuts with respect to changes in the implementation. The lack of explicit design information in the implementation forces programmers to refer to the design information in other ways, for example, based on lexical information and syntactic conventions. However, this restricts the adaptability and evolvability of pointcut expressions, i.e. pointcuts may not designate the intended join points anymore as the source evolves, since it involves too much information that is hard-coded, and often implementation-specific.

6. In Chapter 2, we describe a reference model of aspect-oriented languages and discuss in detail the aspect-oriented abstractions that we mentioned in the previous section.

Tightly Coupled Advice-Pointcut Bindings

In most aspect-oriented languages, the language constructs for expressing aspects, advices and pointcut specifications are tightly coupled with each other. By tight coupling we mean that these constructs either cannot be syntactically separated from each other or they maintain explicit references to each other. These issues may restrict the evolvability of the weaving specifications, result in code that requires intensive maintenance during the development of a system.

Aspect Composition at Shared Join Points

Aspect-oriented languages provide means to superimpose [6] crosscutting code (*advice*) on a given set of join points. It is possible that not just a single, but several advices need to be superimposed on the same join point. Such "shared" join points may give rise to issues such as determining the exact execution order and other dependencies among the aspects. A typical dependency, for instance, is the conditional execution of the advices of different aspects. In general, AOP languages lack abstractions to explicitly express such a dependency; programmers need to use workarounds and extra maintenance code to express the conditional execution among advices. As a result, aspects are tangled with extra behavior besides their intended behavior. Moreover, inappropriate realizations of conditional execution may introduce unwanted couplings between aspects. This renders difficulties in the reuse of aspects. To overcome these problems, explicit language constructs are necessary for the specification of the conditional executions of advices at shared join points.

1.7 Outline of the Thesis and the Approach

Figure 1.2 illustrates the structure of the thesis and the relations among the chapters.

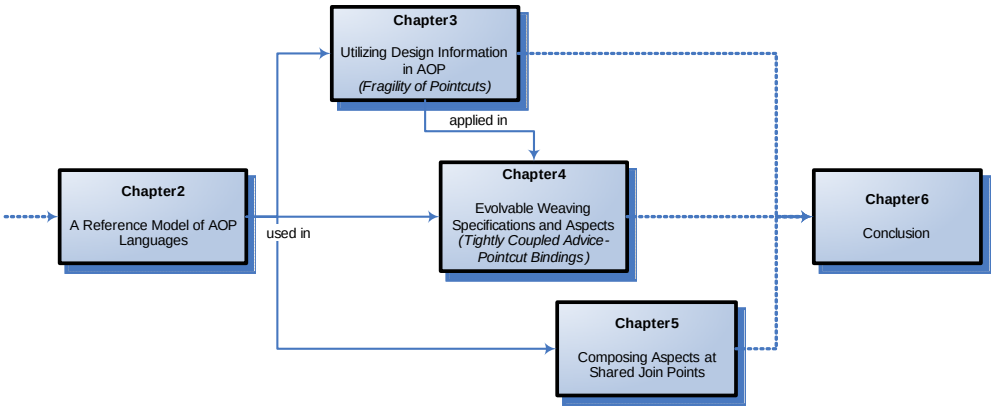

Figure 1.2 *Thesis map*

This thesis consists of the following chapters:

Chapter 2 introduces a reference model, which provides an overview of the state-of-the-art aspect-oriented languages. Furthermore, it exposes the common and distinctive language constructs of these languages within the perspective of different quality dimensions, such as expressiveness and composability. These language constructs are further elaborated in chapter 3, 4 and 5.

Chapter 3 presents an in-depth analysis of the role of design information within the context of aspect-oriented programming. In this chapter, to cope with the *fragility of pointcuts* problem, we propose new language contracts to incorporate the relevant design information in the specification of pointcut expressions. This is achieved by introducing an explicit language construct that captures the necessary design information and is used to designate the join points. This creates more evolvable programs compared to the purely lexical join point designators.

Design information can be associated with program elements at least in 3 ways:

- Manually using annotations
- Automatic derivation based on the existence of other design information
- Through introduction of (possibly crosscutting) annotations:

The main advantage of our approach is the increased evolvability of programs through the reduced syntactic dependency between aspect modules and the structure of the programs. This chapter is based on work published in [8, 12].

Chapter 4 addresses the *tightly coupled advice-pointcut bindings* problem. For this purpose, first the key properties of the current *advice-pointcut binding* mechanisms in the state-of-the-art aspect-oriented programming languages are studied. Based on this study, a new advice-pointcut mechanism is introduced, which provides *associative access* to advices/aspects. By this way, advices/aspects can be designated using their properties or relationships to other units, instead of explicit names. This makes advice-pointcut bindings more evolvable, since advices and aspects are bound through their properties but not through syntactic names. This chapter is based on work published in [11], [12] and [5].

Chapter 5 addresses the *aspect composition at shared join points* problem. First, an extensive analysis is presented on the issues that arise when multiple advices have to be executed at the same join point. Based on this analysis, a set of requirements are identified for the safe and flexible composition of aspects at shared join points. As a solution, a constraint-based language is proposed, which allows partial specification of the composition constraints per aspect. This language can express orderings and various types of conditional execution of advices, at shared join points. In addition, conditions on the presence and/or absence of a given aspect can be defined. The system automatically checks all the independently specified constraints per shared join point and verifies and enforces them accordingly. It is also illustrated how this language can be adopted by the aspect-oriented languages AspectJ and Compose*. This chapter is based on work published in [9, 10].

Chapter 6 evaluates the contributions of the thesis, describes the directions for future work and gives conclusions.

1.8 References

[1] AKSIT, M., AND BERGMANS, L. Obstacles in object-oriented software development. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 1992), ACM Press, pp. 341–358.

[2] D'ABREO, L. Java dynamic proxies: One step from aspect-oriented programming, DevX.com Technical Articles, July 2004.

[3] ELRAD, T., AKSIT, M., KICZALES, G., LIEBERHERR, K., AND OSSHER, H. Discussing aspects of AOP. *Comm. ACM 44*, 10 (Oct. 2001), 33–38.

[4] ERNST, E. Simple, eh? In *SPLAT: Software engineering Properties of Languages for Aspect* (Mar. 2004), L. Bergmans, K. Gybels, P. Tarr, and E. Ernst, Eds.

[5] HAVINGA, W., NAGY, I., BERGMANS, L., AND AKSIT, M. Detecting and resolving ambiguities caused by inter-dependent introductions. In *Proc. 5th Int' Conf. on Aspect-Oriented Software Development (AOSD-2006)* (Mar. 2006), A. Rashid and H. Masuhara, Eds., ACM Press.

[6] KATZ, S., AND GIL, Y. Aspects and superimpositions. In *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)* (June 1999), C. V. Lopes, A. Black, L. Kendall, and L. Bergmans, Eds.

[7] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *Proc. ECOOP 2001, LNCS 2072* (Berlin, June 2001), J. L. Knudsen, Ed., Springer-Verlag, pp. 327–353.

[8] NAGY, I., AND BERGMANS, L. Towards semantic composition in aspect-oriented programming. In *European Interactive Workshop on Aspects in Software (EIWAS)* (Sept. 2004), K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka, Eds.

[9] NAGY, I., BERGMANS, L., AND AKSIT, M. Declarative aspect composition. In *2nd Software-Engineering Properties of Languages and Aspect Technologies Workshop* (Mar. 2004), L. Bergmans, K. Gybels, P. Tarr, and E. Ernst, Eds.

[10] NAGY, I., BERGMANS, L., AND AKSIT, M. Composing aspects at shared join points. In *Proceedings of International Conference NetObjectDays, NODe2005* (Erfurt, Germany, Sep 2005), A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, Eds., vol. P-69 of *Lecture Notes in Informatics*, Gesellschaft für Informatik (GI).

[11] NAGY, I., BERGMANS, L., GULESIR, G., DURR, P., AND AKSIT, M. Generic, property based queries for evolvable weaving specifications. In *3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop* (Mar. 2005), L. Bergmans, K. Gybels, P. Tarr, and E. Ernst, Eds.

[12] NAGY, I., BERGMANS, L., HAVINGA, W., AND AKSIT, M. Utilizing design information in aspect-oriented programming. In *Proceedings of International Conference NetObjectDays, NODe2005* (Erfurt, Gergmany, Sep 2005), A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, Eds., vol. P-69 of *Lecture Notes in Informatics*, Gesellschaft für Informatik (GI).

[13] ROGER PRESSMAN, D. I. *Software Engineering: A Practitioner's Approach European Adaption*. McGraw-Hill International, 2000.

Chapter 2

A Reference Model of Aspect-Oriented Languages

2.1 Introduction

During the last decade, a considerable number of aspect-oriented programming (AOP) languages has been introduced. In this chapter, we propose a reference model which aims to represent the common and distinctive features of AOP languages. This reference model can serve at least for three purposes. First, it provides a basis for the readers to understand the important characteristics of the state-of-the-art AOP languages. Second, it helps us to compare the AOP languages with each other. Third, it exposes the issues that have to be considered when a new AOP language needs to be developed. Obviously, the reference model is derived from the current aspect-oriented languages[1] and has to be reconsidered when new languages become available.

The rest of this chapter is organized as follows: in section 2.2, we discuss shortly the fundamental language concepts of aspect-oriented languages. Section 2.3 explains the design dimensions that we use to explore the design space of the aspect-oriented language concepts. In sections 2.4 and 2.7, we analyse the main language concepts in various aspect-oriented languages, and build up a reference model of these aspect-oriented languages in the view of

1. It is important to mention that we could not take every aspect-oriented language into account when we created this reference model. Languages that were part of this study were primarily those languages that were studied in the language surveys of [12].

the introduced design dimension. In the view of our reference model, section 2.8 discusses two special aspect-oriented languages, DemeterJ and HyperJ. Section 2.9 explains the concepts of Composition Filters [8] and discusses the language constructs of Compose* that is the realization of the Composition Filters model. Finally, sections 2.10 and 2.11 provide discussion and background on related work.

2.2 The Fundamental Concepts of AOP Languages

To illustrate the fundamental concepts of AOP languages, we refer to Listing 2.1, which was introduced in Chapter 1.

```
0) public aspect Tracing{
1)    pointcut stateChanges():
2)       call(void Point.setX(int)) ||
3)       call(void Point.setY(int));
4)
5)    Object around(): stateChanges(){
6)       String m_name =
7)          thisJoinPoint.getSignature().getName();
8)
9)       System.out.println("Entering the method " + m_name);
10)      Object temp = proceed();
11)      System.out.println("Exiting the method " + m_name);
12)
13)      return temp;
14)   }
15)}
```

Listing 2.1 *An example of an aspect in AspectJ*

2.2.1 Join Point

The notion of join point is a fundamental concept in AOP. Join points can be classified as *structural* and *behavioral join points*. Behavioral join points correspond to events in the control flow of a program. For instance, in object-oriented languages, behavioral join points may refer to passing messages and writing on instance variables. In the code (i.e. physical representation) of the program, there are statements that correspond to these events. For instance, a call statement corresponds to the message passing event, an assignment statement corresponds to writing an instance variable, etc. A statement in the code

that corresponds to a behavioral join point in the execution is called *shadow point* in the aspect-oriented literature.

Structural join points are expressed in terms of the syntactic constructs of the programming language. For example, a structural join point may correspond to a particular class in a program. Structural join points will be discussed in detail in section 2.4.1.

As an example for behavioral join point, consider now line 7, in Listing 2.1. Here, the pseudo variable thisJoinPoint is an instance of org.aspectj.lang.JoinPoint and represents the join point context at the moment of execution. In most AOP languages, structural join points are implicit for the programmers, whereas behavioral join points are explicitly represented by keywords (or other language constructs).

Naturally, the set of possible join points (i.e. the types of join points) depends on the characteristics of the adopted programming language. This means, for example, that an object-oriented language and a procedural language expose different types of joinpoints. On the other hand, there may be join point types that share similar properties in these languages. We will continue the discussion on join points in section 2.4.

2.2.2 Pointcut

Pointcut designator is an expression that refers to zero or more join points. Pointcut designators are formulated, typically, using a declarative language for describing the join points where the behaviour of a crosscutting concern (i.e. advice) is executed. In this thesis, the term "pointcut designator expression" will be sometimes abbreviated as "pointcut expression" or simply as "pointcut". Also the term "superimposition" will be used. A pointcut expression is in fact a *quantification mechanism* over the events and/or syntax of a program.

As an example of a quantification over behavioral join points, consider the pointcut stateChanges in Listing 2.1 (lines 1-3). Here, stateChanges designates two join points that correspond to the calling events on the methods setX or setY of the instances of class Point.

A pointcut expression consists of either *primitive pointcuts* or *composite pointcuts*. A composite pointcut expression may be built up from primitive and/or composite pointcuts with the help of composition operators. As an example of a composition operator, consider the characters "||" in Listing 2.1 (line 2). This operator indicates that the pointcut stateChanges either matches a call event on setX or setY.

To cope with the variability in the selected events, a pointcut expression may be parameterized with the properties of the designated join points. For example, the pointcut call(public * Point.*(int)) designates every calling event on the public methods of the instances of class Point, in which every call has an integer parameter and returns an instance of unspecified type. Pointcuts will be discussed in detail in section 2.5.

2.2.3 Advice

An *advice* represents a program module which is to be executed at the designated join points. There are three types of advices *before*, *after* and *around*, which correspond to the program modules to be executed prior, after or instead of the designated events, respectively. An advice is bound to zero or more join points through the use of a pointcut expression. For example in Listing 2.1, the around advice specified in lines 5-14, is bound to the joint point (calling events on setX or setY), through the pointcut stateChanges.

In contrast to the methods of traditional object-oriented languages, advices are not called explicitly. Instead, the execution of an advice is automatically "triggered" when the control flow reaches the join point that is designated. In the literature [17], this property is termed as "advice is *oblivious* to the join point". Consequently, the program modules, in which the events in their control-flow are designated, are also oblivious to the corresponding advices. We discuss the advice abstraction in detail in section 2.6.

2.2.4 Aspects

An *aspect* represents a program module which abstracts advices and pointcut specifications. An aspect may also incorporate member variables, methods, etc. In contrast to object-oriented languages, mostly aspects are instantiated implicitly by the adopted implementation technique. The programmer, however, may have a choice among various instantiation alternatives. For

example, the programmer may indicate that there should be only a single instance of a given aspect (singleton aspect). Or, the programmer may want to create multiple instances of a given aspect. An aspect instance can be shared by the objects that are designated by the pointcut expression of the aspect. In case of a single aspect instantiation, all designated objects share a single aspect instance. In case of multiple instances, each designated object share only its own aspect instance. We will discuss various concepts of aspects in section 2.7.

2.3 Design Dimensions and Design Alternatives

In this section, we will introduce the dimensions and alternatives of the aspect-oriented language concepts, which will help us to explore the design space that the current languages span. In the following sections, for simplicity, we will use the terms dimensions and alternatives for design dimensions and design alternatives, respectively.

Dimension of Semantics

In this section, we will elaborate on the *expressiveness* of the various aspect-oriented language concepts as adopted by the current languages. Of course, not all the languages adopt the same language concept in the same way. Moreover, some language concepts can be specific to a certain language. For example, only AspectJ and AspectWerkz offer the pointcut handler() to refer to an exception handling event.

Dimension of Notation

This section discusses the commonly adopted *notations*. Naturally, there are similarities and differences in the notations used. For example, AspectWerkz and JBossAOP use the Java method notation for representing *advices*. AspectJ and JAsCo, for instance, use a dedicated notation for representing advices.

Dimension of Composition

Here, we explore the specific *composition* mechanisms offered by aspect-oriented languages. For example, languages differ from each other in the way how advices and pointcut expressions are composed together.Also, some languages, for instance, adopt a dedicated notation to order the advices that are superimposed on the same join point.

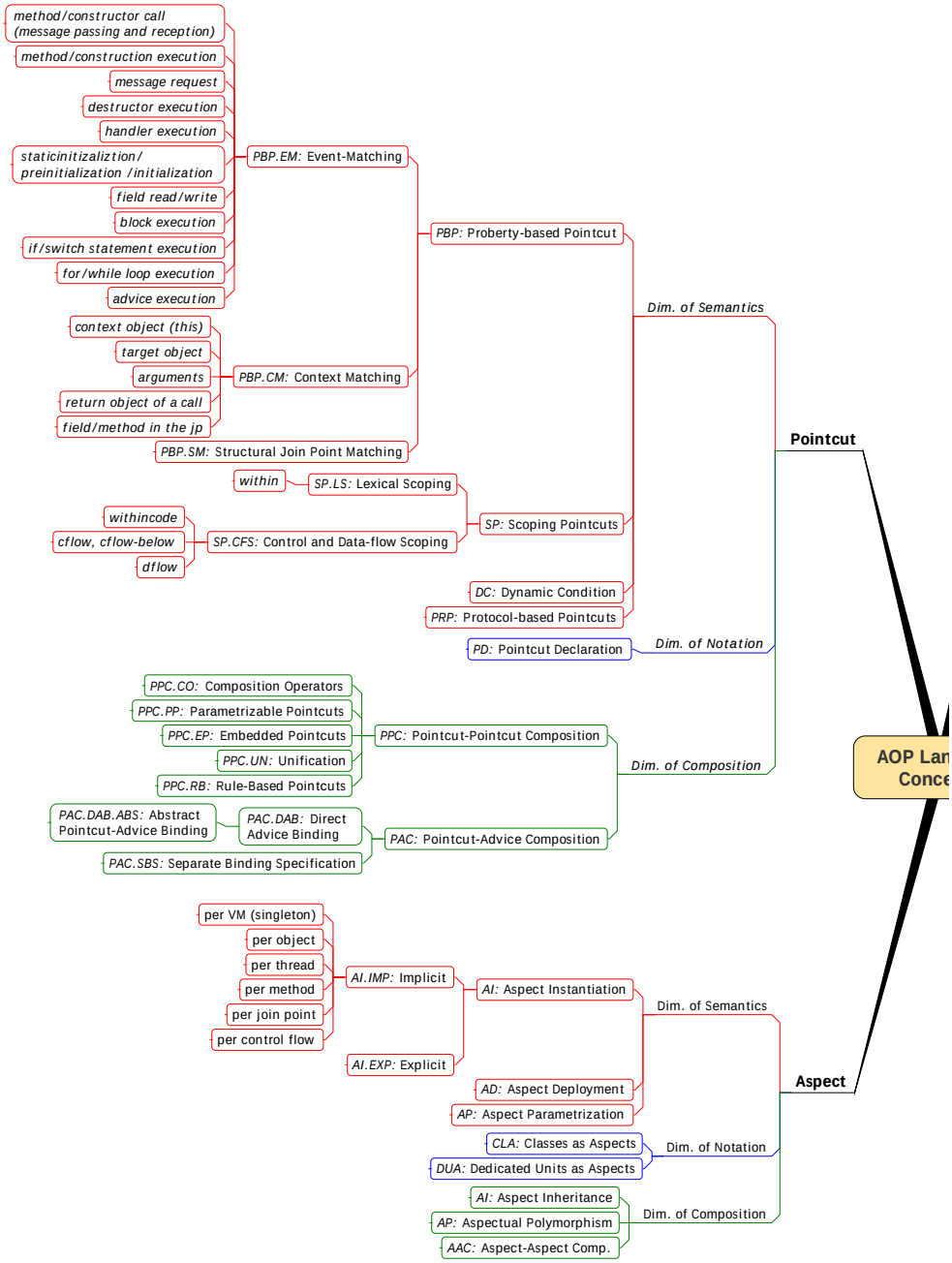

Figure 2.1 *The complete map of design dimensions - side (a)*

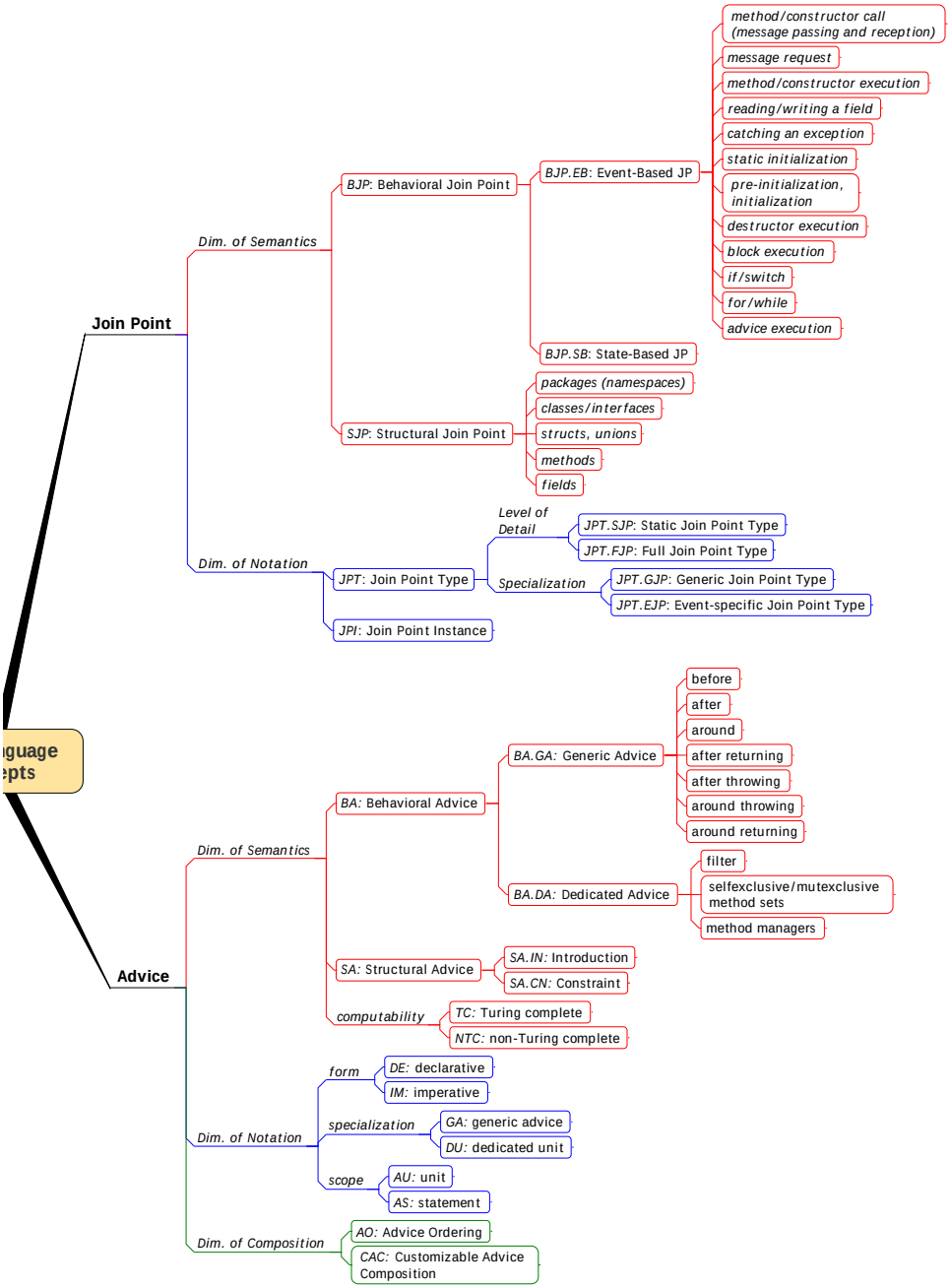

Figure 2.2 *The complete map of design dimensions - side (b)*

2.3.1 A Graphical Notation for Representing Dimensions and Alternatives

For each language concept, we use a graphical notation for depicting the relevant dimensions and alternatives. Consider for example, Figure 2.3. Here the root element on the left hand side of the picture represents the language concept to be analyzed, which is *advice* in this case.The branches directly connected to this root element depict the dimensions and design alternatives of the language concept. These are derived from the fundamental dimensions introduced in the previous section. The design alternatives for a given branch are indicated by rounded rectangles connected to the same branch. For example, in Figure 2.3, there are two design alternatives (*Behavioral Advice* and *Structural Advice*) on the branch that represents the dimension of semantics. A design alternative may have its alternatives as well. In Figure 2.3, for instance, *Generic Advice* is presented as an alternative of *Behavioral Advice*. The dimensions can be further specialized in sub-dimensions. In contrast to the alternatives, sub-dimensions are not necessarily exclusive. For example in Figure 2.3, the sub-branches *form* and *specialization* are sub-dimensions of the dimension of notation. We use abbreviations in the name of design alternatives (e.g. *BA: Behavioral Advice*) to indicate the correspondence of alternatives between the figures and the sections of this chapter.

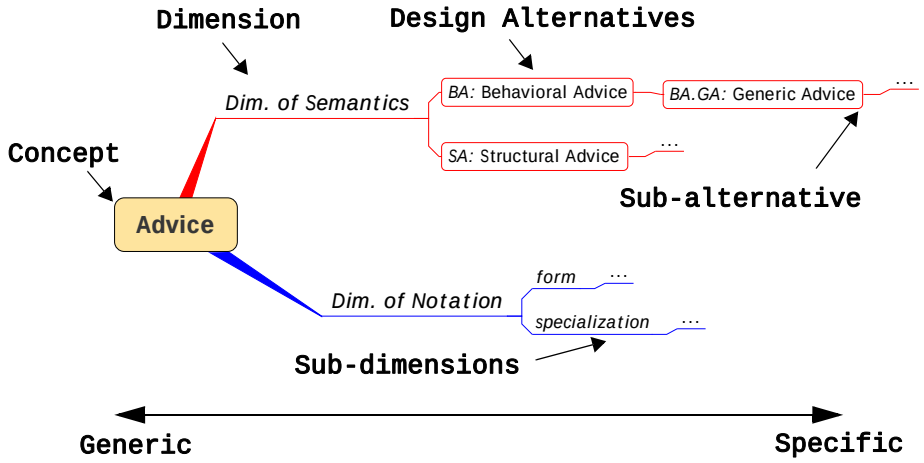

Figure 2.3 *An illustration of the graphical notation that is used in the analysis of the language concept advice.*

A detailed map of the design dimensions of each language concept can be found on pages 24 and 25. We discuss in detail these concepts in the next sections.

2.4 Join Points

Figure 2.4 illustrates the dimensions and alternatives of the language concept joint point. In the following subsections, we will discuss this figure in detail.

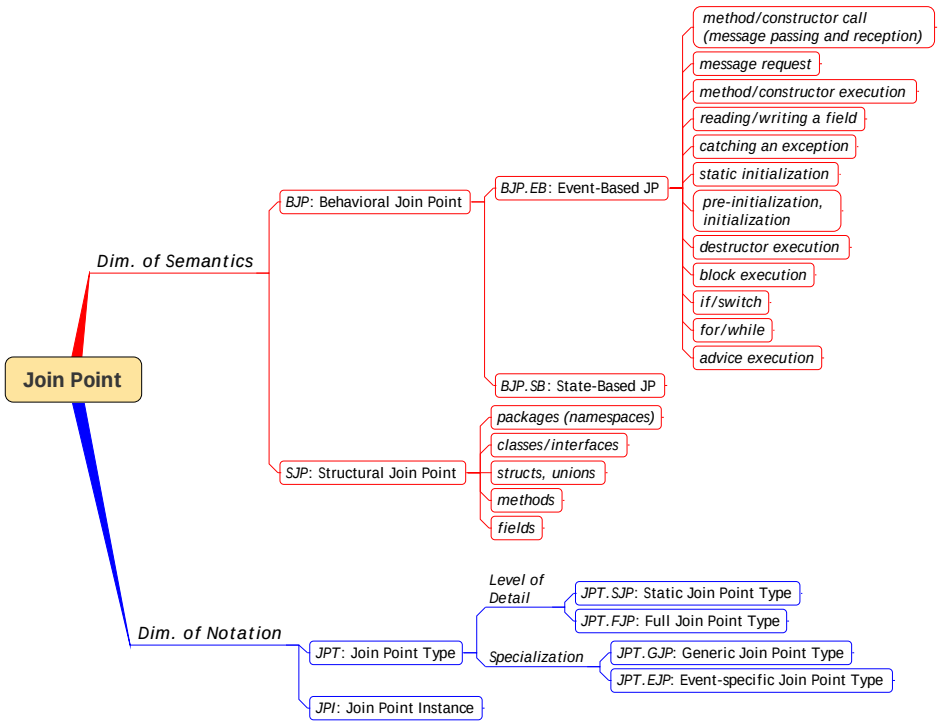

Figure 2.4 *The join point abstraction*

2.4.1 Dimension of Semantics

As we discussed in section 2.2.1, two main categories of join points can be distinguished from the perspective of semantics: behavioral and structural join points.

Behavioral Join Points (BJP)
In general, behavioral join points are particular events in the execution of an application. We distinguish two main categories of behavioral join points: event-based join points and state-based join points.

Event-based join point (BJP.EB)
Event based join points depend on the characteristics of the base language. For example, the following table lists the commonly used event-based join points in case the base language is object-oriented.

Table 2.2 *Event-based join points*

Join Point	*Representative Languages*
method/constructor execution	(default)
method/constructor call (aka. message passing or reception)	AspectJ [23], JAsCo[33], CARMA[12] , AspectC++ [32], AspectWerkz [10], JBossAOP[15]
exception handler execution	AspectWerkz, AspectJ, JBossAOP, JAsCo
field read/write	AspectJ, AspectWerkz, JBossAOP, CARMA
message receive	Compose*
destructor execution	AspectC++
(static/pre) initialization	AspectJ, AspectWerkz
block execution	AspectS
if/switch statement execution	Eos-T[31]
for/while loop execution	Eos-T
evaluation of arithmethic expressions	Fradet et. al. [18]
advice execution	AspectJ

The first four types of join points in this table are supported by most aspect-oriented languages, which adopt an object-oriented base language. There are also join points that are more language-specific: the initalizations in Java, the

destructor execution in C++, and the code block execution in Smalltalk belong to this set.

The join points *message passing* and *message reception* are needed to be distinguished only in the languages that cannot distinguish between the caller and the callee objects of a message passing. In other words, in those languages, there is only one instance variable that refers to either the caller or callee, depending on the type of the join point. When the language is capable of referring to both the caller and calle objects, a single *method call* join point can express both *message passing* and *message reception*. For instance, AspectJ can refer to the caller and callee objects by the pointcuts this() and target(), respectively. The combination of the pointcuts this() && call() can express *message passing*, whereas the combination of call() && target() can express *message reception*[2]. In Compose*, the so-called composition filters are used to define crosscutting behaviour. The filters are classified as input and output filters. The input filters affect the incoming calls, whereas the output filters affect the outgoing calls. Within the context of the input and output filters, the pseudo variable inner refers to the callee and caller objects, respectively.

We consider the *message reception* and the *message request* (e.g. in Compose*) as different kinds of join points: in case of message request, the message is not dispatched to the corresponding target object and method yet, whereas in case of method reception these properties are already determined.

The join points offered by aspect-oriented languages are of course not limited by the examples presented here. For example, Eos-T, which is an aspect-oriented extension of C#, provides join points for control statements and loops. In [18], Fradet et. al. proposed a pointcut to designate the evaulation of arithmetic expressions as join points. AspectJ also provides another interesting type of join point for the execution of advices.

State-based Joints (BJP.SB)
In general, a state-based join point is defined by a state transition in the execution of a program. Languages that support state-based join points defines the

2. The earlier versions of AspectJ had both the pointcuts call() and reception(). In version 1.0alpha1, these pointcuts were merged and the pointcuts this() and target() were introduced to make the pointcut language simpler without giving up its expression power.

aspectual states [5] that crosscut other concerns as explicit program elements. This provides a better modularization technique for crosscutting concerns that require state monitoring. Defining expressive state-based join points is an active research area [5, 11, 36].

Structural Join Points (SJP)
Structural join points are based on the syntax of the base language. Depending on the syntax of the adopted base language, packages/namespaces, classes, interfaces, methods, fields, structs and unions (in C++) are typically used as structural join points.

Structural join points are particularly useful in modifying the structure of programs. For example, in AspectJ the keyword declare parents can be used to replace the superclass of a class with another, or it can be used to extend the interface of a class. We will discuss these possibilities in more detail in section 2.6.1.

2.4.2 Dimension of Notation

Join Point Types (JPT) and Instances (JPI)
AOP languages adopt two alternative ways to denote the current behavioural join point in execution: through (i) pre-defined types, or (ii) queries. Some practical examples of pre-defined types and the corresponding languages are given in the following list:

- **a.** org.aspectj.lang.JoinPoint -- AspectJ
- **b.** org.codehaus.aspectwerkz.joinpoint.JoinPoint -- AspectWerkz
- **c.** org.jboss.aop.Invocation and its subclasses --JBoss
- **d.** jasco.runtime.MethodJoinpoint --Jasco
- **e.** Composestar.Runtime.FLIRT.message.ReifiedMessage --C*[3]

The instances of these types represent "the current join point in execution". These instances generally serve two purposes: to obtain *reflective information* about the join point and to provide some *control* over the execution. As an instance name, typically the pseudo variable thisJoinPoint is adopted; we have

3. Compose*

already shown the use of this pseudo variable in AspectJ in Listing 2.1. Instead of using a pseudo variable, in some languages, information about the current join point is passed as an argument to the corresponding advice. Consider, for example, the AspectWerkz code shown in Listing 2.3. Here, the method log (line 4-13) represents the advice that prints a message before and after the corresponding method is executed. In line 4, information about the current join point is passed as an argument. To transfer the thread of execution from the aspect code to the base code, in line 9, the operation proceed is invoked on the variable joinPoint. In AspectJ, the operation proceed is provided as a language keyword, instead. (The pointcuts are specified in another place, we will discuss this in detail in the following section)

```
0)  import org.codehaus.aspectwerkz.joinpoint.JoinPoint;
1)
2)  public class TracingAspect {
3)
4)     public void log(JoinPoint joinPoint) {
5)        String m_name =
6)           joinPoint.getSignature().getName();
7)
8)        System.out.println("Entering the method " + m_name);
9)        Object temp = joinPoint.proceed();
10)       System.out.println("Exiting the method " + m_name);
11)
12)       return temp;
13)    }
14)}
```

Listing 2.3 *An example of using a join point instance in AspectWerkz*

Join point types can also be classified as *generic (JPT.GJP)* and *dedicated join point* types. AspectJ and AspectWerkz, for example, adopt *generic join point* types such that all sorts of join points are represented by a single join point type. In addition, for performance reasons, *static join points (JPT.SJP)* are introduced to retrieve only static information about the current joint in execution. Examples of languages which adopt *dedicated join point (JPT.EJP)* types are JBoss, JasCo and Compose*. Each different sort of join point defines its own dedicated join point type. For instance, in JBoss, each dedicated join point type is a subclass is the generic join point type. CARMA is an example of languages that adopt a query mechanism instead of join point types. Typically, these languages use *context matching pointcut designators* to retrieve informa-

tion about the current join point in execution[4]. We will discuss elaborate on these pointcut designators in section 2.5.1.

2.5 Pointcuts

Figure 2.5 illustrates the dimensions and alternatives of the language concept pointcut.We iterate over these dimensions and discuss them in detail in the subsequent sections.

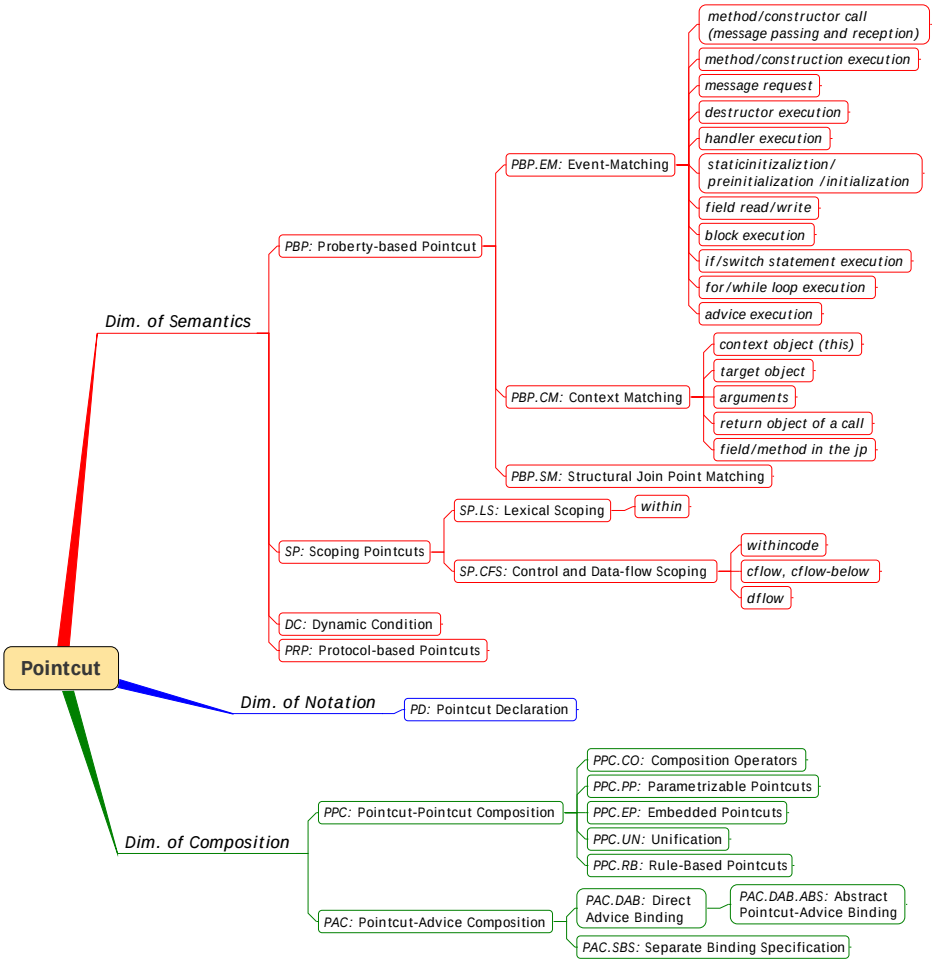

Figure 2.5 *The pointcut concept*

4. However, languages that support join point types, e.g. AspectJ, typically provide context matching pointcuts as well.

2.5.1 Dimension of Semantics

There are two main groups of pointcut designators: *property-based pointcuts* and *scoping pointcuts*. Property-based pointcut describes a set of join points by referring to the properties of join points, such as return type, name and parameters of a call, for instance. Scoping pointcuts accept structural or behavioral join points as a parameter and designate the join points that relate to the parameter according to its definition. This can be static reference such as the corresponding lexical unit, or a dynamic reference such as the related control-flow.

Property-based Pointcuts (PBP)

Property-based pointcuts can be classified as event matching and context matching poincuts[5].

Event Matching (PBP.EM) In current AOP languages, event matching pointcuts refer to the events which correspond to the adopted join points. The following table contains the pointcut designators that correspond to the join points that we have listed in section 2.4.1[6]:

Table 2.4 *Event matching pointcuts*

	Example	
Pointcut	*Language*	*Designator*
method/constructor call	AspectJ	call()
message request	Compose*	{filterelement} in inputfilter
method/constructor execution	AspectJ	execution()
destructor execution	AspectC++	destructor()
exception handler execution	AspectWerkz	handler()
(static/pre) initialization	AspectJ	(static/pre)initialization()
field read/write	AspectJ	get()/set()
block execution	CARMA	blockExecution()
if/switch statement execution	Eos-T	conditional()

5. Colyer categorized the pointcut designators of AspectJ with a subset of these categories in [13]

Table 2.4 *Event matching pointcuts*

for/while loop execution	Eos-T	iteration()
evaluation of arithmethic expressions	Fradet et. al. [18]	DOMAIN () IN()
advice execution	AspectJ	adviceexecution()

Context matching (PBP.CM) Context matching pointcut designators can have two roles: (1) provide information about the context of the join point for a more fine-grained designation; (2) expose the context of a join point in terms of first class entities that are passed to the advices to which the pointcut is bound. Listing 2.5 and Listing 2.6 present two examples to illustrate these roles.

In these two AspectJ examples, the pointcut designators denote exactly the

```
0) /* Case 1: fine-grained designation */
1) pointcut example1():
2)    call (public void set*(..))
3)          && this(Employee);
4)
5) after(Employee ee): example1(){
6)   String name =
7)        ((Employee)thisJoinPoint.getThis()).getName();
8) }
```

Listing 2.5 *Using context matching pointcut designators*

same joinpoints. The pointcut designator in line 2 of Listing 2.5, specifies the pointcuts with the following features: a method call which starts with the prefix 'set', has a public visibility, a void return type, and have an arbitrary number of parameters of arbitrary types. The pointcut designator in line 2 is composed with the pointcut designator of line 3 using the "&&" (and) operator. In line 3, the context matching pointcut designator this refers to the instances of class Employee. The method calls as described in line 2, therefore, have to be made on the instances of class Employee.

6. Note that the pointcut designators in the column Example are listed for representative purposes, the given pointcut may also be supported by more languages. E.g. the designation of method execution is supported by AspectWerkz, JBossAOP, JAsCo, AspectC++, CARMA and other AOP languages that were not included in this study.

There is a slight difference between the line 3 of Listing 2.5 and line 3 of Listing 2.6. In Listing 2.5, the expression "this(Employee)" is used, whereas in Listing 2.6, the pointcut is specified with a parameter (Employee e), and this parameter is bound in the designator this() - i.e. it is written as this(e). To illustrate the differences, we will now refer to Listing 2.5 and Listing 2.6.

```
0) /* Case 2: passing context information */
1) pointcut example2(Employee e):
2)    call (public void set*(..))
3)          && this(e);
4)
5) after(Employee ee): example2(ee){
6)    String name = ee.getName();
7) }
```

Listing 2.6 *Passing context information as parameters*

Line 3 of Listing 2.6 declares an after-advice with the parameter ee of class Employee. This after-advice is coupled with the pointcut example2 and the parameter ee of the advice is passed as an argument to the pointcut. The effect of this program is as follows: when the joinpoint is selected, the call must have been originated from an instance of class Employee, whose identity is denoted by the context matching pointcut this(e) and is passed as a parameter to the advice. In line 3 of Listing 2.6, the identity is used to retrieve the name of the corresponding employee.

Now we will compare the Listings of 2.5 and 2.6 with each other. Both programs print the name of the calling objects. In Listing 2.5, the identity of the calling object is obtained through the pseudo variable thisJoinPoint, whereas in Listing 2.6, the identity is obtained from the context of the joinpoint. The code in Listing 2.6 is considered as a more type-safe solution for the following reason. In Listing 2.5, line 7, the operation getThis() returns an instance of class Object. Therefore, the returned object must be explicitly linked to class Employee using type-casting in implementation. In the context exposure pointcut implementation of Listing 2.6, this relation is explicitly specified without a need to have type-casting. Table 2.7 contains the most well-known context matching pointcut designators from various AOP languages

According to the instances of their arguments, the pointcut designators this() and target() in AspectJ denote to the join points in the currently executing objector in the callee object, respectively. For instance, call(public void set*(..)) && target(Employee) denote to the calls starting with the prefix 'set' on the instances of class Employee, independent of which object makes the call. The primitive pointcut args() can be used to refine or expose the argument of a call. For instance, consider the following pointcut specification: pointcut exposeIntArg(int i): call(public void set*(..)) && args(i); This pointcut denotes to the calls starting with the prefix 'set' and have an int parameter. The variable i can be accessed from the advice which is associated with this pointcut specification.

Table 2.7 *Context matching pointcuts*

	Example	
Pointcut	*Language*	*Designator*
context object of the join point (this)	AspectJ	this()
	CARMA	inObject(?jp, ?object)
target object of the join point	AspectJ	target()
arguments	AspectJ	args()
return object of a call/execution	AspectC++	result()
	CARMA	objectResponse(?object, ?message, ?response)
field/method in the context of the join point	AspectWerkz	hasmethod()/hasfield()

```
0)  pointcut exposeIntArgument(int i):
1)     call (public void set*(..)) && args(i);
2)
3)  Object around(int i): exposeIntArgument(i){
4)     return proceed(++i);
5)  }
```

Listing 2.8 *Increasing an int argument of a call*

Listing 2.8 shows an example advice that makes use of an exposed argument. Here, every call which starts with the prefix 'set' and which has an integer argument is intercepted before it is executed, its integer argument is retrieved and incremented by one, and the new value is substituted in the call and the call is then executed with the new integer value.

In AspectC++, the primitive pointcut result() denotes the join points, whose execution results in an instance of the class used as the parameter of result().

CARMA has an extensive set of context matching pointcuts, where three of them are given in Listing 2.9. These provide information on some dynamic properties about the denoted join point (These predicates are called *dynamic join point property predicates* in CARMA). The predicate inObject provides the context of the object which is related to the join point. The predicate objectVariable provides access to the variable of the object which is denoted by the parameters ?varName and ?value. The predicate objectResponse selects the join point based on the result of the corresponding call. Listing 2.9 shows a pointcut specification in CARMA, which designates every method execution with two arguments, where the value of the second argument is five and the object which executes the call has an instance variable named 'myVariable'. In fact, there are three pointcuts here connected using the logical AND connector (the character "," as in Prolog).

```
0) ?jp matching
1)   reception(?jp,?anyMethod,<?firstArgument,5>),
2)   inObject(?jp,?object),
3)   objectVariable(?object,myVariable,?value)
```

Listing 2.9 *An example CARMA pointcut*

Structural join point matching (PBP.SM) As we discussed in section 2.2.1, structural join points are defined in terms of syntactical constructs of the language. Several AOP languages offer means to designate structural join points. For example, AspectJ adopts so called *type patterns* for designating the name of a syntactic construct in a program. For instance, the type pattern com.acme.model..* designates every class within the package com.acme. The

following listing shows an application of a type pattern within the construct declare @type:

```
0) declare @type:
1)    org.xyz.model..* : @BusinessDomain
```

Listing 2.10 *A type pattern designating structural join points in AspectJ*

This declaration states that the annotation @BusinessDomain is attached to every type defined within the package org.acme.model.

AspectC++, for example, introduces so called *name pointcuts* (also *match expressions*) to designate structural join points. For instance, int C::%(...) matches any member function of class C that returns an int value. AspectC++ has two extra predicates for designating types as structural join points: base() and derived(). The first predicate returns the superclasses of classes used as the parameters of the predicate. The second one returns all the subclasses of classes used as the parameters of the predicate. In the example shown in Listing 2.11, the pointcut scalable designates class Rectangle and all classes derived from class Point which are at the same time direct or indirect base classes of class Rectangle. (Example is taken from [35].)

```
0) class Shape { ... };
1) class Point : public Shape { ... };
2) ...
3) class Rectangle : public Line, public Rotatable { ... };
4)
5) /* --- */
6)
7) pointcut scalable() =
8)  (base("Rectangle") && derived("Point")) || "Rectangle";
```

Listing 2.11 *Using the predicates base() and derived() in AspectC++*

Compose* has an extensive set of predicates to designate program elements as structural join points. These predicates can refer to the properties of program elements and their relationship with other program elements as wellListing 2.12 shows an example of their usage. In lines 0 and 3, the pointcut collectionClass designates an instance of a class myapp.Collection. In the second

part of this expression, in line 2, the predicate (classInheritsOrSelf) selects class myapp.Collection and all its subclasses as a set and binds it to the variable C..

```
0) collectionClass = { C |
1)   isClassWithName(Collection, 'myapp.Collection'),
2)   classInheritsOrSelf(Collection, C)
3) }
4)
5) guiClasses = { C |
6)   isNamespaceWithName(NS, 'myapp.gui'),
7)   namespaceHasClass(NS, C)
8) }
9)
10) returnsStrings = { C |
11)   isClassWithName(Str, 'java.lang.String'),
12)   methodReturnClass(M, Str),
13)   classHasMethod(C, M)
14) }
15)
```

Listing 2.12 *Three example pointcuts in Compose**

The second pointcut guiClasses between lines 5 and 8 designates all classes within the namespace myapp.gui.

The third pointcut returnsStrings between lines 10 and 14 designates all classes that have at least one method that returns a string. In the first pointcut (called *selector* in the terminology of Compose*), named collectionClass, the first predicate selects the class myapp.Collection in the application and if this class exists, it will be bound to the variable Collection. The second predicate (classInheritsOrSelf) maintains the same value for the variable Collection, through the unification mechanism of Prolog[7]. This second predicate selects the class myapp.Collection and all subclasses that are inherited from it and binds them as a result set to variable C. This result set is used by the selector through the variable C as denoted in line 0. The second selector (guiClasses) works in a similar manner: it selects all classes within the namespace myapp.gui. The third selector is a bit more complicated: it selects all classes that have at least one method that returns a String value. We refer to Appendix B for a more complete description of the available Compose* predicates.

7. We will discuss this in detail in section 2.5.2.

Dynamic conditions (DC)
In some AOP languages pointcut designators allow using dynamic conditions, which may, for instance, refer to an argument of a method call.

As shown in Listing 2.13, for instance, in AspectJ, the primitive pointcut if takes a Boolean expression as an argument and designates any call starting with the prefix 'set' that has an int parameter of value 5.

```
0) pointcut exposeIntArgument(int i):
1)    call (public void set*(..)) && arg(i) && if(i==5);
```

Listing 2.13 *Using dynamic condition in a pointcut specification of AspectJ*

In AspectJ, the pointcut if() can only make references to either the context variables of its pointcut specification, or static fields and methods because it is executed outside the context of the corresponding an aspect instance.

Compose* incorporates dynamic conditions in the filter specifications. A typical example is given in Listing 2.14. Here, the conditions are declared between line 3 and 4. The condition enoughCredits is used then in line 6 as an expression of the Error filter.

This filter expression is delimited using braces and contains two elements separated by a comma. A filter element is analogous to a call pointcut of AspectJ. The first filter element (in line 6) ensures that if the condition enoughCredits is *true* then the filter will accept any call (i.e. *.*) of any type.

```
0) concern CreditConcern {
1)   filtermodule TakeCredits {
2)     ...
3)     conditions
4)       enoughCredits : credits.enoughCredits();
5)     inputfilters
6)       check : Error = { enoughCredits => [*.*],
7)                         True ~> [*.play] };
8)     ...
9) }
```

Listing 2.14 *Using condition in Compose**

JAsCo introduces a language construct called hook, which encapsulates both the pointcut and advice specifications as first class entities. Also the method isApplicable() is provided for specifying the corresponding runtime condition. The advices of a hook are allowed to execute only when isApplicable() returns *true*. Listing 2.15 shows an example of using the method isApplicable() (taken from [19]).

```
0) class ConditionalPublishManager extends PublishManager {
1)
2)   boolean subscribed = false;
3)
4)   hook Subscribe {
5)     Subscribe(subscribe(..args)) {
6)       execute(subscribe);
7)     }
8)
9)     isApplicable() { return !subscribed; }
10)
11)    after() {
12)      subscribed = true;
13)    }
14)  }
15)  ...
16)}
```

Listing 2.15 *Using isApplicable() in JAsCo*

In this example, isApplicable() specifies that all advice of the hook Subscribe will only be executed if the variable subscribed is not false.

Scoping Pointcuts (SP)
Scoping pointcuts designate join points based on their relationship to their context, for instance, the lexical unit or the control-flow in which they occur (in contrast to property-based pointcuts that designate a join point based on their direct properties or, the properties of their shadowpoints). Note that both scoping and context-matching pointcuts aim at refining the context of the join points to be designated. However, there are two important differences between context-matching and scoping pointcuts: (1) context-matching pointcuts can refer to only direct properties of the join points, while scoping pointcut may refer to properties of a join point in a larger context through various types of relationships, such as the chain of calls in a control-flow; (2) unlike context-

matching pointcuts, scoping pointcuts cannot expose the context of a join point in terms of first class entities that are passed to the advices to which the pointcut is bound.

Lexical Scoping (SP.LS)
Lexical scoping pointcuts expect pointcuts describing structural join points as their arguments. A lexical scoping pointcut matches a join point based on the information whether the shadow point of the join point is defined within the scope of a structural join point (i.e. a program element).

The lexical pointcut within() can be found in many AOP languages, such as AspectJ, AspectWerkz, JBoss, (AspectC++[8]). It expects packages and types as scopes in its parameter.

Control and data flow-based scoping (SP.CFS)
Control flow-based scoping pointcuts expect pointcuts describing behavioral join points as arguments. They designate join points that occur within the control flow of the given behavioral join point.

Table 2.16 *Scoping pointcuts*

	Example	
Pointcut	*Language*	*Designator*
execution within a method	AspectC++	withincode()
execution within control-flow	AspectWerkz	cflow(), cflowbelow()
execution within data-flow	AspectJ	dflow[x,x]()

The pointcut withincode() expects methods and constructors as scopes in its parameter; it will match any join point where the corresponding join point occurs within the given set of methods or constructors.

The cflow pointcut picks out all join points that occur between entry and exit of each join point P picked out by Pointcut, including P itself. Hence, it picks out the join points in the control flow of the join points picked out by Pointcut. [38]

8. In Aspect++, the pointcut within() can also accept methods as parameters.

The cflowbelow pointcut picks out all join points that occur between entry and exit of each join point P picked out by Pointcut, but not including P itself. Hence, it picks out the join points below the control flow of the join points picked out by Pointcut. These control-flow based pointcuts are supported by various AOP languages, such as AspectJ, AspectWerkz, AspectC++, and JAsCo.

The dflow pointcut is another type of scoping pointcut, its definition is the following [22]: assume x is bound to a value in the current join point, dflow[x,x](Pointcut) matches the join point if there exists a past join point that matches Pointcut,and the value of x originates from a value bound to x in the past join point. More details on dflow can be found in [22].

Protocol-based Pointcuts (PRP)
To the best of our knowledge, protocol-based pointcuts are currently supported only by JAsCo. A protocol-based pointcut allows to specify a dynamic sequence of joinpoints instead of a single joinpoint, e.g. the execution of methods A-B-C in that order. Every transition in the protocol is defined using the JAsCo pointcut language. Advices can be attached at every transition in the protocol [36].

2.5.2 Dimension of Notation

Pointcut Declaration (PD)
Typically, AOP languages use a keyword, e.g. pointcut, that declares a pointcut. The keyword is followed by the identifier of the pointcut. The pointcut declaration may have formal parameters that are bound to the arguments of context matching pointcuts. This is followed by a concrete pointcut specification, see line 1 in Listing 2.17, as an example. Some aspect-oriented frameworks, such

as AspectWerkz and JBoss AOP, work with similar concepts, however, the

```
0) /* a pointcut definition as a program element */
1) pointcut example2(Employee e):
2)    call (public void set*(..)) && this(e);
3)
4) /* a pointcut definition as an XML specification */
5) <pointcut name="example2(Employee e)" expression=
6)    "call (public void set*(..)) AND this(e)">
7)
8) /* a pointcut definition as an annotation specification */
9) @Expression("call (public void set*(..)) && this(e)")
10)void example2(Employee e){}
```

Listing 2.17 *Pointcut definitions in Aspect & AspectWerkz*

pointcut is defined within an XML tag. Line 5 of Listing 2.17 shows an example pointcut declaration in XML that corresponds to the first pointcut of line 1. Besides the XML notation, a pointcut can also be defined by a predefined annotation (Java 1.5) in AspectWerkz; line 9 of Listing 2.17 shows an example that corresponds the previous pointcut declarations.

Note that the common features of all alternatives is that the pointcut definition has an identifier and may have formal parameters. These features will play an important role in the composition of pointcuts: the identifier allows for referencing (reusing) the pointcut definition in another pointcut definition; the parameters allow for exchanging parameters with other pointcuts or the advice which the pointcut is bound to.

2.5.3 Dimension of Composition

Pointcut-Pointcut Composition (PPC)
Pointcuts can be composed with other pointcuts. This allows for *reusing/refining* an existing pointcut by combining it with another (new) pointcut. In the pointcut-pointcut composition dimension, we will present four alternatives that support the composition of pointcuts: *composition operators*, *embedded pointcuts*, *parametrizable pointcuts*, *unification* and *rule-based compound pointcuts*.

Composition Operators (PPC.CO)
Most aspect-oriented languages use standard logic AND, OR and NOT operators to express the composition of predicate-based pointcuts.

```
0)  pointcut example2(Employee e):
1)     call(public void set*(..)) && this(e);
2)
3)  pointcut example3(Employee ee):
4)     example2(ee) || call(public void Manager.get*(..));
```

Listing 2.18 *Using logic operators to compose pointcuts in AspectJ*

Listing 2.18 shows two examples of using operators to compose pointcuts in AspectJ. The pointcut example2 has two pointcuts in its declaration: the pointcut call will designate every call with the given signature. This pointcut is composed with the pointcut this by an AND operator to specify the context of the calls; hence, it narrows down the possible set of join points. The pointcut example3 reuses the previous pointcut and composes it with another pointcut by an OR operator.

Embedded Pointcuts (PPC.EP)
Several AOP languages use *type* and other sorts of *patterns* - that designate structural join points - within the context of pointcuts that designate behavioral join points. In other words, a *structural join point matching pointcut* is embedded into a behavioral pointcut. In this context, the structural join point matching pointcut refers to the properties of the *shadow point* of a behavioral join point. For instance, in AspectJ, type patterns are not the only means for designating structural join points. There are other patterns, such as method and constructor patterns, that can be used in the arguments of event matching pointcuts to refer to the properties of the shadow point of a join point. For instance, the visibility, name, and return type of a method are such properties; all of these properties are part of the method type pattern of AspectJ. The following code fragment shows a simple example of a method pattern:

```
0)  pointcut example():
1)     call(public * Employee.get*(..));
```

Listing 2.19 *Using a method pattern in an event matching pointcut*

By using the bold faced method pattern in Listing 2.19, the pointcut example matches any method call where the corresponding method is 'public', has an

arbitrary return type '*', is within the type 'Employee', starts with the 'get' prefix and has an arbitrary number of parameters '(..)' with arbitrary types. For more information on type and other patterns in AspectJ, we refer to [23].

Parametrizable pointcuts (PPC.PP)
There are languages, e.g. JAsCo, that offer parametrizable pointcut definitions. A common characteristic of such languages is that a pointcut definition has two

```
0) /* JAsCo */
1) class Sample{
2)   hook Example2{
3)     Example2(m1(..),m2(..)){ call(m1) && withincode(m2); }
4)
5)     after(){ ... }
6)   }
7) }
8)
9) connector SampleConnector{
10)   Sample.Example2 ex =
11)     new Sample.Example2( * *.set*(*), * Employee.*());
12)}
```

Listing 2.20 *Hook and Connector definition in JAsCo*

parts: a generic and an application specific one. The generic part of the pointcut definition does not contain any application specific code; hence, it can keep the unit in which it is located generic (i.e. reusable). The application specific part of the pointcut contains information, e.g. signatures, or types, that are specific to a particular application; hence, this part is located in another module.

Listing 2.20 shows a pointcut definition in JAsCo. The constructor of the hook (line 3) contains the generic part of the pointcut definition; the formal parameters are bound to the pointcuts in the body of the constructor. In the connector SampleConnector, the instantiation of the hook (line 11) contains the actual parameters that correspond to the formal parameters in line 3.

Unification in Pointcuts (PPC.UN)
Various AOP languages have pointcut languages implemented on the basis of logic languages, such as Prolog. A common characteristic of these pointcut languages is that they make use of the features of logic languages, such as *unification*. For example, the selector language of Compose* - to designate struc-

tural join points- also supports unification. Listing 2.21 (lines 9-10) illustrates this by a simple example. The selector example2 uses two predicates in its specification. The first predicate binds the class Employee, if it exists, to the variable C. The second predicate will hold the same value for variable C, and it will bind the list of classes inherited from Employee to the variable AnyRes. The selector uses AnyRes as a result set (indicated before the symbol | in line 9), the filtermodule sampleModule will be superimposed on every class assigned to this variable (line 12). Note that the comma after each predicate means logical

```
0) /* a pointcut definition as a filter element */
1) concern Sample{
2)   filtermodule SampleModule{
3)     inputfilters
4)        e: Error = { True => <*.set*> }
5)   }
6)
7)   superimposition{
8)     selector
9)       example2 = { AnyRes | isClassWithName(C,'Employee'),
10)                            inheritOrSelf(C,AnyRes};
11)    filtermodules
12)      example2 <- SampleModule;
13)  }
14)}
```

Listing 2.21 *Selector definition and message filtering in Compose**

AND composition. CARMA, Sally [20], LogicAJ [26] are other examples of languages that use unification.

Logic Rules as Compound Pointcuts (PPC.RB)
Compound pointcuts, i.e. a composition of pointcuts, can also be expressed by the use of multiple (Prolog-like) logic rules in AOP languages that are founded on the concepts of logic languages. For instance, CARMA supports this type of pointcut composition; Listing 2.22 illustrates it by an example. In line 0 and

4, two rules are declared with the same identifiers and parameters. These two

```
0) changesState(?class,?selector) if
1)   shadowIn(?class,?selector,?sp),
2)   assignmentShadow(?sp,?variable).
3)
4) changesState(?class,?selector) if
5)   shadowIn(?class,?selector,?sp),
6)   messageShadow(?sp,?rcvr,?msg),
7)   selfReceiver(?rcvr),
8)   changesState(?class,?msg).
```

Listing 2.22 *Logic rules as compound pointcuts in CARMA*

logic rules gather all state changing methods: the first one designates those methods that assign to a variable of the class; the second one designates methods that self-call a method that assigns a variable. When changeState is used in a pointcut expression both rules will be evaluated by the resolution engine of Prolog.

Advice-Pointcut Composition (APC)
Advices are bound to pointcuts to assign the crosscutting behavior to join point where the crosscutting behavior has to be executed. We have identified two alternatives of the advice-pointcut composition in most AOP approaches: *direct advice-pointcut binding* and *separate binding specification*.

Direct Advice-Pointcut Binding (APC.DAB)
In several aspect-oriented languages the definition of an advice also contains the pointcut - that designates the join points - where the advice should be executed. That is, an advice has a reference to a pointcut in its specification. As a result, an advice cannot be reused and associated with another pointcut, for instance, in another application. This problem can be circumvented by associating the advice with an *abstract pointcut*. (We will discuss the abstract point-

```
0) /* direct advice-pointcut binding */
1) pointcut tracedMethods():...;
2)
3) before(): tracedMethods(){
4)   /* crosscuting behavior */
5) }
```

Listing 2.23 *Direct binding for advice-pointcut composition in AspectJ*

cut mechanism in detail in the next section.) Listing 2.23 shows a simple example of the direct advice-pointcut binding in AspectJ: the declaration of the before advice contains the reference to the pointcut tracedMethods.

Direct advice-pointcut binding can be found, for instance, in AspectJ, AspectC++, JAsCo and CARMA.

Abstract Pointcut-Advice Binding (APC.DAB.ABS)
Abstract pointcuts are introduced with the keywords abstract pointcut. An abstract pointcut definition consists of only the identifier of the pointcut and the formal parameters; i.e. there is no concrete pointcut specification after the formal parameters (line 1 in Listing 2.24). On the other hand, the abstract pointcut, like a normal pointcut, can be bound to advices (line 3 in Listing 2.24) as well. The abstract pointcut can be realized in another aspect which is inherited from the one that contains the abstract pointcut specification (lines 7-9 in Listing 2.24).

The benefit of the abstract pointcut definition is that it allows for deferring the time of specifying the concrete pointcut. Thus, it supports defining generic (reusable) aspects and their customization to different applications. The customization of an abstract pointcuts is illustrated in two different application contexts (lines 7-9 and 12-14 in Listing 2.24).

```
0) public abstract aspect GenericLogger{
1)    abtract pointcut tracedMethods();
2)
3)    before(): tracedMethods(){ /* logging behaviour */ }
4) }
5)
6) /* Application A. */
7) public aspect FigureLogger extends GenericLogger{
8)    pointcut tracedMethods(): call(* FigureElement+.*(..));
9) }
10)
11)/* Application B. */
12)public aspect EmployeeLogger extends GenericLogger{
13)   pointcut tracedMethods(): call(* Employee+.*(..));
14)}
```

Listing 2.24 *Abstract pointcut definition in AspectJ*

Various aspect-oriented languages, e.g. AspectJ, AspectC++ and Sally support the concept of the abstract pointcut definition.

Separate Binding Specification (APC.SBS)
Advices can be independently specified from any particular pointcut. In this case, a separate specification binds an advice to a pointcut. Listing 2.25 shows an example binding specification in AspectWerkz. The advice is specified as a method (called traceEntry) within a class (line 1-5), the pointcut traceMethods is specified in a weaving specification within an XML tag (line 7). The binding specification (line 9-10) binds the method traceEntry as a before advice to the pointcut traceMethods. The consequence of this technique is that advices can be

```
0) /* separate binding specification */
1) public class Tracer{
2)   public void traceEntry(JoinPoint thisJP){
3)     /* crosscuting behavior */
4)   }
5) }
6)
7) <pointcut name="tracedMethods"  expression="…" />
8) <aspect name="TracerAspect" class="Tracer">
9)    <advice name="traceEntry" type="before"
10)           bind-to="tracedMethods"/>
11)</aspect>
```

Listing 2.25 *Separate binding specification for advice-pointcut composition in AspectWerkz*

reused and associated with different pointcuts in different binding specifications. That is, the separate binding specification is an alternative for the reuse of advices, as abstract pointcut are. Note that the separate binding specification specifies when, e.g. before or after, the crosscutting behavior - represented by a method - has to be executed. That is, in contrast with direct advice-pointcut binding, this alternative also allows for reusing the same method to express different types of advices.

2.6 Advices

Figure 2.6 illustrates the design dimensions of the *advice* concept. We iterate and discuss them in detail in the subsequent sections.

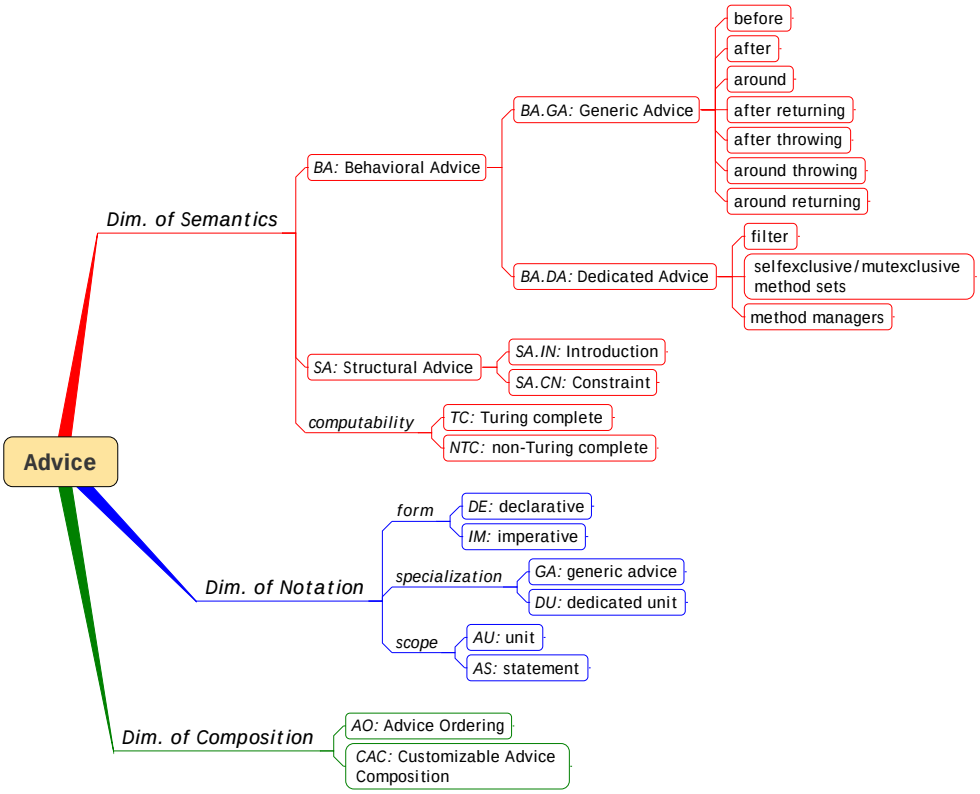

Figure 2.6 *The advice concept*

2.6.1 Dimension of Semantics

Similarly to the categories of join points (i.e. behavioral or structural), we distinguish two main categories of advices: *behavioral* and *structural* advices. In the following section we discuss these categories.

Behavioral Advices (BA)

Behavioral advices are executed on behavioral join points. This also means that these advices are coupled with behavioral pointcuts. The execution of these advices is automatically triggered when the control flow reaches the join point. (That is, as we wrote before, these advices are not explicitly called as opposed to the *method* construct of traditional object-oriented languages.) We distin-

guish further two sub categories of behavioral advices: *generic advices* and *domain-specific advices*.

Generic Advices (BA.GA)
Generic behavioral advices are the units of aspect-oriented languages to formulate the crosscutting behavior in terms of the instructions of generic programming languages, such as Java, for example.

Table 2.26 *Behavioral advices*[1]

	Example	
Advice type	*Language*	*Advice*
execution before the jp	AspectJ	before()
execution after the jp	AspectJ	after()
execution instead of the jp	AspectJ	around()
execution after the jp: when an exception is thrown	AspectWerkz	after() throwing
execution after the jp on a given return type, without exception	AspecWerkz	after() returning
execution instead of the jp: when an exception is thrown	JAsCo	around() throwing
execution instead of the jp on a given return type	JAsCo	around() returning

1. Note that the table presents only example cases, one particular advice can be supported by more aspect-oriented languages.

The generic before, after and around advices are supported by various AOP languages, such as AspectJ, AspectC++, AspectWerkz and JAsCo. The before and after advice are executed before, respectively, after the original join point (i.e. the intercepted event) is executed, while the around advice is executed instead of the original join point.

AOP languages provide a mechanism by which the original behavior can be performed within the context of the around advice. This can be a special

keyword - proceed - of the AOP language or a method in the join point type. We discussed this in detail in section 2.4.2.

It is important to note that the after advice is always executed, regardless if an exception has been thrown in the execution of the method of a join point (i.e. the *intercepted* method). For this reason, a specialized version of the after advice - called after throwing - is supported in a couple of AOP languages (AspectJ, AspectWerkz, JAsCo). This advice may have a parameter that expects the type of an exception and in that case, it executes only when an exception is thrown in the intercepted method. If there is no type specified, the advice will always execute whenever an exception has been thrown.

There is another specialized version of after, called after returning. This type of advice also expects a type as parameter. It executes when the method returns normally (without throwing an exception) *and* the actual type that is returned is the type by which the advice is declared. AspectJ, AspectWerkz and JAsCo support this type of advice.

Note that these types of advices (after throwing and after returning) practically specify extra conditions (type of exception and return value) for the activation of an advice. In this sense, these clauses (throwing and returning) show a similar functionality with the *context matching pointcuts*.

JAsCo supports two other types of behavioral advices: around throwing and around returning. These are similar types to the advices we discussed before; the difference between them is that these advices do not execute after but instead of the original join point.

Domain Specific Advices (BA.DA)
There are languages where advices have domain-specific (or dedicated) semantics. The common characteristic of these language concepts is that the advice has very specialized semantics, as opposed to the *generic advice* concept that is formulated in terms of a Turing-complete programming language. One good example for the *dedicated advice* concept is the *filter*

construct of Compose*. Listing 2.27 shows a simple filtermodule specification that refers to two filters.

```
0) filtermodule TakeCredits{
1)  externals
2)   credits : Jukebox.Credits = Jukebox.Credits.instance();
3)  conditions
4)   enoughCredits : credits.payed();
5)  inputfilters
6)   err : Error = { enoughCredits => [*.*],
7)                   true ~ > [*.play ] };
8)   pay : Meta = { [*. play ] credits.withdraw }
9) }
```

Listing 2.27 *An example filtermodule in Compose**

The content of this filtermodule has already been discussed in section 2.5.1 under Listing 2.14. The specification of the filtermodule uses two pre-defined filter types (line 6 and 8): Error and Meta. These filter types have pre-defined semantics: the Error filter controls (allows or disallows for) the execution of the intercepted message (i.e. the execution of the original join point) depending on its properties and certain states in the system. The Meta filter, if it matches the signature play, specifies that the intercepted message is reified and sent as a parameter to a method (withdraw) to be executed. Besides these filter types, Compose* has other predefined filter types, such as Wait, Substitute and Dispatch.

COOL [24] is a domain-specific aspect-oriented language on synchronization. In COOL, the method sets selfexlusive and mutexlusive, as well as the method manager declarations, act as dedicated advice specifications.

We have identified two benefits of using advices with dedicated semantics. As compared to the generic advices of Turing-complete languages, advices with restricted semantics allow for reasoning, for instance, about their composition, or conflict detection in their composition [16]. Another advantage of this type of advices is that they allow for a very efficient reuse of a domain specific solution, as optimized semantics for a specific domain. For the reason, on the other hand, they are not suitable for expressing generic behavior.

Structural Advices (SA)
In general, structural advices are related to the structure of application; typically, they are performed on structural join points and behavioral join points that can be determined in compile time[9].

Introductions (SA.IN)
Introductions perform modifications on the structure of a program; therefore, these advices are coupled with structural join points. Introductions are called *inter-type declarations* in AspectJ. This type of structural advice introduces new program elements into existing ones. Field and method introductions into classes are quite common in a couple of AOP languages (e.g. AspectJ, AspectC++, Sally). The following listing shows some simple introductions in AspectC++[10].

```
0) pointcut shapes() = "Circle" || "Polygon";
1)
2) advice shapes() : bool m_shaded;
3) advice shapes() : void shaded(bool state) {
4)    m_shaded = state;
5) }
6) advice "%Object" : baseclass(MemoryPool);
```

Listing 2.28 *Some example introductions in AspectC++*

First, a pointcut is defined that designates two classes, Circle and Polygon, as structural join points. The first advice (line 2) introduces a Boolean variable into those classes. The second advice (line 3-5) introduces a method that sets the state of that variable. The last advice (line 6) specifies that every class with a name that ends with "Object" is derived from the class MemoryPool. (The change of a superclass can be achieved by the statement declare parent in AspectJ.)

Classes can also be used as structural join points for introducing *mixins*. (With mixins the class definition defines only the attributes and parameters associated with that class; methods are left to be defined elsewhere as, in CLOS [9],

9. This means that the pointcut expression that designates the join point cannot contain information that can be determined in runtime, e.g. dynamic condition.
10. Examples are taken from [35].

generic functions.) The introduction of mixins is supported by, for instance, JBossAOP and AspectWerkz.

The introduction of annotations (also called *custom attributes* in .NET, *meta-data facility* in Java) is also supported in AspectJ and Compose*. In these languages, annotations can be introduced on various types of program elements, such as fields, methods, classes and packages.

Note that dependency problems may arise when the pointcuts are resolved and the structural advices are performed in the compilation phase. Typically, a dependency problem may occur when a pointcut depends on a structure that is affected by an introduction.

Constraints (SA.CN)
AspectJ and JBossAOP have other types of simple structural advices. In AspectJ, the construct `declare error` is coupled with statically determinable behavioral pointcuts and throws an instance of `IllegalAccessException` whenever the designated join point occurs. This construct is suitable for expressing architectural constraints, e.g. illegal calls to certain units. The statement `declare warning` works in a similar, except that prints a warning message instead of throwing an exception. The common characteristics of these constraint-like advices are that they are coupled with behavioral pointcuts that can be evaluated in compile-time, and they are performed also in the compilation phase.

Sub-dimension of Computability

Non-Turing Complete Advice (NTC) The crosscutting behavior is often expressed in terms of non-Turing complete specifications, typically, in case of *declarative advice* concepts. Note that we have recognized that Compose* and COOL - the languages that we considered in this analysis - had non-Turing-complete advice specifications. Naturally, this does not mean that it is not possible to ’create’ a *declarative advice* construct which is Turing-complete.

Turing Complete Advice (TC) The crosscutting behavior is expressed in terms of Turing complete specification. Typically, imperative advice constructs have this characteristic of computability.

2.6.2 Dimension of Notation

Regarding the representation of advices, we have observed a wide range of alternatives in the current AOP languages. We defined three subdimensions to characterize the representation of advices: *form*, *specialization* and *scope*. Note that these subdimensions are orthogonal to each other: a concrete advice concept of a given aspect-oriented language can be characterized in any subdimension.

Dimension of Form

Imperative Advice (IM) The imperative advice concept includes those languages that have methods and method-like units for the advice concept and the behavior is expressed in terms of instructions of generic programming languages, such as Java.

Declarative Advice (DE) The definition of the advice is a sort of declarative specification that expresses the crosscutting behavior. The declarative definition can cover various language concepts, such as those that we have discussed under the section Domain Specific Advices.

Dimension of Specialization

Generic Advices (Methods) (GA) Several aspect-oriented languages and frameworks (e.g. AspectWerkz, JBossAOP, JAC) use the *method* construct of object-oriented languages to specify the crosscutting behavior. A common feature of these languages is that the actual join point instance is passed as a parameter to a method; i.e. the method always has a formal parameter with the type of the join point. Listing 2.3 (in page 31) shows a simple example for such a method.

Dedicated Units as Advices (DU) Typically, the definition of the advice also includes a keyword that specifies that the advice should be performed before, after or around the actual join point. That is, the definition of the advice contains information about the type of the advice, as opposed to the case where only regular methods represent the advice. For instance, AspectJ, AspectC++, JAsCo have this type of advice-unit within the context of an aspect (or hook, in case of JAsCo).

Dimension of Scope
In this dimension, we distinguish between *advice units (AU)* and *advice statements (AS)*. An *advice statement* differs from an *advice unit* in the sense that it represents the advice behavior as a single statement. That is, this construct does not have the usual characteristics of a unit, such as the scope, for instance. Statements such as declare parent, declare warning in AspectJ or the selfmutex method set of COOL can be classified as declarative advice statements.

2.6.3 Dimension of Composition

Advice Ordering (AO)
It is a common phenomenon in AOP that not just a single but several advices have to be executed at the same join point. (We call such a join point *shared join point.*) In other words, advices are coupled with pointcuts that designate a common set of join points. As advices, in general, are executed sequentially in aspect-oriented languages, many of them offers language concepts to specify the execution order of advices at shared join points:

- **a.** declare precedence -- AspectJ /advice precedence --AspectC++
- **b.** stack --JBossAOP
- **c.** *precedence strategy* --JAsCo

AspectJ has the statement declare precedence to define partial ordering relationships between aspects. This, implicitly, defines an ordering relationship between the advices of those aspects in the case of shared join points. Note that the granularity of the ordering specification is the level of aspects; this implies that two different advices of two aspects with a given precedence cannot have different order specification.

The same granularity is applied in AspectC++; the difference with the AspectJ approach is that the precedence of aspects can be specified on particular join points, whereas the precedence of aspects is generic to every join point in AspectJ.

In JBossAOP, the *stack* construct allows for defining a predefined, ordered set of advices/interceptors that can be referred to within a binding element. Listing

```
0) <stack name="stuff">
1)    <advice name="timer" aspect="org.jboss.TimingAspect"/>
2)    <advice name="trace" aspect="org.jboss.TracingAspect"/>
3) </stack>
4)
5) <bind pointcut="execution(* POJO->*(..))">
6)       <stack-ref name="stuff"/>
7) </bind>
```

Listing 2.29 *The stack construct of JBossAOP and its usage*

2.29 shows an example of the stack construct and its usage. The first part of the listing (between lines 0 and 3) presents a stack named stuff that enumerates two advices. This stack is referred to in a binding specification, in line 8-10. The benefit of this approach, besides the ordering, is that the predefined set can be (re-)referenced to in different binding specifications. In addition to the stack construct, JBossAOP has the *precedence* construct to define a relative ordering of the execution of advices.

In JAsCo, the *connector* construct allows for specifying a precedence for the execution of hooks. The granularity of the specification is the level of advices; this means that different advices of the same aspect can have different execution order.

Custom Advice-Advice Composition (CAC)
There are aspect-oriented approaches that provide a means for the customization of the composition of aspects at shared join points. In these languages, it is possible to express complex interactions among aspects (and advices) besides their execution order.

In JAsCo, the interface CombinationStrategy provides this functionality. An implementation of this interface works like a filter on the list of hooks that are applicable at a certain point in the execution. Each combination strategy needs to implement the method validCombinations that filters the list of applicable hooks and possibly modifies the behavior of individual hooks.

In the Aspect Moderator Framework [14], Constantinides et. al. propose a dedicated class, called *moderator*, to manage the execution of aspects at shared join points. The moderator class, for example, can express the conditional execution of aspects. The application programmer can implement new moderator classes: it is possible to introduce other activation strategies as well.

2.7 Aspects

Figure 2.6 illustrates the design dimensions of the aspect construct. We iterate and discuss them in detail in the subsequent sections.

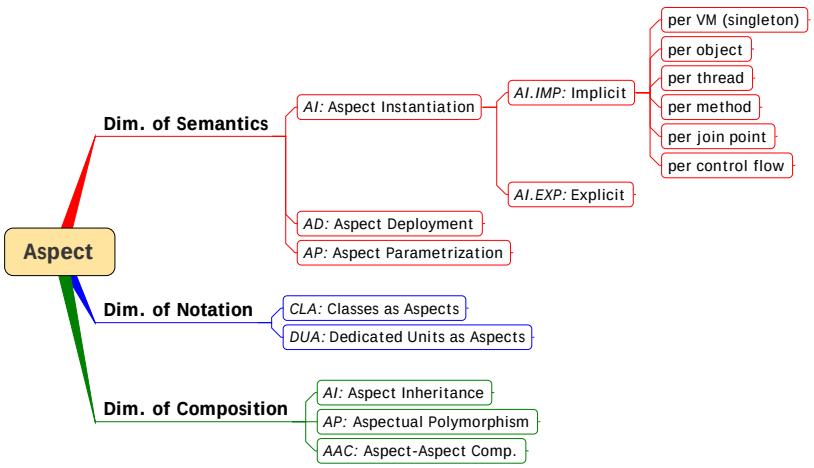

Figure 2.7 *The aspect concept*

2.7.1 Dimension of Semantics

The primary role of the *aspect* concept is to encapsulate the previously discussed language concepts - pointcuts and advices - into a module of an aspect-oriented language. In addition to these constructs, aspects may contain additional, regular methods and member variables in various aspect-oriented languages. Besides, there are other important concerns, e.g. *aspect instantiation* and *deployment*, that aspects may express. We discuss these concerns in the following sections.

Aspect Instantiation (Scope of aspect instances) (AI)
In general, aspects are instantiated and advices run in the context of aspect instances. An aspect instance has a very important property: its execution scope - e.g. in terms of object instances - on which the aspect instance operates. For example, it is common that only one instance of an aspect is created for the complete execution of a program. The advice of this aspect runs whenever the corresponding join point happens in any particular instance of the system. A new aspect instance can also be created and assigned to each new instance of a class; i.e. the scope of the aspect instance is one object instance. In this case, the advice of the aspect runs if the join point occurs in the object instance which is associated with the aspect instance. In short, the instantiation of aspects determines their scope of operation.

In general, AOP languages handle the instantiation of aspects in two ways: (a) most AOP languages handle the instantiation of aspects in an implicit way by offering built-in instantiation strategies; (b) some AOP languages offers language concepts to handle instantiation of aspects in an explicit manner.

Implicit instantiation (AI.IMP)
As opposed to the explicit instantiation mechanism of OOP languages, aspects are implicitly instantiated by the weaver in many aspect-oriented languages. For this purpose, these languages offer standard instantiation strategies that are normally part of the aspect specification. This instantiation strategy determines the scope - e.g. in terms of object instances, control-flow, threads - of an aspect instance in which the aspect instance "operates". The following instantiation strategies are known in the current AOP languages:

Table 2.30 *Scope of aspect instantiation*[1]

	Example	
Scope of Instantiation	*Language*	*Keywords*
per vm (singleton)	AspectJ	default
per object	AspectJ	perthis(), pertarget()
per join point	AspectJ	percflow(),
per method	JAsCo	permethod

Table 2.30 *Scope of aspect instantiation*[1]

per thread	JAsCo	perthread
per class	AspectWerkz	perClass

1. Note that the table presents only example cases, one particular scope of instantiation can be supported by more aspect-oriented languages.

The *per VM* (virtual machine) strategy specifies that only one instance (i.e. a singleton) is created for the full execution of the application. The *per object* strategy specifies that a new aspect instance should be created for each new object instance. Listing 2.31 shows a simple aspect specification that does per object instantiation.

```
0) public aspect SimpleCaching
1)      pertarget(call Circle.getArea()){ ...}
2)
3) /* getting the corresponding aspect instance of
4)  * an object instance */
5) ...
6) void test{
7)    Circle c = new Circle();
8)    ...
9)    SimpleCaching sc = SimpleCaching.aspectOf(c);
10)}
```

Listing 2.31 *An example of per object (pertarget) instantiation in AspectJ*

In this example, the clause pertarget of AspectJ (line 1) specifies that a new instance of SimpleCaching is created for every new target of the calls Circle+.getArea(), i.e. for every new instance of Circle. (Similarly, the clause perthis assigns the aspect instance to the sender of the message in AspectJ.) In AspectJ, every aspect has a static method, called aspectOf, that returns the corresponding aspect instance of an object instance. In line 9, we return the instance of SimpleCaching that corresponds to the given instance of Circle.

The execution scope of an aspect instance can be limited in terms of control-flow as well. The *per join point* strategy specifies that a new instance of an aspect is created whenever the execution enters a join point specified by a pointcut. AspectJ, for instance, supports this type of instantiation by the clauses percflow and percflowbelow. The *per thread* strategy (JAsCo, older version of

AspectWerkz) instantiate and (limit the scope of) an aspect per thread. In JAsCo, it is possible to define custom instantiation strategies by custom aspect factories.

Explicit instantiation (AI.EXP)
There are also languages that support the explicit instantiation of aspects (by using constructors), such as JAsCo, AspectS and CaesarJ[30]. The common characteristic of these languages is that aspects are represented by first class abstractions that can be instantiated.

Aspect Deployment (AD)
Although the concept of aspect deployment exists in a couple of aspect-oriented languages, the concepts of *aspect instantiation* and *aspect deployment* are often not separated. In the reference model, we define the role of aspect deployment as an ability to *enable* and/or *disable* the operation of aspects. For example, a disabled (undeployed) aspect cannot intercept the messages defined in its pointcuts. That is, the difference between aspect instantiation and deployment is that aspect instantiation determines the working scope of an aspect instance, while aspect deployment determines whether the aspect instance is active for the given scope. Aspect deployment is supported by CaesarJ, AspectWerkz, JBoss and JAsCo, for example.

Aspect Parametrization (AP)
AOP approaches that have implicit instantiation strategies and represent aspects by standard object-oriented classes and XML specification, e.g. AspectWerkz and JBossAOP, support the concept of aspect parametrization. This concept allows for passing external parameters to aspects. By using this construct, an aspect can be reused in different application contexts, as different parameters can provide different configurations for the aspect. Listing 2.32

```
0) <aspect ... >
1)     <param name="timeout" value="10"/>
2) </aspect>
```

Listing 2.32 *Aspect Parametrization in AspectWerkz*

shows an example for this. In this case, the parameter can be accessed in the aspect behavior implementation by using the API AspectContext.getParameter("timeout") in AspectWerkz.

Note that this construct is not necessary in languages that support the explicit instantiation of aspects through the use of constructors.

2.7.2 Dimension of Notation

Classes as Aspects (CLA)
In some AOP approaches (e.g. AspectWerkz, JBoss), advices are represented by regular object-oriented methods, aspects are represented by standard classes. This type of representation uses an additional specification, for instance, placed in a separate XML descriptor, that describes the aspect-specific behavior, e.g. advice binding, instantiation strategy, etc. of a class. Listing 2.25 (in page 50) shows a simple example of such a class and specification.

Dedicated Units as Aspects (DUA)
In most AOP approaches, aspects are represented by dedicated language units. This type of unit encapsulates the advices, pointcut and the other related specifications, e.g. *aspect instantiation*, *advice ordering*, etc., of AOP languages. We distinguish two alternatives of this construct: *context units (DUA.CTX)* and *container units (DUA.CON)*. *Context units* provide context (state) information that can be referred to by the encapsulated units. In contrast, *container units* act as simple containers for the encapsulated units and they do not provide context information. Container units are, for instance, the *concern* construct of Compos*, and the *connector* construct of JAsCo.

2.7.3 Dimension of Composition

Aspect Inheritance (AI)
Languages that have dedicated language concepts to represent aspects, e.g. AspectJ and AspectC++, provide an inheritance relationship between aspects in order to support the reuse of advices. In this type of relationship, advices can be reused in the inherited aspects by redefining the pointcuts that they are attached to. It is important to note that we consider aspectual polymorphism in case of such aspect inheritance relationship in which not only pointcuts but also advices can be overridden.

Aspectual Polymorphism (AP)
A few languages support aspectual polymorphism; i.e. the inheritance of aspects in a way, that their advices can also be overridden. In this type of composition, advices are also bound late, similar to the late binding mechanism in object-oriented languages. Aspectual polymorphism is supported by CaesarJ and JAsCo (through the use of refinable methods in hooks), for example.

Aspect-Aspect Composition (AAC)
The composition of advices is often specified on the granularity level of aspects. The literature refers to this type of composition as *aspect-aspect composition* or *aspect composition model* [12]. In the subsection Custom Advice Composition of section 2.6.3, we discussed several alternatives that express the composition of aspects based on the composition of advices.

2.8 Special Languages

Although DemeterJ [28] and HyperJ [34] do not *explicitly* apply AOP concepts, many of their language concepts show similarity to the concepts that we discussed in the previous sections. In this section, we discuss the concepts of DemeterJ and HyperJ with respect to the reference model and the previously discussed AOP concepts.

2.8.1 DemeterJ

The join point model of DemeterJ is the class structure of an application; i.e. DemeterJ deals with *structural join points*. The structure of the application is described as a class grammar that describes all used classes and their relations. Listing 2.33 shows a simple grammar that describes the structure of an application containing Line, Point, XCoord and YCoord classes.

```
0) Line = <p> Point .
1) Point = <x> XCoord <y> YCoord .
2) XCoord = .
3) YCoord = .
```

Listing 2.33 *A simple example grammar*

A **traversal strategy** describes a set of paths in the class graph; i.e. it is a query that selects a set of structural join points in a given *order*. For this reason, traversal strategies can be considered as *structural pointcuts*. The simplest

form of traversal strategies are called **single edge strategies**; e.g. {Line -> XCoord} describes a single edge traversal strategy from Line to XCoord. DemeterJ has a wider range of constructs to specify traversals and to create other sorts of traversals; the interested readers can find more details in [28].

In DemeterJ, the crosscutting behavior is expressed in terms of **traversal methods**, **visitors**, **visitor methods** and **adaptive methods**.

A **traversal method** of DemeterJ acts as a sort of *binding specification*: it consists of a name, a traversal strategy and a list of possible visitor classes. A traversal method performs a traversal on the instance graph of a given class graph. It takes a visitor as an argument and executes its behavior during the traversal. Listing 2.34 shows a simple traversal method that iterates over all points of a line for DisplayVisitor.

```
0)   traversal allPoints(DisplayVisitor) {
1)     to { Point };
2)   }
```

Listing 2.34 *A simple traversal method*

The **visitor** class of DemeterJ can be considered as an *aspect*. A visitor class may contain three types of **visitor methods**:

- Before - A before method is invoked as soon as the traversal reaches an object of the given class.
- After - An after method is invoked after a traversal has finished traversing an object of the given class, on its way back out of the object.
- Around - An around method is invoked in an object of the given class instead of continuing along the traversal. The continuation of the traversal must be explicitly invoked if desired, by using the apply method on the special variable subtraversal, like so:

```
0)     around Point (@
1)        ...
2)        subtraversal.apply();
3)     @)
```

Listing 2.35 *An example visitor method*

From this perspective, a **visitor method** can be considered as the *advice* concept of the reference model. The special variable subtraversal can correspond to the *join point instance* in the representation dimension of the join point concept. Note that DemeterJ uses *dedicated units* for representing advices and aspects, while DJ [27] uses the normal *class* and *method* constructs of Java for representing the above mentioned visitor classes and methods.

The interested readers can find more information about DemeterJ and DJ in [28] and [27].

2.8.2 HyperJ

To discuss the concepts of HyperJ briefly, we refer to [12] *"...HyperJ does not use the terms 'joinpoint model' and 'pointcut language' because it is not based on a dominant decomposition approach such as other aspect languages. Instead of expressing an aspect that crosscuts a base program (in a dominant decomposition), HyperJ allows to express multiple decompositions of the program as separate 'hyperslices' (previously called 'subjects' in Subject-oriented Programming). Each decomposition is called a hyperslice. The intention is that each hyperslice contains the implementation of a single concern using the standard programming language constructs (i.e. it is implemented in standard Java). A set of hyperslices can then be combined into a hypermodule using composition rules. The resulting hypermodule contains all concerns implemented in each hyperslice in the composition."*

In HyperJ, concerns are implemented by regular Java classes. These classes, their fields and methods as well as their namespace can be considered as the s*tructural joinpoint* model of HyperJ. (They are called **implementation artifacts** in the terminology of HyperJ.)

The concerns implemented by different classes need to be identified; this is done by the so-called **concern mapping**. The concern mapping assigns the

implementation artifacts to a set of dimensions and concerns within those dimensions. HyperJ is capable of expressing the following mappings:

- Package mapping: indicates that the entire contents of a package implement a concern in a certain dimension
- Class and interface mapping: indicates that a class or interface implements a concern in a certain dimension
- Operation mapping: indicates that an operation addresses a concern in a certain dimension
- Field mapping: indicates that a field addresses the implementation of a concern in a certain dimension

In the following listing we show two simple mapping as examples:

```
0) package project.tests : Feature.Verification
1) operation check : Feature.Check
```

Listing 2.36 *Two simple concern mappings*

The first mapping specifies that the package project.tests addresses the Verification concern in the Feature dimension. The second mapping specifies that any method with name check pertains to the concern Feature.Check. Since **concern mappings** addresses, in general, multiple program elements, they can be considered as *structural pointcut* specifications in HyperJ.

HyperJ does not deal with constructs such as advices and aspects, concerns are expressed in terms of regular object-oriented methods and classes. For this reason, HyperJ is considered as a *symmetrical* approach, as opposed to the languages that we discussed in the reference model.

Regarding the dimension of notation, HyperJ has a dedicated specification (more like a configuration) language to express certain compositions among the identified concerns.

A **hypermodule** specification describes a particular integration of the units pertaining to some selection of concerns. It identifies **hyperslices** that are to be

integrated in terms of the concerns and specifies **integration relationships**. A hypermodule is defined as follows:

```
0) hypermodule <<hypermodulename>>
1)   hyperslices:
2)     <<dimensionName1>>.<<concerName1>>,
3)     <<dimensionName2>>.<<concerName2>>,
4)     ...
5)   relationships:
6)     <<mergeByName or nonCorrespondingMerge or
7)       overrideByName>>;
8)     <<other relationships>>
9) end hypermodule;
```

Listing 2.37 *A hypermodule specification*

A hyperslice (line 1) is a selection of the relevant concerns that need to be integrated into the hypermodule. The relationships clause (line 5) expresses how the units that implement a concern (i.e. implementation artifacts) are effectively integrated. In general, there is always an overall (main) integration relationship specified and other types of relationships can refine that. There are three types of main integrations relationships:

- mergeByName: the units in the hyperslices that have the same name are integrated into a new unit that merges both.
- nonCorrespondingMerge: the units in the hyperslices that have the same name are not integrated into a new unit.
- overrideByName: the units in the hyperslices that have the same name are integrated in the sense that the last unit in the relation overrides the previous ones. Overriding only affects methods. If classes are combined by overriding, then their corresponding methods override.

Note that these **integration strategies,** considering the dimension of composition**,** explicitly express a sort of custom *aspect-aspect composition* (as aspects are implemented by classes), and also *custom advice composition*. For instance, if methods have the same name in the hyperslices, they will be "merged" into one method.

There are other, special types of relationship in HyperJ that may correspond to certain design alternatives in our model. The relationship **bracket** allows to integrate methods similarly to the *before* and *after advice* constructs in the reference model. The relationship **order** specifies the merging order of methods that are integrated by a merge relationship. To do so, it specifies a partial ordering on methods. From this perspective, one might consider that the relationship **order** corresponds to *advice ordering* in our reference model. However, it is important to note that this relationship expresses an ordering among the *methods* to be integrated by a merging relationship and not by the **bracket** relationship (i.e. advices). In other words, it does not correspond to the *advice ordering* construct of the reference model.

The interested readers can find more information about HyperJ in [34].

2.9 Background on the Compose* language

In this section, we present the language constructs of the aspect-oriented language Compose*. Compose* is implemented on .NET platform and founded on the Composition Filter model. (The Composition Filter model is originated from the Sina language [3].) In sections 2.9.1 and 2.9.2, we explain the concepts of the Composition Filter (CF) model. Section 2.9.1 and 2.9.2 are adapted from [8]. Section 2.9.3 presents and discusses the language constructs of Compose* with respect to the previously introduced reference model.

2.9.1 Concern Instance = Object + Filters

The CF model is a modular extension to the conventional object-based model [37] used by programming languages such as Java, C++, and Smalltalk, as well as component models such as .NET, CORBA, and Enterprise JavaBeans. The core concept of this extension is the enhancement of conventional objects by manipulating all sent and received messages. This allows expressing many different behavioral enhancements since in an object-based system all the externally visible behavior of an object is manifest in the messages it sends and receives. Figure 2.8 illustrates this extension by "abstracting" the implementation object with a layer that contains filters for manipulating sent and received messages. These filters are grouped into subcomponents called filter modules. Filter modules are the units of reuse and instantiation of filter behavior. In addi-

tion to the specification of filters, the filter modules may provide some execution context for the filters.

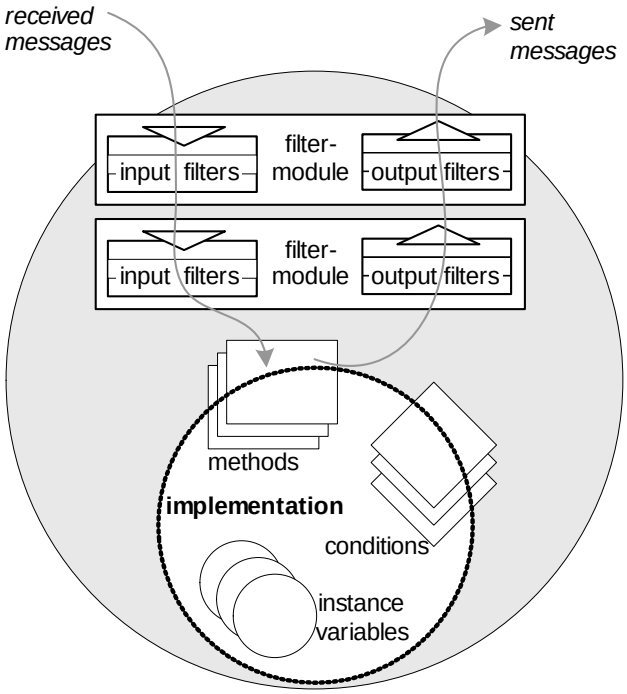

Figure 2.8 *Simplified representation of concern instances with filters*

The filters define enhancements to the behavior of objects. Each filter specifies a particular inspection and manipulation of messages. Input filters and output filters can manipulate messages that are respectively received and sent by an object. After the composition of filter modules and filters, received messages must pass through the input filters and be sent through the output filters.

The enhanced object, which we refer to as the implementation object, may be defined in any object-based language, given the availability of proper tool support for that language. The main requirement is that the object offers an interface of available methods. Two types of methods are distinguished: regular methods and condition methods (conditions for short). Regular methods implement the functional behavior of the object. They may be invoked through messages if the filters of the object allow this. Conditions must implement side-

effect free Boolean expressions that typically provide information about the state of the object.

Conditions serve three purposes:

1. They offer an abstraction of the state of the implementation object, allowing filters to consider only relevant states.
2. They allow filters to remain independent of the implementation details of the implementation object. This has the additional benefit of making the filters more reusable.
3. Conditions can be reused by multiple filter (modules) and concerns.

In summary, conditions enforce the separation of the state abstraction and the message filtering concerns.

2.9.2 Message Processing

We explain the process of message filtering with the aid of Figure 2.9. The description focuses on input filters, but output filters work in exactly the same manner. In Figure 2.9, three filters, A, B, and C, are shown. We assume sequential composition of these three filters. Each filter has a filter type and a filter pattern. The filter type determines how to handle the messages after they have been matched against the filter pattern. The filter pattern is a simple, declarative expression to match and modify messages. Typically, messages travel sequentially along the filters until they are dispatched. Dispatching here means either to start the execution of a local method or to delegate the message to another object.

Figure 2.9 illustrates how a message is rejected by the first filter (A) and continues to the subsequent filter (B). At filter (B) it matches immediately, and is modified. Then the message continues to the last filter (C). In the example, at the last filter, the message matches and is then subject to a dispatch.

Each filter can either accept or reject a message. The semantics associated with acceptance or rejection depend on the type of the filter. Typically, these are the

manipulation (modification) of the message or the execution of certain actions. Examples of predefined filter types are:

- **Dispatch.** If the message is accepted, it is dispatched to the current target of the message; otherwise the message continues to the subsequent filter (if there is none, an exception is raised) [2].
- **Substitute.** Is used to modify (substitute) certain properties of messages explicitly.
- **Error.** If the filter rejects the message, it raises an exception; otherwise the message continues to the next filter in the set [2].
- **Wait.** If the message is accepted, it continues to the next filter in the set. The message is queued as long as the evaluation of the filter expression results in a rejection [6, 7].
- **Meta.** If the message is accepted, the message is reified and sent as a parameter of a new message to a named object; otherwise the message just continues to the next filter. The object that receives the message can observe and manipulate the reified message and reactivate its execution [4].

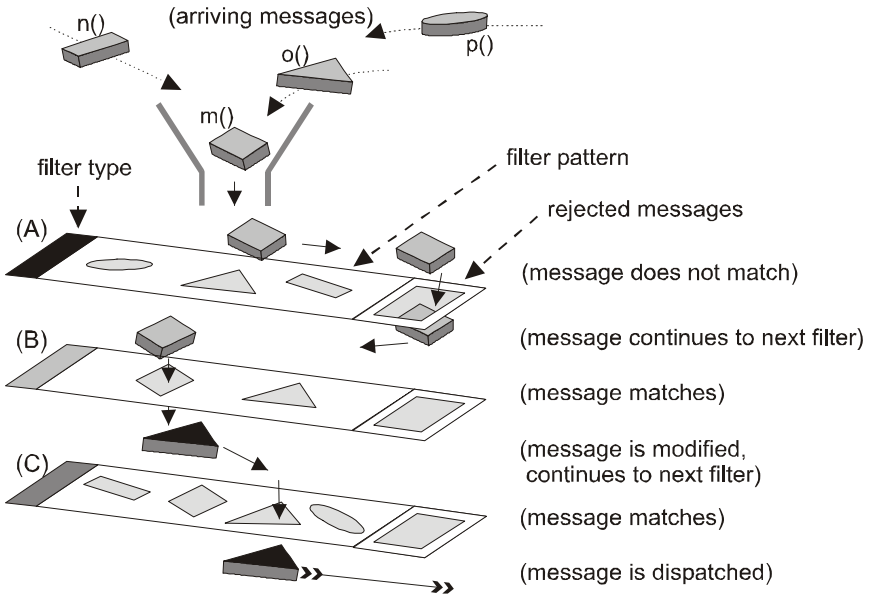

Figure 2.9 *An intuitive schema of message filtering*

2.9.3 Compose* Language

This section presents and discusses the language constructs of Compose* with the respect to the reference model: we map each language construct to the concepts of the reference model.

To explain the language constructs of Compose*, we present a simple concern specification in Listing 2.38.

```
0) concern CreditConcern {
1)   filtermodule TakeCredits {
2)     externals
3)       credits : Jukebox.Credits =
4)                        Jukebox.Credits.instance();
5)     conditions
6)       enoughCredits : credits.enoughCredits();
7)     inputfilters
8)       check : Error = { enoughCredits => [*.*] ,
9)                                  True ~> [*.play] };
10)      withdraw : Meta = { True => [*. play]
11)                                    credits.withdraw }
12)  }
13)
14)  superimposition {
15)    selectors
16)      classes = { Class | isClassWithName (Class,
17)                           'Jukebox.Jukebox ') };
18)    filtermodules
19)      classes <- TakeCredits ;
20)    }
21)  }
22)
23)  implementation in Java by CreditConcernImpl as
24)                                CreditConcernImpl.java{
25)     class CreditConcernImpl{ ... }
26)  }
27)}
```

Listing 2.38 *An example concern specification in Compose**

The **concern** construct is the main unit of modularization to represent concerns in Compose*. Listing 2.38 starts with the concern specification of CreditConcern. The **concern** construct corresponds to the *container unit*

concept of the reference model, as it encapsulates the weaving specification and the crosscutting behavior to be woven, without providing context information for the encapsulated units. A concern specification modularizes three types of language concept: **filter module** (line 1), and **superimposition** (line 14) specifications and **implementation**.

Filter modules are the units of reuse and instantiation of crosscutting behavior. Lines 1-12 of Listing 2.38 presents a filter module specification. The **filter module** construct corresponds to the *aspect* concept of the reference model, more precisely, to the *context unit* concept: it contains the specification of filters, and may also provide some execution context for the filters. The execution context can be given by two types of declarations: internals and externals. Internals are objects pertaining to the filter module, while externals are references to instances created outside the filter module and concern, that are used for representing shared state. Line 3 of Listing 2.38 shows the external declaration credits that is an instance of the type Jukebox.Credits. In addition, the filter module construct may contain condition declarations. A condition declaration declares a condition by its name and maps it to a method. This method must implement side-effect free Boolean expressions. Typically, conditions provide runtime information about the state of an object. For instance, line 5 of Listing 2.38 shows the condition declaration enoughCredits, that is mapped to the method enoughCredits() and executed on the external credits.

The next part of the filter module specification is the declaration of input and output filters. Input filters manipulate the **incoming messages** received by the implementation object, while output filters manipulate the **outgoing messages** sent by the implementation object. Therefore, Compose* deals with two kinds of behavioral join point type: *message reception* and *method call* (i.e. passing a message).

The **filter** construct corresponds to the *dedicated advice* unit of the reference model, as **filters** are the dedicated constructs to represent the crosscutting behavior in Compose*. Regarding the dimension of notation, a filter is a *declarative, non-Turing complete, dedicated unit* concept of the reference model. (However, the Meta-filter can point to a Turing-complete behavior specification, see also on page 77.) As we discussed in the previous section, there are various types of filters. Lines 8-11 of Listing 2.38 shows the declara-

tion of an Error and a Meta-filter. The declaration of a filter starts with an identifier, that is followed by a colon and the type of the filter. The filter is initialized with the **filter pattern** between the curly braces.

A **filter pattern** consists of one or more **filter elements**, connected with the sequence composition operator ','. A filter element consists of a **condition**, an **inclusion (=>)** or **exclusion (~>) operator** and a **matching expression:** <condition> =>|~> <matching expression>. For instance, in the filterement in line 8, the condition enoughCredits is composed with the matching expression [*.*] via the inclusion operator (=>). The semantics of a filterelement are that the matching expression is evaluated only if the condition on the left side is evaluated to *true*. The **matching expression** specifies a method (signature) pattern; using the inclusion operator, the filter accepts (matches) every message that corresponds the given method pattern. Using the exclusion operator, the filter accepts every message but the ones that correspond to the given method pattern. Hence, the inclusion and exclusion operators are also part of the matching expression. In our example, this means that the first filter element will accept every message if the enoughCredits is true.

As we discussed, a condition abstracts a state in a system; therefore, the **condition** construct corresponds to the *dynamic condition* concept of the reference model. The **matching expression** describes the properties, such as name, target, of the shadow point of the designated join point. Hence, the **matching expression** corresponds to the *structural join point matching pointcut* construct of the reference model. The complete **filter element** (including the matching expression) corresponds to the *message reception* pointcut when declared in the input filterset and *method call pointcut* when declared in the output filterset.

The semantics of the **sequence composition operator** (',') between the filter elements are similar to a conditional OR - when the filter element on the left side matches, the whole expression is satisfied, and no further filter elements should be considered. (Thus, the complete filter specification in line 8-9 will only accept the message play if enoughCredits is true, any other message is always accepted.) The **sequence composition operator** corresponds to the *pointcut composition operator* of the reference model. If the message does not match with any of the filter elements, and has reached the end of the filter

elements, it yields the reject semantics of the given filter type. Otherwise, the message does match and it yields the accept semantics of the given filter type. The filter declarations in the filtersets are composed with the **filter composition operator** ';'. The **filter composition operator** expresses the sequental composition of filters in a filterset; hence, it corresponds to the *advice ordering composition* construct of the reference model. It also expresses implicitly the composition of filter elements between the filter specifications; for this reason, it also corresponds to the *pointcut composition operator* construct of the reference model.

The Meta-filter type has a special semantics among the predefined filter types: it reifies the message that is matched by its filter specification, and sends it as a argument in a method call on an instance of an ACT *(AdviCe Type)* class. An ACT class is a regular object-oriented class with methods that take an instance of the type ReifiedMessage in their parameter. For instance, lines 10-11 in Listing 2.38 show an example of Meta filter specification. If the incoming message is play, the filter withdraw will reify and pass it as an argument in the method

```
0) class Credits{
1)   private int credits = 0;
2)   ..
3)
4)   public void withdraw ( ReifiedMessage m) {
5)    credits --;
6)    m.resume();
7)  }
8) }
```

Listing 2.39 *An example ACT (AdviCe Type) class*

call withdraw on an instance of JukeBox.Credits. The control flow continues by the execution of the method withdraw, in the context of a credits object. (See Listing 2.39 for more details.) By calling the method resume() on an instance of a refied message in an ACT method, the execution of the reified message can be resumed, similarly as the proceed keyword resumes the execution of the intercepted event in AspectJ. This means that the **ACT class** and **ACT method** correspond to the *aspect* and *around advice* concepts of the reference model. They are represented, respectively, by the *regular class* and *(Turing-complete) method* constructs in the dimension of notation). In this context, the **Meta-filter** acts as *separate binding specification* of the reference model, as it binds

an ACT method and class to a pointcut which is formulated by the filter elements of the meta filter.

The next part of the concern specification is the **superimposition clause**. The superimposition clause consists of **selector declarations** and **binding specifications**. Line 16 in Listing 2.38 shows an example selector declaration. The selector classes uses a logic variable, Class and a predicate, isClassWithName in its declaration. This predicate will select the class Jukebox.JukeBox and binds it to the variable Class; practically, when the selector classes is referred to, it will refer to the class Jukebox.JukeBox[11]. Selectors are predicate based queries that designate program elements, such as packages, classes, methods and fields. Hence, both the **selector** construct and the **predicates** used in a selector expression correspond to the *structural join point matching pointcut* concept of the reference model.

The selector declarations are followed by the **filter module binding specifications** in the superimposition clause. The **filter module binding specification** construct corresponds to the *separate binding specification* concept of the reference model, as it superimposes a given filtermodule on a set of classes designated by a selector. This means that a new instance of the superimposed filtermodule is associated with each new instance of the designated classes, and each incoming and outgoing message is filtered by the superimposed filtermodule instances. For instance, line 19 in Listing 2.38 shows an example of the filter module binding specification: the filtermodule TakeCredits is bound to the selector classes; hence, TakeCredits is superimposed on the class JukeBox.JukeBox.

Figure 2.10 shows the discussed mapping of the language constructs of Compose* to the reference model. For more information about Compose*, we refer to [8].

11.Note that the variable Class is a logic variable that may refer to not only one, but multiple program elements as well. The subsection *Structural Join Point Matching Pointcuts* in section 2.5.1 provides a detailed discussion about the selector construct of Compose*.

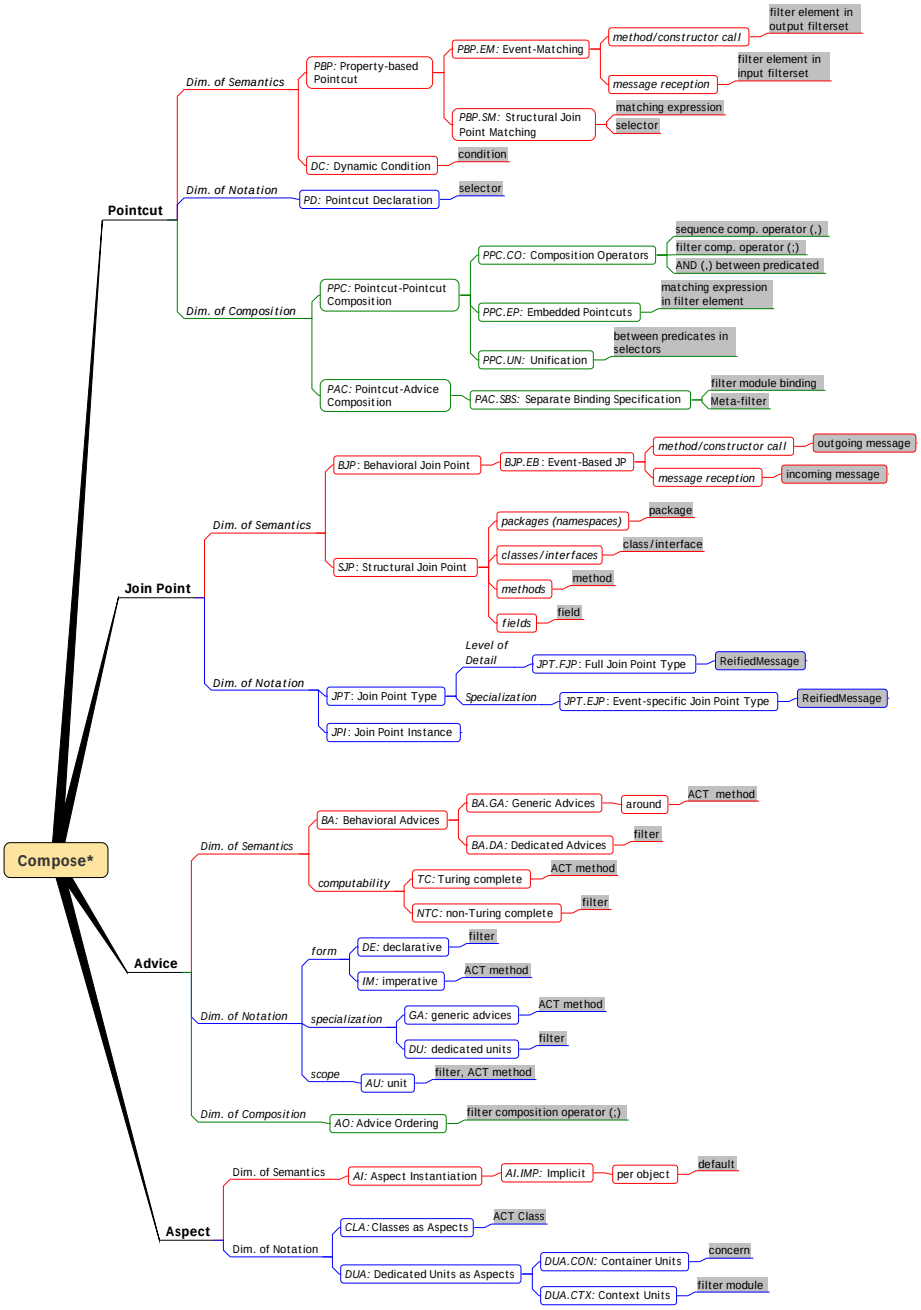

Figure 2.10 *Mapping the language Compose* to the reference model*

2.10 Discussion

It is important to note that we do not consider this model as a final reference model of aspect-oriented languages. It is more like a snapshot of the state of the art aspect-oriented languages. This model will evolve further as the world of these languages is not closed and continuously evolves. We expect that the evolution of these languages can be expressed by extending (e.g. specializing) the design dimensions of our model rather than replacing them.

In general, the principles of aspect-oriented programming can be applied to languages with different programming paradigms, such as object-oriented, procedural and logic programming. In this work, we focus only on aspect-oriented languages that are extending object-oriented languages, such as Java or C#. On the other hand, parts of this reference are applicable to aspect-oriented languages founded on other paradigms. For instance, the join point of a method call in an object-oriented language and a procedural language will likely have similar (e.g. name of the method) and different properties (e.g. only the object-oriented language will have a reference to the target object).

In this chapter, we have discussed various language concepts in the domain of aspect-oriented programming. However, we were mainly focusing on the concepts that are related to aspect-oriented *languages*. Many AOP concepts remained still undiscussed in this study. These concepts, e.g. *wrappers*, *join point instrumentation*, are related to the *execution model* of aspect-oriented languages. These concepts are discussed in [12] that contains a set of surveys on the execution models of aspect-oriented languages. We believe that a similar study to ours can be done to explore the design dimensions of the current execution models, as well as the relationships between the concepts of the language and execution models.

A construct of a concrete language may fulfil the role of more than one language concepts of the reference model. On the other hands, one language concept of the reference model may by realized jointly by more constructs of a concrete language. That is, there might be cases when there is no one-to-one mapping between the constructs of a language and the reference model.

We also observed that there are significant relationships between the discussed language concepts of the reference model. These relationships can be either

conceptual (mandatory) or *alternative*[12], forced by a choice of an alternative of a given language concept. For example, there is a conceptual dependency between the join point and pointcut concepts: join points are designated by pointcuts. This means that there should be a corresponding pointcut designator for each type of join point, otherwise the join point cannot be designated and used. Similarly, the type of a join point determines the type of the advices that can be executed on that join point. In practice, this means that when one develops a join point model, her or she should be aware of the possible type of crosscutting behavior that she intends to perform at a particular type of join point. An alternative relationship occurs, for example, when the designer decides to use implicit aspect-instantiation in her aspect-oriented language. A consequence is that she *may* need to have a construct that supports the parametrization of aspects, which is provided by default in case of using explicit aspect instantiation. However, in this relationship, the first language concept (implicit aspect instantiation) does not require the presence of the second language concept to its work, as opposed to the pointcut-join point relationship, where the join point cannot be used without the presence of the pointcut designator. We believe that an extensive study of the relationships between the language concepts would contribute positively to the current reference model.

2.11 Related Work

[29] investigates the composition concepts of object-oriented languages, AspectJ and HyperJ. It has a different analysis model as compared to our approach: the main dimension of the analysis model is *composition* and the subdimensions are, among others, *composition scope*, *identification* and *deployment*, for instance.

In [26], the authors propose a classification scheme, called *Generic Model for AOP (GEMA)*, as a set of essential and optional features of the current AOP systems. We consider the identified features of GEMA important; however our reference model provides a broader view on the language concepts of the existing aspect-oriented languages. Note that the goals of [26] and our work were also different: the intention of the authors of [26] was to provide a classification scheme of AOP systems, while we focused on exploring the design space of the concepts of the state-of-the-art aspect-oriented languages.

12.We use the terminology of Feature-Oriented Domain Analysis [21] here.

[12] presents a collection of surveys of a set of aspect-oriented languages, and their execution model based on a pre-defined analysis model. The pre-defined analysis model consists of a set of questions, such as *"What are the possible join points? Static? Dynamic?, What are the possible pointcuts?"*, etc. related to the main concepts of AOP languages. Each language in the survey is analyzed in terms of this pre-defined analysis model. As a result, [12] consists of a set of individual surveys of various AOP languages. We built-up our reference model partially based on the language surveys of this document, as well as by analyzing other languages that were not part of [12]. As compared to the analysis model of [12], we chose a wider and more generic set of dimensions. Along these dimensions, we identified and refined possible design alternatives based on the key properties of the language concepts of the state-of-the-art aspect-oriented languages. As a result, our analysis depicts a generic reference model of AOP languages: it presents the common, as well as the distinctive properties of the design alternatives of the AOP language concepts.

2.12 References

[1] AKSIT, M., Ed. *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (Mar. 2003), ACM Press.

[2] AKSIT, M., BERGMANS, L., AND VURAL, S. An object-oriented language-database integration model: The composition-filters approach. In *Proc. 7th European Conf. Object-Oriented Programming* (1992), O. L. Madsen, Ed., Springer-Verlag Lecture Notes in Computer Science, pp. 372–395.

[3] AKSIT, M., AND TRIPATHI, A. Data abstraction mechanisms in SINA/ST. In *3rd Conf. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1988), ACM, pp. 267–275.

[4] AKSIT, M., WAKITA, K., BOSCH, J., BERGMANS, L., AND YONEZAWA, A. Abstracting object-interactions using composition-filters. In *Object-Based Distributed Processing*, R. Guerraoui, O. Nierstrasz, and M. Riveill, Eds. Springer-Verlag Lecture Notes in Computer Science, 1993, pp. 152–184.

[5] AL, N. M., AND RASHID, A. A state-based join point model for aop. In *VAR 2005: Workshop on Views, Aspects and Roles* (2005).

[6] BERGMANS, L. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.

[7] BERGMANS, L., AND AKSIT, M. Composing synchronisation and real-time constraints. *Journal of Parallel and Distributed Computing 36* (1996), 32–52.

[8] BERGMANS, L., AND AKSIT, M. Principles and design rationale of composition filters. In *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, Eds. Addison-Wesley, Boston, 2005, pp. 63–95.

[9] BOBROW, D. G., DEMICHIEL, L. G., GABRIEL, R. P., KEENE, S. E., KICZALES, G., AND MOON, D. A. Common lisp object system specification. *SIGPLAN Not. 23*, SI (1988), 1–142.

[10] BONÉR, J. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)* (Mar. 2004), K. Lieberherr, Ed., ACM Press, pp. 5–6.

[11] BOUCKÉ, N., AND HOLVOET, T. State-based join-points: Motivation and requirements. In *Dynamic Aspects Workshop* (Mar. 2005), R. E. Filman, M. Haupt, and R. Hirschfeld, Eds., pp. 1–4.

[12] BRICHAU, J., AND HAUPT, M. Survey of aspect-oriented languages and execution models. Tech. Rep. AOSD-Europe-VUB-01, AOSD-Europe, May 2005.

[13] COLYER, A. AspectJ. In *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Akc sit, Eds. Addison-Wesley, Boston, 2005, pp. 123–143.

[14] CONSTANTINIDES, C., BADER, A., AND ELRAD, T. An aspect-oriented design framework for concurrent systems. In Lopes et al. [25].

[15] DALAGER, C., JORSAL, S., AND SORT, E. Aspect oriented programming in JBoss 4. Master's thesis, IT University of Copenhagen, Feb. 2004.

[16] DURR, P., STAIJEN, T., BERGMANS, L., AND AKSIT, M. Reasoning about semantic conflicts between aspects. In *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software* (2005).

[17] FILMAN, R. E., AND FRIEDMAN, D. P. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Akc sit, Eds. Addison-Wesley, Boston, 2005, pp. 21–35.

[18] FRADET, P., AND SÜDHOLT, M. An aspect language for robust programming. In Lopes et al. [25].

[19] FRAINE, B. D., VANDERPERREN, W., SUVEE, D., AND BRICHAU, J. Jumping aspects revisited. In *DAW: Dynamic Aspects Workshop* (Mar. 2005), R. Filman, M. Haupt, and R. Hirschfeld, Eds.

[20] HANENBERG, S., AND UNLAND, R. Parametric introductions. In Aksit [1], pp. 80–89.

[21] KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. S. Feature oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, CMU, Nov. 1990.

[22] KAWAUCHI, K., AND MASUHARA, H. Dataflow pointcut for integrity concerns. In *AOSDSEC: AOSD Technology for Application-Level Security* (Mar. 2004), B. De Win, V. Shah, W. Joosen, and R. Bodkin, Eds.

[23] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *Proc. ECOOP 2001, LNCS 2072* (Berlin, June 2001), J. L. Knudsen, Ed., Springer-Verlag, pp. 327–353.

[24] LOPES, C. V. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[25] LOPES, C. V., BLACK, A., KENDALL, L., AND BERGMANS, L., Eds. *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)* (June 1999).

[26] MEHNER, K., AND RASHID, A. Towards a generic model for aop (GEMA). Tech. Rep. CSEG/1/03, Computing Department, Lancaster University, UK, 2003.

[27] ORLEANS, D., AND LIEBERHERR, K. DJ: Dynamic adaptive programming in Java. In *Metalevel Architectures and Separation of Crosscutting Concerns 3rd Int'l Conf. (Reflection 2001), LNCS 2192* (Sept. 2001), A. Yonezawa and S. Matsuoka, Eds., Springer-Verlag, pp. 73–80.

[28] ORLEANS, D., AND LIEBERHERR, K. J. DemeterJ. Tech. rep., Northeastern University, 2001.

[29] OSTERMANN, K. Towards a composition taxonomy. Tech. Rep. CT SE 2, Siemens AG, 2001.

[30] OSTERMANN, K., AND MEZINI, M. Conquering aspects with Caesar. In Aksit [1], pp. 90–99.

[31] RAJAN, H., AND SULLIVAN, K. Aspect language features for concern coverage profiling. In *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)* (Mar. 2005), P. Tarr, Ed., ACM Press, pp. 181–191.

[32] SPINCZYK, O., GAL, A., AND SCHRÖDER-PREIKSCHAT, W. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific* (2002), Australian Computer Society, Inc., pp. 53–60.

[33] SUVÉE, D., AND VANDERPERREN, W. JAsCo: An aspect-oriented approach tailored for component based software development. In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (Mar. 2003), M. Aksit, Ed., ACM Press.

[34] TARR, P., AND OSSHER, H. Hyper/J user and installation manual. Tech. rep., IBM T. J. Watson Research Center, 2000.

[35] URBAN, M., AND SPINCZYK, O. *AspectC++ Language Reference*. pure-systems GmbH, May 2005.

[36] VANDERPERREN, W., SUVÉE, D., CIBRÁN, M. A., AND FRAINE, B. D. Stateful aspects in JAsCo. In *Software Composition* (2005), T. Gschwind, U. Aßmann, and O. Nierstrasz, Eds., vol. 3628 of *Lecture Notes in Computer Science*, Springer, pp. 167–181.

[37] WEGNER, P. Dimensions of object-based language design. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1987), ACM Press, pp. 168–182.

[38] XEROX CORPORATION. The AspectJ programming guide.

Chapter 3

Utilizing Design Information for Evolvable Pointcuts

Traditionally, in aspect-oriented languages, pointcut designators select join points of a program based on lexical information such as explicit names of program elements. However, this reduces the adaptability of software, since it involves too much information that is hard-coded, and often implementation specific. We claim that this problem can be reduced by referring to program units through their design intentions. Design intention is represented by annotated design information, which describes for example the behavior of a program element or its intended meaning. In this paper, we analyze four techniques that are regularly used in state-of-the-art object-oriented languages in associating design information with program elements. Also, the usage of design information in the weaving process of aspect-oriented languages is illustrated and their deficiencies are outlined. Accordingly, we formulate requirements for the proper application of design information in aspect-oriented programming. We discuss how to use design information for the superimposition of aspects, and how to apply superimposition to bind design information to program elements. To achieve this, we propose language abstractions that support semantic composition: the ability to compose aspects with the elements of the base program that incorporate certain design information. We demonstrate the application of design information to improve the reusability of aspects.[1]

3.1 Introduction and Motivation

The process of software development generally consists of refinement of conceptual knowledge towards an executable program. During this process,

1. This chapter is based on work published in [22] and [23].

"ideas" or design artifacts are mapped onto implementation artifacts. Typically, the actual implementation contains the artifacts that are necessary for *execution*. Consequently, certain conceptual knowledge that expresses the intentions of a design is not explicitly represented in the final program. In this chapter, we analyse the impact of this information-loss with respect to the pointcut expression.

In aspect-oriented languages, a pointcut designator expression specifies a composition interface where the behaviour of a (sub)program can be modified or enhanced by composing with one or more *advices* that represents the behaviour of a crosscutting concern. Although *design information*[2] is not necessary for correct execution, it is generally required to avoid fragility of pointcuts with respect to changes in the implementation. The lack of explicit design information in the implementation forces programmers to express the design information in other ways, for example based on syntactic conventions. In this chapter, we argue that specifying pointcuts by designating the syntactic properties of artifacts (and perhaps the state) of the program only, can be too restrictive for evolving programs. For this purpose we present a linguistic mechanism that can be used to express design properties in the pointcut specifications. We also evaluate this mechanism with respect to the recent proposals along this direction.

This chapter is structured as follows: section 3.2 presents an analysis of the various ways programmers use to encode design information in a program. Section 3.3 proposes a language construct for attaching design information to the desired places in programs and for referring to this design information in pointcut specifications. In section 3.4 we present an implementation and application of the proposed language construct in the aspect-oriented language Compose*. Section 3.5 provides details about the implementation. Section 3.6 discusses the related work. Section 3.7 discusses the consequences of the techniques we propose in this chapter. Section 3.8 provides an assessment of our approach in terms of software quality factors, such as comprehensibility, predictability, adaptability, evolvability and modularity. Finally, section 3.9 concludes the chapter.

2. In this work, we consider a piece of design information as a sort of property, since it describes the intentional meaning (i.e. design intention) of a program unit.

3.2 Problem Analysis

Programmers use various techniques to express design intentions in the form of design information attached to certain program elements. In this section, we present and analyse four commonly used techniques for representing design information in state-of-the-art object-oriented languages, such as Java or C#. We also illustrate how AOP languages (may) utilize design information in the pointcut expressions. In the analysis, we show that some of these techniques result in fragile programs especially with respect to evolving requirements. The analysis concludes with a set of requirements for using design information in aspect-oriented languages

3.2.1 Naming Patterns

Technique: A common method for expressing design information is to use naming conventions or stylistic naming patterns [24] in the identifiers of a program. A typical example in Java is illustrated by the following code:

```
1) public class Customer {
2)    private String firstName;
3)    private String lastName;
4)    private String email;
5)    …
6)    public String getFirstName(){ return firstName;}
7)    public void setFirstName(String fname) {
8)       firstName=fname;
9)    }
10)   …
11)   public String getEmail() { return email; }
12)   public void setEmail(String nemail) {  email=nemail; }
13)   …
14)}
```

Listing 3.1 *An example of naming patterns*

This example shows a very simple convention in Java: a method that queries a given instance variable starts with the ’get’ prefix, while the updater method has the ’set’ prefix. There are other well-known naming patterns, such as the add and remove prefixes for maintaining the items of a collection, or the test prefix used by the test fixtures of JUnit [3].

Programmers may use these patterns for the sake of more organized, comprehensible code, but there are frameworks, e.g. JavaBeans [4] or JUnit, which in fact rely on these naming patterns for proper operation. In the latter case naming patterns are not only for expressing design information but they also are explicitly referred to (i.e. they act as 'hooks') in those frameworks. More discussion about naming patterns can be found in [24].

Possible use: The following example shows how naming patterns can be used in combination with wildcards in a pointcut designator expression[3]:

```
1) pointcut queryMethods():
2)    within (Customer) && execution (public * get*());
3) pointcut updateMethods():
4)    within (Customer) && execution (public void set*(..));
```

Listing 3.2 *An example of combining pointcuts with naming patterns*

The example in Listing 3.2 shows two pointcuts. The first one designates the execution of each method that starts with the prefix ’get’ within the Customer class. The second pointcut does the same thing with the ’set’ prefix. The intention of the first pointcut is to designate the execution of methods that query the state of a Customer instance, while the second one designates the execution of 'update' methods. Note that both pointcuts rely on the disciplined application of the naming patterns.

Discussion: In this example, certain properties are hard-wired into the signatures of the base code and the weaving is done based on these signatures. However, programmers need to keep in mind the coding conventions: (a) using the ’set’ prefix to denote the behavior of the method for all *setter* methods, and (b) avoiding incidental naming ambiguities, such as settle() and settings() in this case. This phenomenon has been also identified as the ’arranged pattern problem’ by Gybels et. al. in [15].

The problem stems from the fact that the design information is not separated, but encoded in the structure - more precisely, in the identifiers - of the program. We claim that instead of encoding it in the program, design information should

3. We use AspectJ[19, 11] notation because of its wide use in practice.

be explicitly associated with program units via dedicated language constructs. We will refer to this requirement as *separability of properties*.

The technique of naming patterns has another deficiency which is caused by the fact that it is possible that a unit has to participate in more than one pattern. As an example, consider the member variable that stores the email address in Listing 3.1. Assume that two frameworks, independently from each other, have to be coupled with this variable. The first framework deals with persistence in the whole system, that is, it stores the email address in a database. The second framework is used for encrypting textual data; in this case, it encrypts the email address. To express these dependencies, we might use a naming pattern with an identifier, such as private String persistent_encrypted_email. However, this syntactic solution has several problems. The more patterns are applied, the more juxtaposition of textual identifiers is required in the signatures. In addition, the programmers of a framework need to be aware of the fact that more properties may appear in a signature, not only those properties that are specific to their own framework.

As the example shows, it is an important capability to handle not only single but also multiple pieces of design information attached to the same program unit. We will refer to this requirement as *multiple properties*.

3.2.2 Structural Patterns

Technique: When using structural patterns, the software engineer adapts the structure of the program, without affecting its behavior, for the sole purpose of attaching design information. For example, a frequently applied technique in Java is to use *marker interfaces* [24]. A marker interface is an interface declaration that does not contain any signature declarations. Consider the following example:

```
1)   public interface PersistentRoot {}
2)
3)   public class Customer implements PersistentRoot{ … }
```

Listing 3.3 *An example of a marker interface*

In this example, an (empty) marker interface PersistentRoot is declared, the Customer class then implements this interface. This does not change the behavior of the Customer class, but only 'marks' the class as being a 'PersistentRoot'.

Typical examples of marker interfaces in Java are the java.io.Serializable, java.lang.Cloneable and java.util.EventListener interfaces.

Possible use: Marker interfaces in AOP languages can be designated using the pointcuts this or target:

```
1) pointcut queryMethods():
2)    this(PersistentRoot) && execution (public * get*());
```

Listing 3.4 *An example of combining pointcuts and marker interfaces*

Discussion: Other examples of structural patterns that can be used to attach a certain meaning to an element of a program, are dummy methods (i.e. methods that are not intended to be called), dummy variables and dummy arguments with specific types to indicate a meaning. Such patterns can be used by pointcut designators to identify certain join points within a program.

A difference between structural patterns and naming patterns is that in the latter, the semantic properties are hard-wired purely into the identifiers. This makes them very difficult to maintain, e.g. adding multiple properties is problematic. This does not necessarily apply to all structural patterns, for example a class can implement more than one interface, so it is possible to attach multiple properties via marker interfaces. One problem with marker interfaces is that they can be applied only to classes.

A general problem of both structural and naming patterns is that they statically attach design information to classes. This implies that the information will be applied in every application that (re-)uses these classes, whereas this might not be desirable: for example, PersistentRoot is a property that tends to be specific to an application, not to the characteristics of the class. (That is, a class is not necessarily persistent in every application.) It is also possible that a specific property can be used by different frameworks by coincidence, and they interpret the property in different ways. To solve this problem, we think that design information should be dynamically attachable to units and be configurable according to the needs of different applications. We will refer to this requirement as *late binding*.

3.2.3 Annotations

Technique: The .NET platform (supporting various languages) has annotations (also called *custom attributes*) [5] to bind design information to a range of language constructs. The metadata facility of Java 1.5 [6] realizes the same technique. Annotations are defined as first class entities, they can have arguments and various constraints can be applied on them. The following example shows a definition of an annotation in Java:

```
1)    @Target(TYPE);
2)    public @interface PersistentRoot{
3)       public String tableName() default "unassigned";  }
4)
```

Listing 3.5 *An annotation definition in Java*

@Target is a meta-annotation that constrains the type of the unit to which the newly defined annotation can be attached. The value of the argument is TYPE, which means that PersistentRoot can be attached only to classes and interfaces.

```
1) @PersistentRoot("CUSTOMERS") (1)
2) public class Customer {
3)    @Persistent()             (2)
4)    String firstName;
5)    …
6)    @Persistent()             (2)
7)    String email;
8)
9)    @Query()                  (3)
10)   public String getFirstName(){ return firstName; }
11)
12)   @Update()                 (4)
13)   public void setFirstName(String fname) {
14)      firstName=fname;
15)   }
16)   …
17)   @Update()                 (4)
18)   public void setEmail(String nemail) { email=nemail; }
19)   …
20)}
```

Listing 3.6 *Annotations as design information*

The definition of PersistentRoot has one String argument called tableName. Since a default value is provided, it is not necessary to fill in the argument when the annotation is used.

Listing 3.6 illustrates how annotations can be applied to attach design information to class Customer, which was presented in Listing 3.1. The annotation @PersistentRoot (1) is attached to class Customer to indicate that the instances of this class should be persistent. The annotation @Persistent (2) denotes that the fields email and firstName of class Customer should be stored as a persistent variable. The annotation @Query (3) is attached to those methods that do not change the state of an instance of class Customer, while the annotation @Update (4) is attached to those methods that cause state change.

Possible use: The idea of defining pointcuts based on annotations is not new; however, among the current AOP technologies only JBoss [13], AspectWerkz [10], AspectJ [11] and JasCo [25] support annotations as reference points for designating join points. For example, the execution of the methods that change the state of Customer could be designated by the following pointcut in JBoss:

```
1)  <bind pointcut="execution(Customer->@Update(..))">
2)     <interceptor class=… />
3)  </bind>
```

Listing 3.7 *Using annotations in a pointcut designator of JBoss*

Discussion: The first problem is a that, like naming patterns and marker interfaces, annotations are also statically bound to the units that they are attached[4] to.

The second problem with annotations is that they are usually scattered. In other words, it is possible that an annotation is attached to multiple units over the whole application. For example, it is quite common that the annotation @Author("X.Y.") is attached to every class within one or more packages. Meta-annotations (i.e. annotations attached to the definition of other annotations), such as the annotation @Retention, are often scattered too. The attachment of scattered annotations manually is error-prone and should be automatically

4. If the retention policy of an annotation is SOURCE in Java it is discarded by the compiler and not recorded in the bytecode.

generated, whenever it is possible. We will refer to this requirement as *support for scattered properties*.

3.2.4 Automatically-derived Semantic Information

Technique: As the field of aspect-oriented programming evolves, the need for more 'expressive' pointcuts becomes apparent. This is illustrated by Kiczales' keynote in [18]; he argued that the ways to express pointcuts should be as close as possible to the intention of the designer. This naturally leads to proposals for expressing pointcuts that do not directly refer to elements within the program (source), but that refer to those points in the program or the execution of the program that fulfil a certain property. These join points can only be determined by reasoning about the semantics of the program and the adopted programming language.

Possible use: A well-know example is the primitive pointcut in AspectJ named cflow(). It selects all the points in the execution of a program that occur between the entry and exit of one or more join points provided as its argument. Clearly, this does not refer to the syntax and structure of the program itself. This means that the patterns in the execution of the program can only be identified by taking into account the semantics of the programming language. Other examples of semantic language patterns are discussed in [15]; here, advanced pointcut expressions can be defined, which can reason about (the execution of) the program to determine the join points.

All these techniques (i.e. cflow, pointcuts of [15]) have in common that the pointcut expression is not just referring to the names and structure of the program, but can only be resolved by reasoning about the semantics of the language under consideration.

Discussion: Automatically-derived semantic information is used to capture the intention of the designer by analysing the semantic patterns of a program (execution). The two inputs to this analysis process are the program itself, and the semantics/meaning of the programming language involved. However, not all relevant semantic information/ intentions can be derived from these sources: certain semantics are defined by the domain and normally not encoded into the program (as it is not required for the execution itself). We have previously mentioned the example of the persistence of individual program

elements; it depends solely on the particular requirements of the application; it does not affect the behavior of the core functionality itself, and may apply to both classes and the instance variables individually.

Since certain design information cannot be algorithmically derived, we should be able to attach domain specific design information as well. We will refer to this requirement as *dealing with domain specific semantics.*

Note that naming and structural patterns are not suitable for semantic reasoning; however, they can enrich programs with domain specific semantic properties.

3.2.5 Summary

In the previous sections, we have presented different ways how design information can be bound to, or derived from, different units of an application. Table 3.8 summarizes how the presented techniques support the identified requirements:

Table 3.8 *Techniques vs. requirements*

	Separability of Properties	*Multiple Properties*	*Scattered Properties*	*Late Binding*	*Domain Specific Properties*
Naming Patterns	no	no	no	no	yes
Structural Patterns	no	yes	no	no	yes
Annotations	yes	yes	no	no	yes
Derived Properties	yes	yes	yes	yes	no

The columns of this table represents the issues (i.e. requirements) that we identified in the previous sections, the rows represents the analysed techniques. *Separability of properties* requires that instead of encoding in the program, design information should be distinctly connected to program units via dedicated language constructs. This cannot be achieved by naming and structural patterns as they encode the design information either in the identifier or the structure of the program. Annotations are dedicated abstractions to attach design information to various program elements. Derivation techniques can also express design information in terms of abstractions that are independent from the program. *Multiple properties* (in the second column of Table 3.8)

ensures that multiple design information can be attached to a certain program element. Almost every technique satisfy this requirement to certain extent; however, the solution provided by naming patterns had several problems, as we discussed in 3.2.1. *Scattered properties* (in the third column of Table 3.8) requires that a single design information that is applicable to several program elements of an application should not be scattered over in the source of the entire application. *Late binding* requires that design information should be dynamically attachable to units and be configurable according to the needs of different application contexts. Both of these latter two requirements were supported by only the derivation techniques[5]. *Domain specific properties* ensures that domain specific design information can be assigned to the program elements of an application. Naturally, this is possible with every technique except the derivation (semantic reasoning) technique, as the domain specific design information cannot be derived from the structure of the application, in that case.

According to the above analysis, the combination of annotations and semantic reasoning seem to be the ideal solution to represent design information in AOP languages. However, there are certain problems that need to be addressed to make these techniques to work:

1. AOP languages need to support explicit pointcut designators that can refer to annotations[6].
2. There must be means by which scattered annotations can be superimposed.
3. There must be means by which the place of annotations can be derived based on certain rules (e.g. the derivation of annotations can be driven by semantic reasoning).
4. There must be means to ensure the decoupling of design information from the base code; i.e. they must not be always statically bound to the program.

5. Since the first version of this work, the pointcut language of aspect-oriented languages has further evolved. At time of writing, a few aspect-oriented languages supports some of these requirements already.
6. The need for expressing 'semantic pointcuts' was also identified in [21].

3.3 General Approach

In the previous section, we discussed the requirements and the problems to be addressed to support the use of design information in aspect-oriented programming. In this section, we will explain our general approach towards those problems. Figure 3.1 (b) offers an overview of our approach, as compared to the state-of-the-art aspect-oriented approaches shown in Figure 3.1 (a). In Figure 3.1 (a), the white-coloured shapes show the essential concepts of aspect-oriented programming, that are relevant to this chapter. The large box at the top represents the logical program elements of the (base) program. Program elements can be selected by pointcut specifications. This selection is typically based on the lexical and structural properties of the program elements or on the results of a more advanced program analysis techniques (for example, control flow analysis). The picture illustrates that superimposition is defined based on the specifications of advices and the corresponding pointcut specifications.

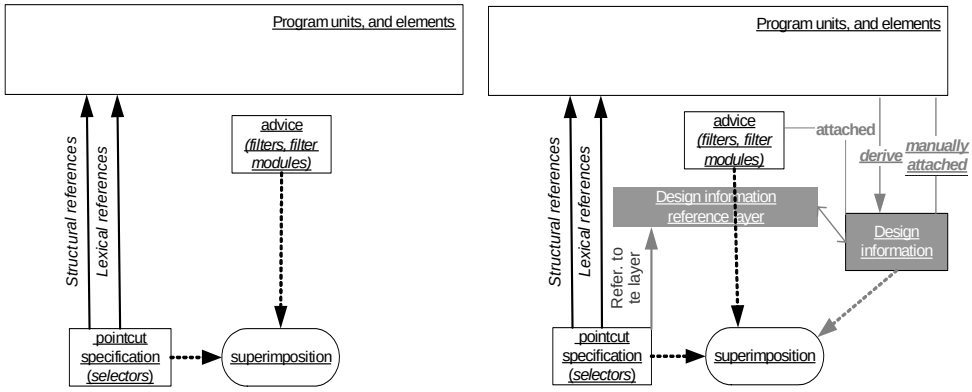

Figure 3.1 *(a) a traditional AOP approach; (b) aspect composition through semantic reference layer*

In Figure 3.1 (b), the grey parts illustrate the additional elements and relations that we propose in this chapter. The key element of our approach is that, instead of referring directly to the program, the specification of pointcuts incorporates references to design information. We aim to select program elements based on the annotated design information. This is achieved by the *design information reference layer*. We use the term *semantic composition* when the elements of a composition have been 'collected' by referring to design information. There are

several ways in which design information (cf. semantic property) may be associated with program elements:

- Attach it manually with annotations
- Derive it based on the existence of other design information.
- Attach it through superimposition of possibly crosscutting annotations (as indicated by the grey dashed arrow between design information and superimposition in Figure 3.1)

Finally, the figure also illustrates that the connection between superimposition and advice (i.e. the selection of advice), can be made based on its associated design information.

The key benefit of our approach is that it reduces direct dependencies between the crosscutting concern and the program source. This is realized by introducing a separate abstraction layer in between, which aims at describing the join point through specific design information. We consider this approach more resilient to changes in the program and/or requirements because join points can now be designated based on the associated design information instead of lexical and structural information.

3.3.1 Pointcuts with design information

To incorporate design information in the pointcut designation process, we present two design alternatives here although there might be other possible solution as well.

As we have shown before, annotations (i.e. custom attributes in .NET) are considered as extra properties on the signature of a unit. Thus, one simple alternative is to extend the type patterns in the pointcut designators with annota-

tions. For instance, the execution of the methods that change the state of a persistent class is designated by the following pointcut in AspectJ:

```
1) /* Alternative 1. */
2) pointcut updateMethods():
3)    within(@PersistentRoot *) && execution(@Update * *(..));
4)
5) /* Alternative 2. */
6) pointcut foo(Customer c):
7)    target(c) && hasAttribute(c, @PersistentRoot) && …
```

Listing 3.9 *Alternatives of pointcut referring to annotations*

Another possible alternative (line 6-7) could be to use dedicated predicates for custom attributes, such as the predicate hasAttribute(). This is illustrated by the second alternative of Listing 3.9.

However, this latter approach is suitable for a unit that can be explicitly referenced in other pointcuts only. For example, in AspectJ, a class can be explicitly referenced in the context matching target() and this() pointcuts, or a parameter in the 'args pointcut', while a method as a unit can only be explicitly referenced through its call or execution. In short, the existing pointcut mechanism of a language may also influence the way how annotations are referenced in the pointcuts.

3.3.2 Superimposition

In this subsection, we discuss the meta-model and the designating language that plays an essential role in the superimposition of annotations.

Meta-model

To superimpose annotations, it is necessary to know what type of units can have annotations. The set of possible types varies from language to language, although there is a small set of units that is common to every object-oriented language, such as *class*, *method*, *parameter* and *member variable* (field). For example, according to the JSR-175 specification [6], we can attach annotations to the following units in Java: types, fields, methods, parameters, constructors, local variables, annotations, and packages. Note that types of units that are specific to an aspect-oriented language, such as the *aspect* and *advice* in

AspectJ, or *concern* and *filtermodule* in Compose* could have annotations as well, but this is of course not supported by the standard of the base language.

However, the type of the unit is only one property that can be referred to designate a unit. There are also other important properties that can be used for designation, such as the visibility, modifier or name of a unit.

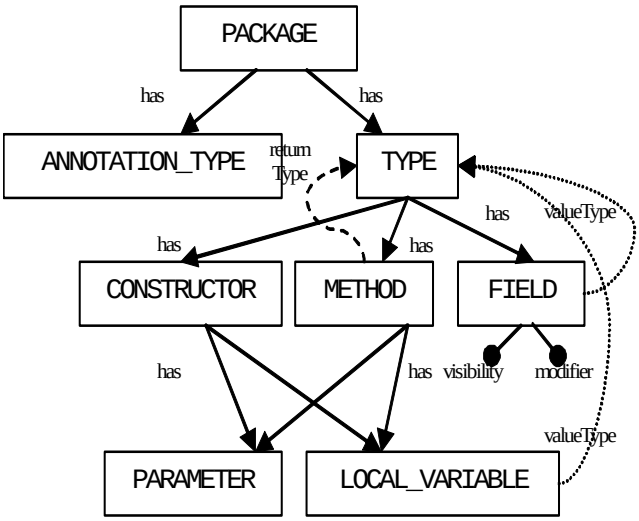

Figure 3.2 *A part of the meta-model*

The superimposition of annotations cannot only be based on the properties of units but also the relationships between units. For instance, one important relationship is the aggregation relationship between various units and program elements. This relationship is used, for example, when an annotation should be attached to "*every field within a given class*". Figure 3.2 illustrates a hierarchy of the program elements and units based on the aggregation relationship (labelled by the word *has*).

There are other important relationships and properties as well. For example, in Figure 3.2 we indicated two additional types of relationships. The valueType relationship indicates the type of a field or a local variable (also applicable to the parameter node). The returnType relationship from the node Type to Method shows the type of the return value of a method. The full meta-model used in the superimposition of annotations is discussed in [16].

The possible properties of units and the relationships between them make up the static meta-model of the target hierarchy that can be constructed based on source or byte-code analysis. The meta-model essentially determines the way of querying the units to which annotations are to be attached.

Specification language for pointcuts

In order to designate certain units, it is also necessary to provide a language that can express queries on program units based on the properties and relationships of the meta-model. To express queries on the representation of complex program structures, the language should be capable of handling nested structures of heterogeneous types. Predicate-based query languages (e.g. JQuery [17]) have already been proposed to express such queries. In addition, a predicate-based language can provide features such as unification, recursion and a built-in reasoning mechanism on properties [15] that significantly improve the flexibility of a pointcut language. We also believe that a predicate-based language provides an intuitive use for programmers. For these reasons, we use a predicated-based language to formulate the queries of program elements based on the proposed meta-model.

Listing 3.10 shows an example of how an element of the previously discussed meta-model can be queried in a predicate-based language. In the example, we use Compose* for illustrating such a predicate-based query language. In Compose*, a query of program elements can be formulated in the *selector* language construct. Every selector has a name (for example, guiComponents), which is unique within the concern where it is defined. This selector designates (=) a set that consists of possible values found by the Prolog engine for a variable (GuiType), where the predicates after the '|' puts constraints on these values. In this example, the first predicate (isNamespaceWithName) binds the namespace com.myCompany.gui to the GuiNS variable. The second predicate binds every unit (e.g. interface or class) within the selected namespace to the GuiType variable. Because of the unification mechanism of Prolog, however, only the

types that also satisfy the third predicate (isTypeWithAttribute) will be bound to the GuiType variable.

```
1) superimposition {
2)   selectors
3)     guiComponents =
4)      { GuiType |
5)        isNamespaceWithName(GuiNS, 'com.myCompany.gui'),
6)        namespaceHasType(GuiNS, GuiType),
7)        isTypeWithAttribute(GuiType,'public').
8)      };
9) }
```

Listing 3.10 *A selector of Compose* designating annotated types*

To resolve such a query, the fact base of the Prolog engine is filled up based on the repository of a Compose* project; this repository is essentially an instance of the meta-model that we have discussed previously.

With respect to the meta-model, the predicate language consists of predicates that refer to the properties of units (e.g. isClass(C), isClassWithAttribute(Class, ClassAttribute)) and the relationships between units (e.g. isSuperClass(SuperClass, SubClass), classHasMethod(Class, Method)). Besides, we defined "convenience" predicates to express more complex structures (views) on the meta-model (e.g. inherits(Parent, AnyChild) to check if a class is indirectly a subclass of another class). As a result, the current language consists of a large number of predicates; the complete list of predicates is described in Appendix B. In this chapter, we focus only on those predicates that are relevant for annotating design information. In section 3.4, we show in more detail how this predicate language can be used to carry out a matching process on annotations and how to superimpose (introduce) annotations on program units.

Binding Annotations to Program Elements

We distinguish various alternatives to modularize the specification of binding of annotations to program units. In the following, we have distinguished three options:

Manual (source-code) binding. The annotations are embedded and attached one by one to the selected units in the base code. In this way, as we discussed in the Problem Analysis section, annotations are statically bound; it can be

difficult to reuse the units with annotations if the same annotations are not valid in each application. On the other hand, it is important to note that in case of certain domains (e.g. security, synchronization) programmers may intentionally want to bind annotations statically to ensure certain constraints in different applications. In this case, every application that reuses a program unit with annotations will have the same set of annotations. The manual binding approach is not suitable for scattered annotations.

Binding via a shadow file. The binding between an annotation and a programming unit is located in one or more separate, shadow files (e.g. an XML file). By using this approach, annotations are not bound statically; an application can have its own set of shadow files that contains the application specific binding. (This approach can also be applied in JBoss.) To realize this, there must be a framework that can manage the shadow files in each program. Using this approach, scattered annotations can be handled, too. However, this framework must be able to deal with the superimposition specification of annotations.

Binding via an aspect. The binding is formulated in a superimposition specification that is placed in a module representing an aspect. With the help of this approach, we can avoid annotations that are statically bound. Naturally, the problem of scattered properties is solved as well. However, every program and other tools, such as integrated development environments (IDEs) must be able to interpret the superimposition specification when they need the information if an annotation is attached to a certain unit. This problem can be solved by an annotation compiler that resolves the superimposition specification and attaches the annotations directly to the corresponding units (e.g. manipulates the bytecode of the unit). Thus, the annotations will be statically bound as in the first case. Note that this approach can be combined with the second alternative: the annotation compiler generates the XML file that represents the binding of annotations as meta-data. Note that the shadow file (of the second option) can be regarded a kind of specialized implementation of an aspect. With the shadow file, the weaver is logically seen as part of the framework.

3.4 Extending Compose* with Annotations

In this section, we outline how design information annotations can be adopted in Compose* [9, 12], which is our aspect-oriented language implementation for .NET

3.4.1 Definition of Annotations

Annotations have to be defined before they can be attached to a unit. In .NET annotations are defined as classes that extend the System.Attribute class. The following example shows the definition of the annotation Persistent.

```
1) [AttributeUsage(AttributeTargets.Field)]
2) public class Persistent : Attribute{
3) }
```

Listing 3.11 *The definition of the annotation Persistent*

The annotation AttributeUsage is a meta-annotation (i.e. an annotation bound to the definition of an annotation) defined by the .NET framework and it specifies that the annotation Persistent can be attached to only fields.

3.4.2 Annotation-based selection

To designate join points based on annotations, pointcuts have to be able to refer to annotations. In Compose*, annotations can be referenced in two ways. Listing 3.12 illustrates this by an example. The first alternative is to extend the type patterns with annotations in the set of input filters, as it is shown at (1): [@Update *] means that every method with the annotation Update will be matched by the Meta filter. (Without @Update, all methods would match.) For the Meta filter, when a method matches, it is reified and passed as a parameter to the updateAction() method that is executed on a PersistentManager instance.

The second alternative at (2) shows how logic predicates can be used to formulate queries based on matching annotations. This selector will designate all classes that have the annotation PersistentRoot by querying all units in the system and selecting those that match the applied predicates.

In the filtermodules part of the superimposition, the filtermodule Updating is superimposed on each class that is designated by the selector (c.f. query)

named persistentClasses. Thus, every instance of these classes will have an instance of the filtermodule Updating superimposed.

```
1) concern Persistence{
2)    filtermodule Updating{
3)       externals
4)          pm : PersistenceManager =
5)                PersistenceManager.instance();
6)       inputfilters{
7)         redirect : Meta =
8)            {[@Update *] pm.updateAction};          (1)
9)         dispatch : Dispatch =
10)           {inner.*}
11)      }
12)   }
13)   superimposition{
14)      selectors
15)       persistentClasses =
16)        {PersClass | isClass(PersClass),           (2)
17)           classHasAnnotationWithName(PersClass,
18)           'PersistentRoot')
19)        };
20)      filtermodules
21)       persistentClasses <- Updating;
22)   }
23)   implementation in Java; …
24)}
```

Listing 3.12 *Definition of the concern Persistence (cf. aspect)*

In this way, annotations are referenced two times. First, the intercepted methods are filtered based on the annotations in the filtermodule specification. Secondly, annotations are also referred to by predicates to superimpose the filtermodules.

3.4.3 Superimposition of Annotations

In the previous section we have shown how annotations can be referenced in selectors and type patterns within filters (i.e. they are the "pointcuts" of

Compose*). In this section we illustrate how annotations can be superimposed through selectors. A simple example is shown in Listing 3.13.

```
1) concern AssignAuthor{
2)    superimposition{
3)      selectors
4)        allGUITypes =
5)         { GuiType | isNamespaceWithName(GuiNS,
6)                     'com.myCompany.gui'),
7)                     namespaceHasType(GuiNS, GuiType)};
8)      annotations
9)        allGUITypes <- Author("John Smith");
10)  }
11)}
```

Listing 3.13 *Superimposition of the annotation Author*

The superimposition specification of Compose* has been extended with a new part called annotations. This part gives place to the specification of binding design information to selectors. In our example, we superimpose the annotation Author("John Smith") on allGUITypes at (4). As a result of this, the annotation Author("John Smith") will be attached to every type (classes, interfaces, enumerations, etc.) within the namespace com.myCompany.gui.

3.4.4 Annotations in matching Expression - Defining Adaptable Aspects

Problem Description and Example

We will now introduce an example to illustrate how design information can be utilized in the matching expressions of filters. The source code in Listing 3.14 shows the example concern Caching.For the sake of efficiency, the computation of certain values is cached. CachingModule intercepts calls on a method which does the computation (here, getPerimeter() and getArea()), and instead of performing the method, it returns the previously computed value that is cached (see the Dispatch filter of CachingModule). Also, each call that can change the values used in the computation (i.e. setRadius) is intercepted and the state of the

cache is set to invalid (see the filtermodule MaintainCache in Listing 3.14).

```
1)  concern Caching{
2)    filtermodule CachingModule{
3)      internals
4)         c : CacheACT;
5)      conditions
6)         isInvalidCache : c.isInvalidCache()
7)      methods
8)         getStoredValue : c.getStoredValue();
9)         updateStoredValue : c.updateStoredValue();
10)     inputfilters
11)        /*only for Circle*/
12)        Update : Meta =
13)          { isInvalidCache() =>
14)              [getPerimeter, getArea] updateStoredValue }
15)        /*only for Circle*/
16)        disp : Dispatch =
17)          {!isInvalidCache() =>
18)               [getPerimeter, getArea] getStoredValue }
19)   }
20)
21)   filtermodule MaintainCache{
22)     ...
23)     inputfilters
24)        /* only for Circle! */
25)        change : Meta =
26)          { [setRadius] setInvalidCache }
27)      } …
28)    }
```

Listing 3.14 *Caching with application specific signatures*

When the state of the cache is invalid, the new value is computed and stored in the cache again, and the state of the cache becomes valid (see the Meta filter of CachingModule)

In the example code of Listing 3.14, the concern Caching is superimposed on class Circle which has the following interface:

- methods that perform computation: getPerimeter, getArea
- the method that changes the values used in the computation: setRadius (only the radius is used in the computation methods)

For each method that performs a computation (e.g. getPerimeter, getArea), the return value and a Boolean variable that indicates the state of the cache are stored. This functionality is implemented in the method updateStoredValue (of CacheACT).

Problem: In this example, the signatures of Circle (in bold) are explicitly referred to in the filters; thus, CachingModule cannot be adopted by a new concern with new computation methods (e.g. the method getVolume() of class Sphere). In general, the adaptability of filtermodules is limited because the signatures always have to be explicitly specified in the type patterns of the specification of the filters.

Solution & Example Revisited
By referring to design information, the concern Caching can be defined in a reusable manner so that programmers can customize it to different applications. The following figure presents an adaptable version of the concern Caching.

The original source of Caching has been changed in two places. At (1) the actual signatures of the methods that perform the computation are replaced by the annotation Computation. Thus, these methods are not directly referred through their names but they are referred through the annotation Computation. Similarly, at (2) the actual signatures of the methods that can change the input values of a computation are replaced by the annotation ChangeInput. Hence, these methods are also indirectly referred through the annotation ChangeInput.It is also necessary to attach the above-mentioned annotations to the right methods in the base code. A possible solution is to "manually" embed the annotations in the source of the base classes. Thus, in our example (lines 30-35 in Listing 3.15), the methods getPerimeter and getArea have to be tagged by the annotation Computation, since they perform the computation to be cached.Similarly, the method setRadius has to be labelled by the annotation ChangeInput, since it changes the input values of the computation. Note that embedding annotations in the base code can lead to several problems; instead of this technique, the

superimposition of these annotations might provide a better solution. We discussed this issue in detail in the previous section.

```
1) concern Caching{
2)   filtermodule CachingModule{
3)     internals
4)        c : CacheACT;
5)     conditions
6)        isInvalidCache : c.isInvalidCache()
7)     methods
8)        getStoredValue : c.getStoredValue();
9)     inputfilters
10)       /*for every computation method*/
11)       Update : Meta =
12)          {isInvalidCache() =>
13)  (1)         [@Computation] updateStoredValue }
14)
15)       /*for every computation method*/
16)       disp : Dispatch =
17)  (1)      {!isInvalidCache() =>
18)             [@Computation] getStoredValue }
19)  }
20)
21)  filtermodule MaintainCache{
22)    inputfilters
23)       /* for every method that changes the
24)        * input of the computations */
25)  (2)  change : Meta = { [@ChangeInput] setInvalidCache }
26)   }
27)   …
28)}
29)
30)/* === */
31)
32)class Circle{
33)   [Computation] public double getPerimeter(){ return …; }
34)   [Computation] public double getArea(){ return …; }
35)   [ChangeInput] public void setRadius(double r){  …  }
36)   …
37)}
```

Listing 3.15 *Reusable Caching with Annotations*

Discussion: Using annotations in the filters instead of using explicit signatures allows for defining reusable (adaptable) filtermodules. Thus, if a filtermodule contains only annotations, it will be free of signatures that are specific to an application. To customize the filtermodules to different applications, the annotations referred in the filtermodules need to be attached to the necessary units in the base code. By the combination of mechanisms (using annotations in filtermodules plus superimposing annotations on the base code), methods are indirectly intercepted based on their semantic properties. As result of this, concerns can be adapted in a generic way.

In AspectJ, this technique is equivalent to referring to annotations in pointcuts. Note that the same problem could be handled by abstract pointcuts as well. However, by using design information and generic aspects, the customization of aspects is "automatically" managed, as long as design information is used in a disciplined manner.

3.5 Implementation

Figure C.1 in Appendix C presents the architecture of Compose*, the realization of Composition Filters on .NET platform. Both the annotation matching and introduction mechanisms that we proposed in this section were implemented within the LOgical Language (LOLA) and Superimposition ANalysis Engine (SANE) modules. For further implementation details, we refer to [16].

Appendix B describes the predicates that can be used in the *selector* construct of Compose* presented in this and the following chapter.

3.5.1 Limitations

The matching of annotations in filterelements is currently implemented by a workaround: this is an extra if statement in an ACT to determine if the intercepted method has the proper annotation attached.

In the current implementation, selectors can match only by the type of an annotation; the arguments of an annotation (or the member values of annotation instances) cannot be referred to. This issue is going to be resolved in the subsequent releases of our tool by providing additional predicates to formulate selectors based on that information as well.

3.6 Related Work

3.6.1 On Annotating Design Information

Attaching annotations to certain units of programs and models is not a new idea in software engineering. One of the first appearance of annotations can be found in POOL-I [4]. The specification of a type in POOL-I is augmented with a collection of properties which are merely identifiers. These properties are used by the compiler to determine if one type is a subtype of another one. As the authors say, "Ideally, such a property identifier serves as an abbreviation for a formal specification of some aspect of an object's behavior".

In UML [7], stereotypes can be attached to the elements of a model to express certain design intentions and properties. Tagged values are another means to attach design information in UML.

Before the appearance of semantic annotations in .NET or Java 1.5, programmers could attach semantic information in the form of attributes within Javadoc tags. The Attrib4j [2] and Apache Common Attributes [1] projects realized this technique. However, attributes were still not treated as first class entities, unlike annotations in .NET or Java, in these projects.

3.6.2 On Aspect-Oriented Programming

AspectWerkz [8] is a dynamic aspect-oriented framework for Java that is capable of embedding aspects into the base code through annotations. In other words, there is a set of custom annotations that expresses an AOP language, and one can write his or her aspects by using these custom annotations in the base code. AspectWerkz allows for matching on annotations in pointcuts; however, it does not support introducing annotations (this functionality is mentioned as a future work). Hence, the problem of scattered annotations and late binding is also not addressed.

JBoss AOP [13] is a Java based aspect-oriented framework usable in any programming environment and integrated with the application server of JBoss. Late binding is possible in this framework; besides inserting the annotations into the bytecode of classes (static binding), the annotation compiler can generate a separate XML file that contains the metadata (i.e., the binding of custom attributes). However, the framework allows for matching on annotations in

pointcuts, and introducing annotations only to a limited set of units (classes, methods, constructors and fields) through a more limited pointcut language than ours. (Java 1.5 allows for more types than only these four; e.g., annotations can be attached to packages, annotations, etc. according to the JSR-175 [5] specification.)

JQuery [17] is a flexible, query-based source code browser, developed as an Eclipse plug-in. In JQuery, users can define their own queries to select certain elements of a program. The JQuery query language is defined as a set of TyRuBa [14] predicates which operate on facts generated from Eclipse JDT's abstract syntax tree. The predicates of JQuery are dedicated for Java and the factbase of JQuery is based on Java sources and bytecode files. In Compose* we also use a predicate language to specify selectors as queries. Our predicates are dedicated for Compose* and the factbase is based on the repository model of a Compose* project.

In [8], A. Colyer's proposes to extend the pointcut language of AspectJ to do matching based on annotations. Note that the syntax given in [8] was for illustrative purposes only. Recently, in the latest version of AspectJ, the above mentioned proposal had been implemented as well. In AspectJ, annotations can be introduced through a new language construct, called declare annotation. Using type patterns in this construct allows for introducing annotations over multiple units, i.e. it addresses the problem of scattered properties. However, type patterns have a limited expressiveness to designate units that annotations are attached to, as compared to the predicate language we propose. The difference is that type patterns can only designate units based on the properties of those units, while our predicate language can designate units based on the context of units, as well. Typically, relationships with other units (e.g. inheritance, aggregation) and properties of related units make up the context of a unit.

In [20], R. Laddad investigates the application of metadata in AOP. In this work, he gives practical hints and guidelines how to use and how not to use annotations in combination with AOP, particularly, with AspectJ. In our paper, we also investigated various novel application possibilities of using annotations in AOP, such as providing reusable aspects and evolvable weaving specifications (for instance, by designating advices based on annotations.)

3.7 Discussion

In this section, we discuss the consequences of the techniques we proposed in this chapter, how they can be realized, and finally we summarize the chapter and outline some future work.

3.7.1 Benefits and contribution

The primary contribution of this chapter is that it presents an in-depth analysis of the role of design information in the context of aspect-oriented programming. We showed that there is a need for annotating programs with design information. We also demonstrated that the integration of design information with aspect-oriented composition mechanisms offers a means of coupling that is both manageable and powerful. The main benefits of design information annotations that we introduced in this chapter are:

- The ability to select join points based on design information (which can never be derived from the program itself).
- The programmer can freely choose what the appropriate locations are to define annotations: within the code, co-located with the code or separated in an aspect.
- The usage of design information makes aspects, especially pointcut expressions, less vulnerable to changes of the program. The reason is that they avoid dependencies of the pointcut expression upon the structure or the naming conventions of the program.
- The dependencies between program elements can be more precise and easier to understand by referring to the design intentions instead of syntactic structure or naming.
- Finally, our proposal supports the definition and customization of reusable aspects.

The related work section discussed several recent programming language implementations that offer -to varying degrees- implementation techniques that are necessary to deal with design information. However, almost none of that related work has pointed out the problems of current ways to deal with semantic properties, or explicitly motivated the need for applying these techniques[7]. Only the work of Gybels et. al. [15] explicitly discusses these problems. However, the solution they propose is based on automatic derivation of

properties, which cannot always be realized, as we pointed out in section 3.2.4. A preliminary version of the analysis and the solution presented in this chapter was given in [22]. In the next chapter, we propose several ways to apply annotations in the context of aspect-oriented programming.

3.7.2 Limitations and suggestions

This section lists a number of issues that result from our proposal, along with some suggestions how these can be tackled:

- It is important that software engineers use a consistent and coherent set of design information for each sub domain of an application (whether from a technical/solution domain, or from the application/problem domain). For example, programmers cannot make use of our approach if they use terms such as 'setter', 'writer', or 'updater' in an inconsistent manner.

  In other words, disciplined programming is still required to keep the correct design information associated with the appropriate program elements. However, we believe that through the language abstractions proposed by this chapter, the situation can be improved: we have illustrated how superimposition and derivation can be used to attach semantic properties. In addition, we plan two ways to address this issue: (1) by investigating design-level support and the automatic derivation of annotation specifications from stereotypes in UML diagrams. (2) By searching for techniques that can automatically derive certain common annotations. We believe that the development and use of ontologies for identifying a consistent set of semantic properties in a particular domain can be a useful approach.
- The annotated design information may require parameters for passing context information; if the property is superimposed, it is typically hard to include such context information. One alternative to deal with this is by accessing the context through a generic reflective interface.

7. In fact these appear to be rather technology-driven by the introduction of annotations in Java 5. However the motivation of introducing annotations in java is formulated as: "One of the primary reasons for adding metadata to the Java platform is to enable development and runtime tools to have a common infrastructure and so reduce the effort required for programming and deployment." [6]

3.8 Trade-off Analysis of Software Quality Factors

3.8.1 Comprehensibility

Explicitely adding design information to program elements increases the comprehensibility of the code; programmers can read the intended functionality of a program element. By *selecting* join points based on annotations also increases the comprehenesibility of pointcuts and aspects: the dependencies between program elements can be easier to understand by referring to the design intentions, instead of syntactic structure or naming patterns. (If the latter would exactly express the intention, it would be preferred, though). Note that by using design information the *intention* of the composition becomes clearer for the reader of the source code; however, the concrete dependencies are less visible, as the pointcuts may now refer to (generic) annotations instead of concrete program elements.

Superimposing (introducing) design information to program elements may decrease the comprehensibility of the code. The main problem is that an annotation might be attached to a program element - in the byte-code of application through a superimposition - but this is not visible in the original source code. One possible solution to this problem is to indicate the presence of an annotation in a comment, or through IDE tool support, for example.

3.8.2 Evolvability

The usage of design information makes aspects, especially pointcut expressions, less vulnerable to changes in the source of a program. The reason is that they avoid dependencies of the pointcut expression upon the structure or the naming conventions of the program. Consequently, this has a positive impact on the evolvability of the code as long as the use of annotations is kept consistent.

3.8.3 Predictability

Using explicit design information, in the form of annotations, does not improve the predictability of the source, as compared to naming or structural patterns. To use the design information in a correct manner, programmer should not forget to attach it where they need to do.

On the other hand, this issue can be addressed by, for example, deriving the design information from design documentation, such as UML diagrams. Also, other analysis techniques, such as control-flow and data-flow analysis, can help in determining, and associating certain semantic properties with program elements.

3.8.4 Adaptability

In general, the use of annotations increases the adaptabilty of the components of a system: by providing a technique that allows for the superimposition of annotations, annotations do not need to be statically bound to program elements (i.e. components) anymore. This means that a given component can be reused and adapted to different application contexts by superimposing different annotations on it. This is an important feature in large-scale software development where the reuse of existing components is always a central issue.

3.8.5 Modularity

The use of design information in the form of annotations has a positive impact on the modularity of the system: using annotations in pointcuts introduces a level of indirection in pointcut matching. In other words, a pointcut that refers to lexical or structural elements can be now substituted with a pointcut that refers to design information, in the form of an annotation. In this way, the pointcut is not directly coupled with the base code and the aspects will not have explicit dependencies to concrete program structures.

By superimposing (introducing) annotations, scattered annotations can be expressed within a single statement. This means that an annotation that would otherwise be present in the source of several modules can be localized only in one module.

3.9 Conclusion

Software that is developed today is making frequent use of design information that is encoded in some way into the source code. In this chapter, we argue that it is unavoidable to add such design information (a) since not all relevant design information can be derived from the executable source code, and (b) when one wants to refer to program elements based on design intentions, rather

than trying to capture the right set of elements by referring to their 'accidental' lexical and structural properties.

In section 3.2 we have analyzed four techniques currently used for binding and deriving design information in traditional object-oriented languages, such as Java or C#. We have illustrated how aspect-oriented software often utilizes design information in its pointcut expressions, and we have presented deficiencies when using any of these four techniques.

To conclude the problem analysis, we have formulated three requirements for adopting design information in aspect-oriented programming:

1. Pointcut expressions may need to be able to refer to design information.
2. There is a need to support the superimposition of design information.
3. It must be possible to do late binding of design information to program units and elements.

In section 3.3 we have analyzed how design information can be used in the superimposition of aspects and how superimposition can be applied to bind design information to the base code. Based on this analysis, in section 3.4 we have shown how a concrete aspect-oriented language (Compose*) can be extended to support modeling design information in such a way that the above mentioned requirements are fulfilled:

1. Section 3.4.2 has presented how design information can be used in pointcut expressions to designate join points.
2. Section 3.4.3 has illustrated how design information can be superimposed or derived.
3. The bindings between the design information and units can be expressed by superimposition specifications and localized in a concern, which is the aspectual module of our language (see section 3.4).

A number of research topics for future research have appeared as the result of this work; some of those address the limitations as explained in section 3.7.2. Other potential future work is about (a) the ability to apply the notion of seman-

tic composition to other composition techniques than the superimposition mechanism, or (b) the exploitation of semantic composition for the purpose of modeling product lines and variability management

Finally, we observe that the technology for using design information together with AOP is becoming available in several languages and environments, and we believe that this chapter can contribute to a better understanding of the importance of using design information and the possible applications.

3.10 References

[1] Apache Common Attributes homepage. http://jakarta.apache.org/commons/sandbox/attributes/.

[2] Attrib4j homepage. http://attrib4j.sourceforge.net.

[3] JUNIT homepage. http://www.junit.org.

[4] Javabeans(tm) specification 1.01 release, Sun Microsystem, Aug. 1997.

[5] C# language specification, ECMA International, Dec. 2002.

[6] A metadata facility for the java programming language, 2002. JSR 175 Public Draft Specification.

[7] *OMG Unified Modeling Language Specification, Version 1.5*. Object Management Group Inc., Mar. 2003.

[8] AspectWerkz homepage. http:aspectwerkz.codehaus.org/.

[9] BERGMANS, L., AND AKSIT, M. Principles and design rationale of composition filters. In *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, Eds. Addison-Wesley, Boston, 2005, pp. 63–95.

[10] BONÉR, J. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)* (Mar. 2004), K. Lieberherr, Ed., ACM Press, pp. 5–6.

[11] COLYER, A. AspectJ. In *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Akc sit, Eds. Addison-Wesley, Boston, 2005, pp. 123–143.

[12] Compose* project. http://composestar.sf.net.

[13] DALAGER, C., JORSAL, S., AND SORT, E. Aspect oriented programming in JBoss 4. Master's thesis, IT University of Copenhagen, Feb. 2004.

[14] DE VOLDER, K., BRICHAU, J., MENS, K., AND D'HONDT, T. Logic metaprogramming, a framework for domain-specific aspect programming languages. http://www.cs.ubc.ca/~kdvolder/binaries/cacm-aop-paper.pdf.

[15] GYBELS, K., AND BRICHAU, J. Arranging language features for pattern-based crosscuts. In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (Mar. 2003), M. Aksit, Ed., ACM Press, pp. 60–69.

[16] HAVINGA, W. Designating join points in Compose* - a predicate-based superimposition language for Compose*. Masters thesis, University of Twente, 2005.

[17] JANZEN, D., AND DE VOLDER, K. Navigating and querying code without getting lost. In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (Mar. 2003), M. Aksit, Ed., ACM Press.

[18] KICZALES, G. Making the code look like the design, Keynote Talk at AOSD'03, Mar. 2003.

[19] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *Proc. ECOOP 2001, LNCS 2072* (Berlin, June 2001), J. L. Knudsen, Ed., Springer-Verlag, pp. 327–353.

[20] LADDAD, R. AOP and metadata: A perfect match. In *AOP@Work Series* (Mar. 2005), IBM Techincal Library.

[21] LOPES, C., BERGMANS, L., D'HONDT, M., AND TARR, P., Eds. *Workshop on Aspects and Dimensions of Concerns (ECOOP 2000)* (June 2000).

[22] NAGY, I., AND BERGMANS, L. Towards semantic composition in aspect-oriented programming. In *European Interactive Workshop on Aspects in Software (EIWAS)* (Sept. 2004), K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka, Eds.

[23] NAGY, I., BERGMANS, L., HAVINGA, W., AND AKSIT, M. Utilizing design information in aspect-oriented programming. In *Proceedings of International Conference NetObjectDays, NODe2005* (Erfurt, Gergmany, Sep 2005), A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, Eds., vol. P-69 of *Lecture Notes in Informatics*, Gesellschaft für Informatik (GI).

[24] NEWKIRK, J., AND VORONTSOV, A. A. How .NET's custom attributes affect design. *IEEE Software* (2002).

[25] SUVÉE, D., AND VANDERPERREN, W. JAsCo: An aspect-oriented approach tailored for component based software development. In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (Mar. 2003), M. Aksit, Ed., ACM Press.

Chapter 4

Evolvable Weaving Specifications

In the current aspect-oriented languages, advices and pointcuts are, in general, explicitly associated. This results in weaving specifications that are less evolvable and need more maintenance during the development of a system. To address this issue, we propose associative access to advices and aspects: a designating mechanism that allows for referring to aspect/advices through their (syntactic and semantic) properties in advice-pointcut bindings. First, this chapter presents an extensive analysis of the advice-pointcut binding mechanisms of the state-of-the-art AOP approaches. Based on this analysis, we extend the current weaving (superimposition) specification of our aspect-oriented approach, Compose. In the new specification, we apply queries that can designate filtermodules and other types of units (e.g. annotations) based on their properties. As an evaluation of our work, we present a tradeoff analysis about the new weaving specification with respect to several software quality factors, in particular expressiveness, evolvability and comprehensibility. The chapter ends with a discussion of related work and conclusions.*[1]

4.1 Introduction

Advices and pointcuts are among the most important language concepts of aspect-oriented languages [11]. Advices associated with pointcuts form a large part of the weaving (*superimposition*[2]) specification by describing, respectively, the subjects and places of weaving. We argue in this chapter that the way

1. This chapter is based on work published in [20], [21] and [15].
2. The composition of aspects with the base classes is called superimposition.

these elements are associated has a significant influence on the weaving specification with respect to software quality factors, such as expressiveness, evolvability and comprehensibility [19].

The current advice-pointcut binding mechanisms of AOP languages maintain explicit dependencies to advices and aspects. This results in weaving specifications that are less evolvable and need more maintenance during the development of a system. We believe that this issue can be addressed by providing *associative access* to advices and aspects instead of using explicit dependencies in the weaving specification. To this aim, we propose to use a designating (query) language in advice-pointcut bindings that allows for referring to aspect/advices through their (syntactic and semantic) properties.

This chapter is structured as follows: section 4.2 presents an extensive analysis of the advice-pointcut binding mechanism of various AOP approaches. Section 4.3 outlines our approach. Section 4.4 discusses the extension of Compose* [4, 7] for supporting evolvable weaving specifications. First, we provide a background on the current binding mechanism of Compose*. In the subsequent sections, we show our proposal to extend the superimposition specification of Compose* based on the analysis we performed before. These subsections also present a tradeoff analysis in which we evaluate the proposed mechanisms in the view of the above mentioned software quality factors. Section 4.5 provides details about the implementation. Finally, we close the chapter with related work and contribution in section 4.6.

4.2 An Analysis of Weaving Specifications

In the following sections, we look at the weaving specifications of the state-of-the-art AOP approaches, such as AspectJ [17, 10], AspectWerkz [3, 5] and JBoss [8]. In particular, we examine how certain elements (e.g. pointcuts, advices, etc.) of the weaving specification are associated with each other, how these elements can be reused in another weaving specification, what can be subjects of a weaving specification and to what degree the weaving specification can capture the evolution of concerns. By reflecting on these issues, our goal is to identify those language features that may have significant impact on a weaving specification.

4.2.1 Associating Advices with Pointcuts

Regarding the coupling between the advices and pointcuts, we identified two main categories of advice-pointcut bindings: *strong* and *loose coupling.*

Strong Coupling

In case of strong coupling, the definition of an advice includes the pointcut to which the advice is permanently bound.

The advice construct of AspectJ is a typical example for this type of association. Listing 4.1 shows a simple example of binding a before advice to a pointcut called tracedMethods.

```
1) pointcut tracedMethods(): execution…;
2) before(): tracedMethods(){
3)   if (TRACELEVEL == 0) return;
4)   if (TRACELEVEL == 2) callDepth++;
5)   printEntering(thisJoinPoint.getSignature());
6)   …
7) }
```

Listing 4.1 *Advice-Pointcut binding in AspectJ*

Strong association has a negative impact on the reusability of advices and aspects: the problem is that an advice is permanently bound to a pointcut. This has two consequences: (1) the advice cannot be associated with a new pointcut (i.e. with new join points); (2) the code of an advice cannot be reused from other advices, since an advice is not called like an ordinary method; it is only executed when the join point is reached by the control flow.

To circumvent the first problem, AspectJ has the concept of *abstract pointcut* which allows for deferring the specification of a pointcut that is already associated with an advice. In the example of Listing 4.2, the before advice is associated with the abstract pointcut tracedMethods in the abstract aspect Tracing. This abstract pointcut is concretized in the GUITracing aspect that inherits from Tracing. By applying this technique, the before advice can be associated with various pointcuts. The only disadvantage of this approach is that the comprehensibility of the code decreases, especially when the size of the project scales

up. The reason is that every subaspect should be read to determine the complete set of pointcuts (and join points) that the advice is associated to.

```
1) abstract aspect Tracing{
2)    abstract pointcut tracedMethods();
3)
4)    before(): tracedMethods(){
5)       if (TRACELEVEL == 0) return;
6)       if (TRACELEVEL == 2) callDepth++;
7)       printEntering(thisJoinPoint.getSignature());
8)       …
9)    }
10)   …
11)}
12)aspect GUITracing extends Tracing{
13)   pointcut tracedMethods():
14)      within (com.app.gui.*) && …;
15)}
```

Listing 4.2 *An example for an abstract pointcut in AspectJ*

To avoid the second problem and provide code that can be reused by several advices, we need to refactor the code of an advice into ordinary methods that can be called from any advice. We illustrate this in Listing 4.3 that shows the previous example after the corresponding refactoring.

```
1) abstract aspect Tracing{
2)    abstract pointcut tracedMethods();
3)    before(): tracedMethods(){
4)       Tracer.traceEntry(
5)            thisJoinPoint.getSignature());
6)    }
7)    …
8) }
9) public class Tracer{
10)  public static void traceEntry(String sign){
11)      if (TRACELEVEL == 0) return;
12)      if (TRACELEVEL == 2) callDepth++;
13)      printEntering(sign);
14)      …
15)   }
16)}
```

Listing 4.3 *Using helper methods in AspectJ*

In this example, the code of the before advice is replaced by a call to the traceEntry method of Tracer that contains the original code of the advice. As a result, the method can be called (i.e. reused) from different advices. Note that the information from the thisJoinPoint variable had to be extracted in the advice and passed as a parameter to the method, since this variable is available only in the context of an advice. Another important issue is that an around advice cannot be refactored in this way; the reason is similar: the proceed keyword is applicable only within the context of *around* advices in AspectJ.

Practically, abstract pointcuts and helper methods can be considered as useful techniques when one wants to create adaptable advices; however, they also have disadvantages, as we have presented them in the examples.

Loose Coupling
Advices and pointcuts are weakly associated if they are specified independently from each other and coupled in an independently defined binding specification.

The advice binding construct of AspectWerkz is a typical example of weak association. Listing 4.4 shows the implementation of the example of tracing in AspectWerkz. In AspectWerkz, aspects are represented by classes and the methods of those classes may act as advices. In the example above, the method traceEntry of Tracer contains the advice. The XML descriptor (lines 11-19) contains a pointcut definition called tracedMethods and an aspect specification. The aspect specification declares that the class Tracer will be treated (e.g. instantiated) as an aspect in the system, while the advice binding connects the method traceEntry as a *before* advice to the previously defined pointcut. Note that this sort of aspect specification may allow for reusing a class (that acts as an aspect) in the realization of different aspects (e.g. using different instantiation strategies). Similarly, using this type of advice binding allows for binding a method to different (types of) advices. As a result, it is possible to reuse not

only pointcuts but also the crosscutting functionality, represented by the method, for the implementation of different advices.

```
1) public class Tracer{
2)   public void traceEntry(JoinPoint thisJP){
3)       if (TRACELEVEL == 0) return;
4)       if (TRACELEVEL == 2) callDepth++;
5)       String signature =
6)          thisJP.getSignature().getName();
7)       printEntering(signature);
8)       …
9)   }
10)}
11)<aspectwerkz>
12)   …
13)   <pointcut name="tracedMethods"
14)             expression="…" />
15)   <aspect name="TracerAspect" class="Tracer">
16)      <advice name="traceEntry" type="before"
17)               bind-to="tracedMethods"/>
18)   </aspect>
19)</aspectwerkz>
```

Listing 4.4 *Advice-pointcut binding in AspectWerkz*

4.2.2 Multiplicity in Bindings

In the current AOP languages, the general case is that a pointcut specification (designating many join points) is associated with an advice in the binding specification.

The advice construct of AspectJ is a good example of a many-to-one binding. In AspectJ, advices are always defined with a pointcut reference (or with the pointcut definition itself). For example, Listing 4.1 shows a definition of a *before* advice bound to a pointcut called tracedMethods. Note that a pointcut can be bound to an arbitrary number of advices; however, a new binding specification should be defined for each binding. Listing 4.5 shows an example of this:

two before advices, located in different aspects, are bound to the same pointcut in two advice definitions.

```
1) aspect TracerAspect{
2)    pointcut tracedMethods(): execution…;
3)
4)    before(): tracedMethods(){
5)       if (TRACELEVEL == 0) return;
6)       if (TRACELEVEL == 2) callDepth++;
7)       printEntering(thisJoinPoint.getSignature());
8)       …
9)    }
10)}
11)aspect AnotherTracerAspect{
12)   before(): TracerAspect.tracedMethods(){
13)      /* another advice bound to the same pc */
14)      …
15)   }
16)}
```

Listing 4.5 *Two advices bound to the same pointcut in AspectJ*

The same problem appears in AspectWerkz, as well: a pointcut can be bound to several advices but we need to create a new binding specification for each case. Listing 4.6 shows an example for this. First, a pointcut called tracedMethods is declared; this is followed by two aspect mappings that contain two advice bindings. Note that in the fully qualified advice reference we refer not only to advices but also to the aspects that contain the advices.

```
1) <pointcut name="tracedMethods"
2)           expression="…"/>
3)
4) <aspect class="TracerAspect">
5)    <advice name="traceEntry" type="before"
6)            bind-to="tracedMethods"/>
7) </aspect>
8)
9) <aspect class="AnotherTracerAspect">
10)   <advice name="traceEntry" type="before"
11)           bind-to="tracedMethods"/>
12)</aspect>
```

Listing 4.6 *Two aspects bound to the same pointcut in AspectWerkz*

There are certain cases when it is necessary to weave not only one aspect or advice but a set of aspects or advices at a given join point. Clearly, the many-to- one type of binding specification is not sufficient in these cases. Whenever a new advice has to be bound to a given pointcut, a new complete binding specification has be created for it as well. As a result of this, as Listing 4.6 shows, we will get a set of binding specifications that contains (partially) duplicated information, which makes the code difficult to maintain.

Note that shared join points may appear when we bind a set of advices to the same pointcut, since advices will be woven at the same join point. We may need to provide control over the execution order of advices to avoid possible conflicts.

Many-to-Many Binding
In the advice-pointcut binding specification, a pointcut can be bound to a set of advices. Although this sounds as an obvious alternative, not too many aspect-oriented languages support this alternative.

JBoss provides a mechanism by which many-to-many bindings can be created between a pointcut and several advices. At the same time, this mechanism controls the execution order of advices. Using the previously introduced example, we illustrate this mechanism in Listing 4.7. TracerAspect and AnotherTracerAspect are implemented by classes, as previously. Both classes have the methods (traceEntry) that will behave as advices when these classes are mapped to aspects. Subsequently, the aspects are listed with the corresponding advices between the stack tags. The order of the listed aspects and advices does matter, since they are executed in the order of listing when the join point they are attached to is reached. This whole stack (called Tracing) is bound to the tracedMethods pointcut in the binding specification, defined within the bind XML tags.

Note that this technique is an enumeration that is similar to the binding of AspectWerkz, since the aspects are listed within the stack tags. On the other hand, using the stack structure for enumerating the aspects has some benefits as well. First, the stack acts a 'virtual' module for a set of aspects that are wrapped into it. Thus, a set of aspects can only be referred to by one reference in the binding specification. (In other words, a set of aspects can be bound to

several pointcuts through only one reference.) Secondly, as aspects are organized into a stack structure, the execution order of their advices at shared join points is also provided.

```
1) public class TracerAspect{
2)    Object traceEntry(Invocation object)
3)                                  throws Throwable
4)    {…}
5) }
6)
7) public class AnotherTracerAspect{
8)    Object traceEntry(Invocation object)
9)                                  throws Throwable
10)   {…}
11)}
12)
13)<stack name="Tracing">
14)  <advice name="traceEntry"
15)          aspect="TracerAspect"/>
16)  <advice name="traceEntry"
17)          aspect="AnotherTracerAspect"/>
18)</stack>
19)
20)<pointcut name="tracedMethods" expr=… />
21)
22)<bind pointcut="tracedMethods">
23)       <stack-ref name="Tracing"/>
24)</bind>
```

Listing 4.7 *Enumerating advices in the binding specification in JBoss*

However, there is a problem with using enumeration (or similar techniques, like the stack) in many-to-many bindings: the binding specification has to be modified whenever a new aspect is involved in the binding specification (or an existing one is removed). In other words, the enumeration based techniques do not provide *evolvable* binding specifications.

4.2.3 Associative Access

If we take a closer look at the binding specifications we can see an important difference between the bindings of AspectJ and AspectWerkz (or JBoss). In the case of the latter languages, we refer to the name - that is, the identity - of an aspect and advice in the binding. That is, aspects and advices are bound to

pointcuts based on their identity. In general, aspects or advices can share common properties. For instance, aspects with similar functionality can have a common (design information) property that denotes their analogous semantics. Taking the well-known example of aspect-oriented programming, we can classify aspects with monitoring functionality as *monitoring aspects*. Similarly, other design information can be easily associated with aspects or advices, considering various issues, such as their domain (e.g. *security, persistence aspects*), or their implementation characteristics (e.g. *singleton aspects*). As another well-know example, we can mention the classification used in the AspectJ Programming Guide [23]: *development, production aspects*.

By involving the design information (semantic properties) in the binding specification, aspects can be *designated for weaving based on their design intention*. For instance, assume a simple example: we would like to bind the advices of "development" aspects to a given pointcut. That is, we would like to bind each advice of all "development" aspects to a certain join point. By referring to aspect and advices through their semantic properties, we can provide *associative access* to aspects and advices. The benefit of this approach is that the weaving specification becomes more evolvable and less fragile to changes. Currently, none of the existing aspect-oriented languages provides designating mechanisms that can be used to select aspects and advices for weaving. In section 4.4, we show how Compose* can be extended to select aspects for superimposition.

Note that in case of the advice designation that we described above, shared join points may occur, similar to when we enumerated (or partially duplicated) the binding specifications. As we wrote before, this issue should be addressed to avoid possible conflicts.

4.2.4 Subjects of Weaving

In the current AOP approaches, the weaving specification, in general, binds behavioral advices to pointcuts. However, different types of introductions and structural constraints (see the reference model in Chapter 2) are typical examples for structural advices that are bound to pointcuts. Similar to behavioral advices, these structural advices formulate crosscutting concerns over the structure of the application.

In Figure 4.1, we show an example from AspectJ where the subjects of pointcut binding are not behavioral advices.

```
1) /*--- AspectJ Examples ---*/
2)
3) pointcut register():
4)    call(void Registry.register(FigureElement));
5) pointcut canRegister():
6)    withincode(static * FigureElement.make*(..));
7)
8) declare error:
9)    register() && !canRegister(): "Illegal call";
10)
11)declare annotation:
12)   org.xyz.model..* : @BusinessDomain;
```

Listing 4.8 *Different types of structural advices bound to pointcuts in AspectJ*

The first example of AspectJ is the statement declare error. This construct lets the compiler signal an error with the specified message ("Illegal Call") if the join point specified by the pointcut register() && !canRegister() matches. The construct declare annotation, in the newest release of AspectJ [22], specifies that all types defined in a package with the prefix org.xyz.model have the @BusinessDomain annotation. The common characteristic of these examples is that we bind pointcuts to structural advices, in this case, an *architectural constraint* and an *introduction*. Note that "org.xyz.model..*" is not an official pointcut in AspectJ; it is only a regular expression with a limited expressiveness, as compared to the fully expressive pointcut language of Compose*.

4.2.5 Summary

In our analysis, we have identified the following key properties that can improve the weaving specification:

Loose coupling provides reusable crosscutting behavior: advices and aspects can be reused in (i.e. adapted to) different applications, if they can be specified independently, and assigned to pointcuts later. For the same reason, loose coupling is a prerequisite for developing aspect libraries.

Many-to-many bindings yield more expressive binding specifications: as opposed to many-to-one bindings, a single specification can express the bind-

ing between a set of advices and a set of pointcuts. This means that we can avoid creating a new specification for each binding between an advice and the pointcut.

Associative access allows for designating a set of advices and aspects, based on their properties (or relationships to other units). Hence, the weaving specification is more evolvable and less fragile to changes, since advices and aspects are implicitly bound to pointcuts, through their properties.

Weaving subject polymorphism: in section 4.2.4 we observed that various types of language elements can be bound to pointcuts. This can lead to increased expressiveness of the language, as well as improved uniformity.

4.3 The Approach

Our goal is to enhance the binding specification to incorporate the features that we identified in the previous section. To do so, we propose abstractions that support these features. Figure 4.1 depicts an overview of our approach.

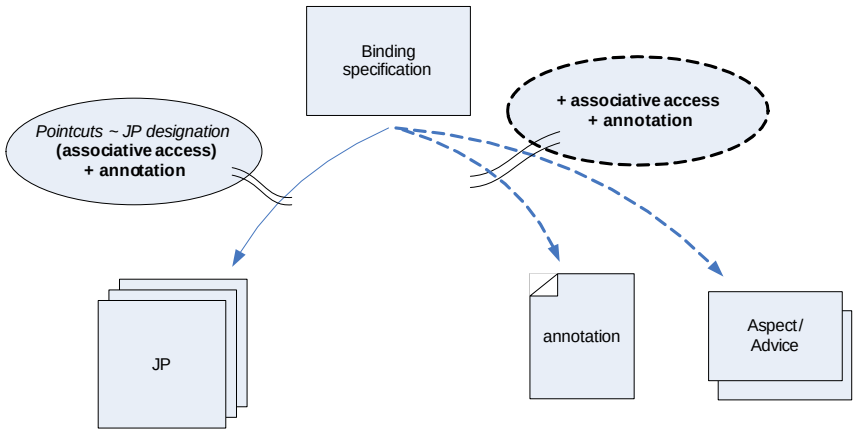

Figure 4.1 *Associative access in weaving specifications*

Pointcuts describe join points based on the properties of the join point (or shadow point) to be designated; i.e. pointcuts introduce *a level of indirection* in referring to particular join points. Similarly, we introduce a level of indirection (i.e. *associative access*) in referring to particular advices in the weaving specification (indicated by the dashed arrow). In this way, a set of join point

can be associated with a set of advices in a single binding specification. Technically, this means that we attach design information (in the form of annotations) to advices and aspects, and propose dedicated pointcuts to designate these units based on the annotated design information (Note that we proposed the designation of join points in a similar manner in the previous chapter.)

To superimpose (i.e. introduce) annotations on different program elements, we propose a pointcut language for *structural join point matching* that can also refer to annotations (attached to various program elements, such as classes, fields, methods, etc.). In this way, we can provide a technique to introduce annotations based on annotations and other properties that are already attached to other program elements. We call this technique *derivation of annotations*.

In the next section, we present an extension to the current superimposition specification of Compose* that realizes these features.

4.4 Extending Compose*

The following section gives details about the current superimposition specification of our aspect-oriented approach, Compose*. In section 4.4.2 and 4.4.3 we show our proposal to extend the superimposition specification of Compose*. These sections also contain a tradeoff analysis in which we evaluate the proposed mechanisms.

4.4.1 The Superimposition Specification of Compose*

Listing 4.9 presents a simple example case for the superimposition specification of Compose*.

Using the AspectJ terminology, we apply two *development* concerns in this example: Tracing and Profiling. Tracing contains a filtermodule (SimpleTracing) that implements the crosscutting functionality, i.e. tracing the execution of methods, in this case. Similarly, Profiling also has a filtermodule specification (SetsCounting) to realize the profiling feature: counting the changes of member variables through update methods (e.g. methods with 'set' prefixes). Since we are focusing on the aspect/advice-pointcut composition, we omitted to show the implementation of these filtermodules. The third, independent concern called WeaveDevelopmentAspects contains the superimposition specification.

```
1)  concern Tracing{
2)    filtermodule SimpleTracing{ … }
3)
4)    implementation in Java; …
5)  }
6)
7)  concern Profiling{
8)    filtermodule SetsCounting{ … }
9)
10)   implementation in Java; …
11) }
12)
13) concern WeaveDevelopmentAspects{
14)   …
15)   superimposition{
16)     selectors
17)       figureClasses = { FClasses |
18)          isClassWithName(Class,'FigureElement'),
19)          inInheritanceTree(Class,FClasses)
20)       };
21)     filtermodules
22)       figureClasses <- SimpleTracing,SetsCounting;
23)   }
24)   …
25) }
```

Listing 4.9 *Enumerating filtermodules for superimposition in Compose**

The superimposition specification consists of two parts: a selector definition (lines 14-18) and a filtermodule binding (lines 19-20). In Compose*, every selector has a name (figureClasses), which is unique within the concern where it is defined. This selector designates a set that consists of possible values for a variable (FClasses), where the predicates after the '|' puts constraints on these values. The first predicate (isClassWithName) binds the class FigureElement to the Class variable. The second predicate binds every class inherited from FigureElement to the FClasses variable by using unification. In the filtermodules part of the superimposition, the SimpleTracing and SetsCounting filtermodules are superimposed on each class that is designated by the previously defined selector (i.e. every class inherited from FigureElement). Thus, every instance of these classes will have an instance of the filtermodules superimposed.

Based on the classifications we presented in section 4.2, the filtermodule binding specification of Compose* can be described as many-to-many binding with loose coupling. However, we can also see in the example of Listing 4.9 that filtermodules are enumerated in the binding specification. As we discussed, the enumeration technique results in difficult code maintenance and not evolvable advice-pointcut bindings. For this reason, in the following sections, we show a proposal to extend Compose* with a new binding mechanism that can provide evolvable weaving specifications.

4.4.2 Querying Filtermodules

Instead of enumerating filtermodules, we aim at providing associative access to them. To achieve this, we apply selectors, the existing query mechanism of Compose*, to designate filtermodules based on their properties. This is illustrated in Listing 4.10 by a simple example.

The example starts with a concern specification, Tracing, that has two filtermodules: SimpleTracing and AdvancedTracing. The second concern, WeaveTracingModules, contains the superimposition specification. The first selector (figureClasses, in lines 13-17) was already explained the previous example; it will query all the classes inherited from the FigureElement class. The second selector (tracingModules) will query all the filtermodules within the Tracing concern (lines 18-23). In the filtermodule binding, we bind the selector tracingModules to the figureClasses selector. This means that every instance of the selected classes will have an instance of all filtermodules queried by tracingModules[1]. In this way, selectors are used with two purposes: (a) designating the classes on which the filtermodules are being superimposed; (b) designating the filtermodules that we will superimpose.

Trade-off Analysis

Positive Impacts on Software Quality Factors

Instead of enumerating the filtermodules one by one, we were able to designate a set of filtermodules based on their properties (or relationships to other units).

1. If we designate and bind a set of filtermodules to a selector, the problem of shared join points will occur. To avoid this problem we have to handle this issue in parallel by providing a superimposition order for filtermodules. We address this issue in Chapter 5.

Hence, the weaving specification becomes *expressive* with the application of queries both on the base classes and the superimposed filtermodules.

```
1) concern Tracing{
2)   filtermodule SimpleTracing{ … }
3)   filtermodule AdvancedTracing{ … }
4)
5)   implementation in Java;
6)   …
7) }
8)
9) concern WeaveTracingModules{
10)  …
11)  superimposition{
12)    selectors
13)      figureClasses =
14)        { FClasses |
15)          isClassWithName(Class,'FigureElement'),
16)          inInheritanceTree(Class,FClasses)
17)        };
18)      tracingModules =
19)        { FModule |
20)          isFilterModule(FModule),
21)          concernHasFilterModule(Concern,FModule),
22)          isConcernWithName(Concern,'Tracing')
23)        };
24)    filtermodules
25)      figureClasses <- tracingModules;
26)  }
27)  …
28)}
```

Listing 4.10 *Querying filtermodules in Compose**

When a new filtermodule is introduced within the concern Tracing, it is automatically captured by the query; therefore, the weaving specification becomes more *evolvable* as compared to the enumeration based technique.

Negative Impacts on Software Quality Factors

The programmer of filtermodules (or aspects) should be aware of the selectors that can potentially capture a newly introduced filtermodule. It might be intended to query the new filtermodule; however, this might not be intended at

all. Note that the enumeration of the filtermodules represents *explicit* dependencies on the filtermodules in the weaving specification. By changing enumerations to queries we turns these explicit dependencies into *implicit* ones, since queries refer to filtermodules through their properties. For this reason, the weaving specification becomes less transparent (*comprehensible*) for programmers but more intentional, as compared to the enumeration technique. Tools and intelligent IDEs (integrated development environments) can help to resolve these implicit dependencies and show them explicitly. For instance, the AJDT project [6] is a typical example of such a tool.

We have used only syntactical properties (e.g. a name of a concern) and structural relationships (e.g. a filtermodule contained by a concern) to formulate queries in the examples. However, if we have to modify (e.g. refactor) the code for certain reason, the queries that have worked until now may fail to work in the future. For instance, by renaming the concern Tracing, the tracingModules selector will not work in the last example. That is, the fragile pointcut problem [18, 12] may apply not only to the pointcuts that designate join points in the base code but also to the 'pointcuts' (or queries) that designate the aspect/ advices for weaving. In the following section, we propose to use annotations on filtermodules to address this problem.

4.4.3 Annotating Filtermodules & Concerns

Annotations (as known as custom attributes in .NET [1], or metadata facility in Java [2]) can be generally used to associate (semantic) properties with a language unit. In the previous chapter, we presented how pointcuts can make use of annotations attached to the base code. In this section, we show how filtermodules can be selected by using the generic query mechanism of Compose*. Listing 4.11 gives an example.

In Listing 4.11, we use the same example that we used in Listing 4.9, in section 4.4.1. Tracing and Profiling are both development aspects; hence, we attach the annotation Development to them. In WeaveDevelopmentAspects, we introduce a new selector, developmentModules, that queries all filtermodules contained by a concern with the annotation Development (lines 26-32). In the filtermodule binding specification, we bind this selector to the previously defined figureClasses selector. As a result, all filtermodules of each concern with the

annotation Development will be superimposed on the selected figure element classes.

```
1) [Development]
2) concern Tracing{
3)   filtermodule SimpleTracing{ … }
4)
5)   implementation in Java;
6)   …
7) }
8)
9) [Development]
10)concern Profiling{
11)  filtermodule SetsCounting{ … }
12)
13)  implementation in Java;
14)  …
15)}
16)
17)concern WeaveDevelopmentAspects{
18)  …
19)  superimposition{
20)    selectors
21)       figureClasses =
22)         { FClasses |
23)           isClassWithName(Class,'FigureElement'),
24)           inInheritanceTree(Class,FClasses)
25)         };
26)      developmentModules =
27)         { FModule |
28)           isFilterModule(FModule),
29)           concernHasFilterModule(Concern,FModule),
30)           concernHasAnnotationWithName(Concern,
31)           'Development') };
32)
33)     filtermodules
34)       figureClasses <- developmentModules;
35)  }
36)  …
37)}
```

Listing 4.11 *Querying filtermodules for superimposition based on their annotations*

Trade-off Analysis

Positive Impacts on Software Quality Factors
Whenever a new concern is introduced with the annotation Development, its filtermodules are automatically involved in the weaving specification. In addition, the concerns can have now arbitrary names, since the query does not refer to their names but to their design information, expressed by an annotation (Development) in this case. In general, the use of design information makes aspects, especially pointcut expressions and queries, less vulnerable to changes to the program. They provide *evolvability*, since they allow for avoiding dependencies of pointcut expressions upon the structure or naming conventions of a program.

The dependencies between program elements can now be made more precise and easier to understand by referring to semantic information (design intentions). Hence, there is a positive impact on *comprehensibility* as well.

Negative Impacts on Software Quality Factors
Disciplined programming is required to keep the correct semantic properties associated with the appropriate program elements.

Similarly, it is important that software engineers use a consistent and coherent set of semantic properties for each sub domain of an application (whether from a technical/solution domain, or from the application/problem domain). For instance, if programmers use annotations such as 'setter', 'writer', or 'updater' inconsistently, our approach has a limited value. For this reason, there is negative impact on the *predictability* of the superimposition specification.

4.4.4 Derivation of Annotations

In the previous chapter, we introduced a simple extension to the existing mechanism used to superimpose filtermodules in Compose*: a new language construct that specifies the superimposition of annotations on a set of selected program elements. The selector mechanism itself is exactly the same as the one used for superimposing filtermodules, and has Turing-complete expressiveness. Program elements can be selected based on their name, properties and relations to other program elements (i.e. based on the static structure of the

application), including annotations. A simple example is presented in Listing 4.12.

```
1) concern AppSpecificPersistence {
2)   superimposition{
3)     selectors
4)       transientClasses =
5)         { AnySess | isClassWithName(S, 'SessionID'),
6)                     classInheritsOrSelf(S, AnySess) };
7)     annotations
8)       transientClasses <- TransientClass;
9)   }
10)}
```

Listing 4.12 *An example of the introduction of annotations.*

In Listing 4.12, the class SessionID and its subclasses are selected by the selector transientClasses (lines 4-6). The annotation TransientClass will be superimposed (introduced) to this set of selected classes (line 8).

Previously, we extended the selector language of Compose* to use annotations as a selection criterion and introduced a language construct to superimpose annotations on a set of selected program elements. These two features can be combined to achieve the derivation of annotations.

To illustrate this, we extend the previous example by defining the following rule: If a field within a class that is marked by the annotation PersistentRoot is of a type that has the annotation TransientClass attached, it should be marked by the annotation TransientField. This way, we can specify an exception to the general rule that all fields within a class marked by the annotation PersistentRoot will be kept in a persistent data store. We express this example in Compose* in Listing 4.13. Here, the selector transientField selects all fields F, as long as they are of a type that has the annotation named TransientClass attached (line 3-5).

The annotation TransientField is superimposed on these fields (line 7).

```
1) superimposition {
2)   selectors
3)     transientFields = { F |
4)        typeHasAnnotationWithName(T, 'TransientClass'),
5)        fieldType(F,T) };
6)   annotations
7)     transientFields <- TransientField;
8) }
```

Listing 4.13 *Deriving an annotation*

Trade-off Analysis

Positive Impacts on Software Quality Factors
By extending Compose* to enable the superimposition and derivation of annotations, it is possible to separate the annotations from implementation classes, thus preventing the scattering of annotations. Also, this enables better separation between concerns, as they can select program elements based on the existence of annotations that may or may not have been attached by other concerns. Hence, the introduction of annotations has a positive impact on *modularity* and *adaptability*.

By supporting the derivation of annotations, dependent annotations (and complete annotation hierarchies) can be automatically superimposed. In this way, the inconsistencies caused by the manual attachment of such annotations can be avoided as well. Hence, the derivation of annotations has a positive impact on *predictability*.

Negative Impacts on Software Quality Factors
The use of derivation of annotations in several concerns could make it hard for a programmer to keep track of what will match a certain selector expression. In other words, the derivation technique may provide less *comprehensible* code as compared to the regular annotation introductions. However, by ensuring the declarativeness of selector expressions and superimposition of annotations, we believe that we could keep the use of this mechanism as straightforward as possible for the programmers.

4.5 Implementation

The superimposition and derivation of annotations as described in this chapter has been implemented the LOgical Language (LOLA) [14] and Superimposition ANalysis Engine (SANE) modules of the Compose* project (see Figure C.1 in Appendix C). A limitation in the current version is that parameters of annotations cannot be queried yet. Also, we intend to add support for writing superimposed annotations back to the IL code (to support non-aspect oriented frameworks). This functionality has not been implemented yet (i.e. superimposed annotations can only be used within Compose*).

4.6 Conclusion

4.6.1 Related Work

In section 4.2, we discussed the binding specification of AspectJ[17, 10], AspectWerkz [3, 5] and JBoss [8] in detail. Both AspectJ and JBoss provide facilities for weaving other subjects (e.g. annotations and introductions) than advices. The advantage of our approach over the weaving specification of these languages is the ability of selecting advices and aspects in the advice-pointcut bindings. Currently, none of above mentioned languages offers such a mechanism.

In the most recent version of AspectJ [22], it is possible to attach annotations to aspects and advices. Thus, a pointcut can designate the execution of an advice based on an annotation. However, unlike our approach, advices still cannot be selected and bound to pointcuts based on this information. Another difference is the inherent expression power of the pointcut languages. Compose* uses a Turing-complete general-purpose language (Prolog) with a pre-defined library of useful predicates, whereas AspectJ uses a more strictly defined, custom-defined pointcut language. Basically there is a tradeoff between supporting powerful reasoning within the pointcut language (here, Compose* offers more power) as opposed to reasoning about the pointcut expressions (which is easier in AspectJ).

JQuery [16] is a flexible, query-based source code browser, developed as an Eclipse plug-in. In JQuery, users can define their own queries to select certain elements of a program. The query language of JQuery is defined as a set of

TyRuBa [9] predicates which operate on facts generated from Eclipse JDT's abstract syntax tree. The predicates of JQuery are dedicated for Java and the factbase of JQuery is based on Java sources and bytecode files. In Compose* we also use a predicate language to formulate queries for defining selectors. The predicates of our selector language are dedicated for Compose* and the factbase is based on the repository model of a Compose* project.

4.6.2 Discussion

One might ask why it is useful or necessary to derive annotations (like TransientField in Listing 4.13) if their places can also be designated directly by the pointcuts that could express the rules above. Using such pointcuts can in fact be sufficient when these annotations are only used within pointcut expressions of aspects. However, (derived) annotations can be used by third party tools or frameworks as well. In many cases, we can derive whether a certain annotation should be attached based on the existence of other annotations, certain types of statements or structural combinations of program elements (i.e. 'software patterns'). In these cases, using derivation removes the need to manually specify where annotations have to be attached (either in the concern source or the source of the base application). Also, the use of (derived) annotations can increase the separation between concerns. Basically, better modularization is gained at the cost of an extra layer of indirection. Which choice is best depends on the scenario - i.e. the need for better modularization is typically an issue in large software applications.

However, the consequence of this derivation technique is that there will be ordering dependencies between the evaluation of queries and the superimposition of annotations. In [15], we analysed these dependencies and identified cases where dependency problems may arise. Based on this analysis, we designed an approach and implemented it as an algorithm to resolve the ordering dependencies and detect the possible dependency problems. We showed that this algorithm will always terminate, either by providing a correct resolution of the dependencies, or detecting if the superimposition specification is ambiguous. For further details, we refer to [15].

4.6.3 Contributions

In this chapter, we presented an extensive analysis of the pointcut-advice binding mechanisms of various aspect-oriented languages. Based on this analysis,

we proposed an extension to the current superimposition specification of Compose*, that can be realized in other languages in a similar manner. In the new specification we applied queries, founded on a generic, predicate-based language, that allow for designating both the places (i.e. structural join points) where we want to weave, and the units (e.g. filtermodule, annotation) that we want to weave.

We provided a trade-off analysis for each new abstraction that we introduced: the weaving specification is more expressive and evolvable compared to the existing binding mechanisms. By applying annotations (design information), we improved the evolvability of the binding specification to a greater extent; additionally, it is more intuitive to understand the composition by using design information. On the other hand, we observed that the weaving specification becomes less transparent (comprehensible) for programmers. We also observed that the use of annotations requires disciplined programming to bind the correct semantic properties to the appropriate program element. We believe that both of these issues can be addressed by tool support. Intelligent IDEs can improve the transparency problem by resolving and showing the implicit dependencies among the units involved in the weaving process. The "human error factor" related to the use of annotations can be lessened by using reasoning mechanisms and tools that can automatically derive annotation specifications, for example, based on control-flow or data-flow analysis. We consider addressing these issues in future work.

4.7 References

[1] C# language specification, Dec. 2002.

[2] A metadata facility for the java programming language, 2002. JSR 175 Public Draft Specification.

[3] AspectWerkz homepage. http:aspectwerkz.codehaus.org/.

[4] BERGMANS, L., AND AKSIT, M. Principles and design rationale of composition filters. In *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, Eds. Addison-Wesley, Boston, 2005, pp. 63–95.

[5] BONÉR, J. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)* (Mar. 2004), K. Lieberherr, Ed., ACM Press, pp. 5–6.

[6] CLEMENT, A., COLYER, A., AND KERSTEN, M. Aspect-oriented programming with AJDT. In *Analysis of Aspect-Oriented Software (ECOOP 2003)* (July 2003), J. Hannemann, R. Chitchyan, and A. Rashid, Eds.

[7] Compose* project. http://composestar.sf.net.

[8] DALAGER, C., JORSAL, S., AND SORT, E. Aspect oriented programming in JBoss 4. Master's thesis, IT University of Copenhagen, Feb. 2004.

[9] DE VOLDER, K. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[10] ECLIPSE FOUNDATION. AspectJ home page. http://eclipse.org/aspectj/.

[11] ELRAD, T., AKSIT, M., KICZALES, G., LIEBERHERR, K., AND OSSHER, H. Discussing aspects of AOP. *Comm. ACM 44*, 10 (Oct. 2001), 33–38.

[12] GYBELS, K., AND BRICHAU, J. Arranging language features for pattern-based crosscuts. In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (Mar. 2003), M. Aksit, Ed., ACM Press, pp. 60–69.

[13] GYBELS, K., HANENBERG, S., HERRMANN, S., AND WLOKA, J., Eds. *European Interactive Workshop on Aspects in Software (EIWAS)* (Sept. 2004).

[14] HAVINGA, W. Designating join points in Compose* - a predicate-based superimposition language for Compose*. Masters thesis, University of Twente, 2005.

[15] HAVINGA, W., NAGY, I., BERGMANS, L., AND AKSIT, M. Detecting and resolving ambiguities caused by inter-dependent introductions. In *Proc. 5th Int' Conf. on Aspect-Oriented Software Development (AOSD-2006)* (Mar. 2006), A. Rashid and H. Masuhara, Eds., ACM Press.

[16] JANZEN, D., AND DE VOLDER, K. Navigating and querying code without getting lost. In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (Mar. 2003), M. Akc sit, Ed., ACM Press.

[17] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *Proc. ECOOP 2001, LNCS 2072* (Berlin, June 2001), J. L. Knudsen, Ed., Springer-Verlag, pp. 327–353.

[18] KOPPEN, C., AND STÖRZER, M. PCDiff: Attacking the fragile pointcut problem. In Gybels et al. [13].

[19] NAGY, I., AND BERGMANS, L. Towards semantic composition in aspect-oriented programming. In Gybels et al. [13].

[20] NAGY, I., BERGMANS, L., GULESIR, G., DURR, P., AND AKSIT, M. Generic, property based queries for evolvable weaving specifications. In *3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop* (Mar. 2005), L. Bergmans, K. Gybels, P. Tarr, and E. Ernst, Eds.

[21] NAGY, I., BERGMANS, L., HAVINGA, W., AND AKSIT, M. Utilizing design information in aspect-oriented programming. In *Proceedings of International Conference NetObjectDays, NODe2005* (Erfurt, Gergmany, Sep 2005), A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, Eds., vol. P-69 of *Lecture Notes in Informatics*, Gesellschaft für Informatik (GI).

[22] THE ASPECTJ TEAM. The AspectJ 5 development kit developers's notebook, 2005. http://eclipse.org/aspectj/doc/next/adk15notebook /index.html.

[23] XEROX CORPORATION. The AspectJ programming guide.

Chapter 5

Composing Aspects at Shared Join Points

Aspect-oriented languages provide means to superimpose aspectual behavior on a given set of join points. It is possible that not just a single, but several units of aspectual behavior need to be superimposed on the same join point. Aspects that specify the superimposition of these units are said to "share" the same join point. Such shared join points may give rise to issues such as determining the exact execution order and the dependencies among the aspects. In this chapter, we present a detailed analysis of the problem, and identify a set of requirements upon mechanisms for composing aspects at shared join points. To address the identified issues, we propose a general declarative model for defining constraints upon the possible compositions of aspects at a shared join point. Finally, by using an extended notion of join points, we show how concrete aspect-oriented programming languages, particularly AspectJ and Compose, can adopt the proposed model.*[1]

5.1 Introduction

The so-called join point model is an important characteristic of every AOP language [7]. It defines a composition interface (“hooks”) - i.e. the types of locations, where the behavior of a (sub)program can be modified or enhanced, by superimposing aspectual (crosscutting) behavior. A join point is a specific element in the structure of a program (e.g. a class) or a specific event in the execution of a program (e.g. a method call). Almost all AOP languages allow the composition of independently specified aspectual behavior (i.e. *advice*) at

1. This chapter (except sections 5.5 and 5.9) is based on work published in [10] and [9].

the same join point, which we will refer to as a shared join point (SJP). The composition of multiple *advices* at the same join point raises several issues, such as: What is the execution order of the *advices*? Is there any dependency between them? These issues are not specific to certain AOP languages but they are relevant for almost every AOP language.

This chapter presents a novel and generic approach for specifying aspect composition at SJPs in aspect-oriented programming languages. We propose a declarative specifications of both ordering constraints and controlling constraints among aspects. In the following section, we will first introduce an example, which will be used for explaining the problems that may occur when composing aspects at SJPs, and present a detailed analysis of the problem. This analysis results in a set of requirements. In section 5.3, for specifying aspect composition at SJPs, we introduce a simple, generic model, which we term as Constraint Model. Section 5.4 discusses how the constraint model is enforced. Section 5.5 presents the applied algorithms. In section 5.6, we show how the concepts of the constraint model can be integrated with aspect-oriented programming languages. Section 5.7 provides details about the implementation. Section 5.8 discusses the related work. Section 5.9 provides an assessment of our approach in terms of software quality factors, such as comprehensibility, predictability, adaptability and modularity. Finally, section 5.10 draws conclusion and discusses the contributions of this chapter.

5.2 Problem Analysis

The superimposition of multiple *advices* on a particular join point involves several issues. To explain the possible problems, we introduce an example application, which will be used throughout the chapter.

5.2.1 Example

The example consists of a simple personnel management system. Class Employee, shown in Figure 5.1, forms an important part of the system. In particular, we will focus on the method increaseSalary(), which uses its argument to compute a new salary.

Our example has been defined as a scenario, which introduces a new requirement at each step. Applying the principle of separation of concerns, we imple-

ment each of these requirements by separate *aspects* that will be superimposed on the same join point (as well as others): in this example, after the call of the method increaseSalary() of class Employee[2].

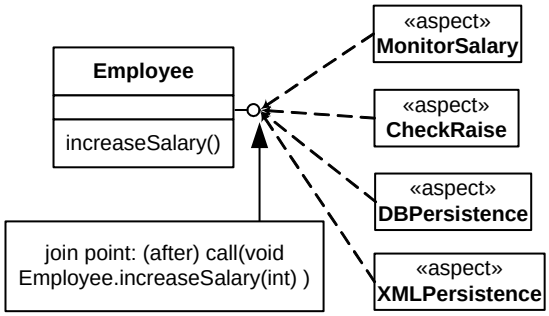

Figure 5.1 *Class Employee and its superimposed aspects*

We will use AspectJ for illustrative purposes. At each step, we show the possible problems that can occur at the SJP. We present an analysis of these problems and formulate the requirements towards their solution.

5.2.2 Primary Requirements

Step 1 – Monitoring Salaries
Assume that the first requirement in this scenario is to introduce a logging system for monitoring changes in salaries. This requirement is implemented by the aspect MonitorSalary in Listing 5.1.

In line 3, the pointcut salaryChange will designate every join point that is a call the method increaseSalary of class Employee. Whenever a salary is increased,

2. Note that not every aspect will be superimposed on the same set of join points. However, for all aspects there is a common join point which can be designated by the pointcut "`call(void Employee.increaseSalary(int))`" in AspectJ.

the *advice* (lines 6-11) will print a notification, including the information about the employee and the type of salary change.

```
1)  public aspect MonitorSalary{
2)     …
3)     pointcut salaryChange(Employee e, int l):
4)        target(e) && call(void increaseSalary(l));
5)
6)      after(Employee person, int level):
7)        salaryChange(person, level){
8)          System.out.println(“Salary increased to level"+
9)                              level + ” for person ”+ person);
10)         …
11)    }
12)}
```

Listing 5.1 *The advice and pointcut of the aspect MonitorSalary*

Step 2 – Persistence

The second requirement in the scenario states that certain objects must store their state in a database. After each state change in the corresponding objects, the database has to be updated as soon as possible. We consider persistence as a separate concern to be implemented as an aspect[3]. The abstract aspect DBPersistence contains the advice that performs the update operation on a persistent object:

```
1)  public abstract aspect DBPersistence
2)     pertarget (target(PersistentObject)){
3)
4)    abstract pointcut
5)       stateChange(PersistentObject po);
6)
7)     after(PersistentObject po): stateChange(po){
8)       System.out.println("Updating DBMS...");
9)       po.update();
10)       … }
11)}
```

Listing 5.2 *The abstract aspect DBPersistence*

3. There are several issues, such as connection, storage, updating and retrieval that have to be considered when dealing with persistence. For simplicity, we will focus here only on updating. More details about implementing persistence by aspects can be found in [12].

The following definition applies the abstract aspect DBPersistence to class Employee:

```
1) public aspect DBEmployeePersistence extends DBPersistence{
2)    /* Class Employee implements the interface
3)     * of PersistentObject */
4)   declare parents:
5)       Employee extends PersistentObject;
6)
7)   pointcut stateChange(PersistentObject po):
8)     call(void Employee.increaseSalary(int))
9)      && target(po) && … ;
10)  …
11)}
```

Listing 5.3 *An implementation of DBPersistence: DBEmployeePersistence*

These two aspects together implement the necessary behavior for making class Employee persistent. If the data of a persistent object changes, the corresponding information must be updated in the database too (Listing 5.2, the advice of the aspect). Changes to the state of the object are captured by the pointcut designator stateChange(PersistentObject po), which is implemented in DBEmployeePersistence. Note that the aspect MonitorSalary, which was required for the first scenario step, and the DBEmployeePersistence are now superimposed at the same join point.

Even though in most AOP languages *aspects* can be specified independently, once they are superimposed on the same join point, they may affect each others functionality. The problem may occur when both *aspects* and *classes* are added to a system. Figure 5.2 illustrates these two cases. On the left hand side, we show that superimposing a new aspect (CheckRaise) introduces a SJP, together with the previously superimposed aspect MonitorSalary. On the right hand side of the figure, it is illustrated that adding a new class can also introduce a new SJP, particularly when there are wildcards in *pointcut designators*.

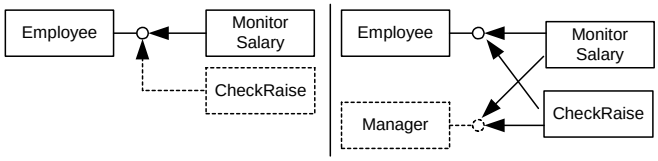

Figure 5.2 *Examples of creating possible SJPs*

Problem: Because the database needs to be updated as soon as possible after the state change occurs in the object, the advice of the aspect DBPersistence has to be executed before the advice of the aspect MonitorSalary.

Analysis: As the example illustrates, due to semantic interference, different execution orders among aspects at SJPs may exhibit different behavior. We distinguish the following categories of interference:

(A) *No difference in the observable behavior* – For example, consider two *aspects* where both do not refer to the effect of the other but maintain solely their own state. Changing the execution order of the two aspects at a SJP will not be observable after the execution of the *advices* of these two *aspects*.

(B) *Different order exhibits different behavior* – We have distinguished three subcategories of this category:

(B1) The change in the order affects the observable behavior but there is no specific requirement what the behavior should be – As an example of this case, assume that *one* aspect is designed to trace the change in salary and the other one to notify the employee's manager about any change in the salary. If the requirement is solely "both aspects should execute", it does not matter which aspect executes first. If there is an explicit requirement, however, the following category may apply:

(B2) The order of aspects *does matter* because there is an *explicit requirement* that dictates the desired order of aspects – A typical example is the interference between the aspects MonitorSalary and DBPersistence. The order between these aspects may seem to be not relevant, because they are defined as independent *aspects*. However, for DBPersistence there is a requirement: it should execute as soon as possible after a state change[4] occurs. Since there is no such requirement for MonitorSalary, this implies that DBPersistence must be executed before MonitorSalary.

(B3) There is no explicit requirement for an order, but certain execution orders can violate the desired *semantics* of the aspects. For instance, when multiple

4. In fact, in this case the rationale for this feature has to do with the observable different behavior in the case of crashes.

advices lock shared resources, deadlocks may occur in certain execution order of advices. This means that due to the semantics of the advices, there is in fact some kind of dependency between these advices and *implicit ordering requirements* have to be considered.

Requirement 1: Ordering Aspects – To ensure the required behavior of the superimposed *aspects* at SJPs, it must be possible to specify the execution order of the aspects[5].

Step 3 – Checking Salary Raises

Assume that the next requirement in this scenario is to ensure that an employee's salary cannot be higher than his/her manager's salary. Thus, a raise is not accepted if it violates this criterion. This is enforced by the aspect CheckRaise:

```
1) public aspect CheckRaise pertarget(target(Employee) ){
2)    private boolean _isValid;
3)    public boolean isValid(){ return _isValid; }
4)
5)    before(Employee person, int level):
6)       MonitorSalary.salaryChange(person,level){
7)          _isValid = true;
8)    } // workaround for conditional execution
9)
10)   after(Employee person, int level):
11)      MonitorSalary.salaryChange(person,level){
12)         Manager m=person.getManager();
13)         if ((m!=null) && (m.getSalary() <=
14)           person.getSalary()) ){
15)          //Warning message
16)         System.out.println("Raise rejected");…
17)           //Undo
18)          person.decreaseSalary(level);
19)         //workaround for conditional execution
20)          _isValid = false;
21)   }}}
```

Listing 5.4 *The aspect CheckRaise*

5. Some AOP languages, for example AspectJ, provide means to specify precedence between aspects, which implies an execution order.

The *advice* of this *aspect* (Listing 5.4) will check the new salary after the method increaseSalary() is executed[6]. If the rule is violated, a warning message will be printed and the salary will be set back to its original value.

Problem: Adding the aspect CheckRaise affects the composition; if this aspect fails, the DBPersistence aspect must not be executed because the employee's data has not changed. That is, the execution of the aspect DBPersistence depends on the outcome of the aspect CheckRaise.

Analysis: Implementing conditional execution of aspects is not trivial since the AOP languages do not provide explicit language mechanisms for this purpose. For example, in AspectJ we can use workarounds, such as maintaining Boolean member variables in aspects, but effective (incremental) composition cannot be achieved in this way. A possible workaround is illustrated by the highlighted code in Listing 5.4. As the example will show, it is necessary to introduce extra advices to maintain the Boolean variables and additional if-statements in the existing aspects to handle these variables. Consider, for instance, Listing 5.5 which shows a modified version of DBPersistence. A new if-statement has been added to check if the raise has been accepted by the aspect CheckRaise before executing the original behavior of the advice.

Another disadvantage of this solution is that *aspects* will depend on each other. That is, to realize the expected behavior of the composition, *aspects* will need to refer to each other directly. The invocation of the method isValid in Listing 5.5 is a typical example of such a dependency. In addition, problems will also occur when CheckRaise, for some reason, is removed from the project.

Requirement 2: Conditional execution – This requirement refers to a case when the execution of an aspect depends on the outcome of other aspects. Only if the outcome of these aspects satisfy a certain criterion, the dependent *aspect*

6. An alternative solution could be the prevention of an invalid raise using a *before advice* (as a pre-condition) instead of an *after advice*. However, this is not feasible in all cases; e.g. it is undesirable to repeat complex salary calculations, as this creates replicated code and may also incur a performance penalty. Another argument is that the method increaseSalary() may be overriden and one cannot rely on the current body.

is allowed to execute. To avoid workarounds and their shortcomings, direct language support is needed for expressing this type of dependency.

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

Listing 5.5 *The modified version of DBPersistence composed with CheckRaise*

Step 4 – Updating XML Representations

Assume that the fourth requirement in the scenario states that if the database is not available, persistence must be implemented using XML files. This means, for each instance of Employee, an XML file has to be generated. If the regular persistence does not take place (e.g. because of database connection problems), the file must be updated after each state change of an instance of class Employee. This is realized by the aspect XMLPersistence in Listing 5.6. This aspect has one advice, which calls the method that rewrites the XML file if the salary (or other data) changes.

```
1) public aspect XMLPersistence {
2)    after(XMLPersistentObject po): stateChange(po){
3)      if ((CheckRaise.aspectOf((Object)po).isValid())
4)          &&(!DBEmployeePersistence.aspectOf(
5)             (Object)po).isUpdated())
6)        po.toXML();
7)     }
8) }
```

Listing 5.6 *The aspect XMLPersistence*

In this example, XML files must be updated only if the aspect DBPersistence has not been able to update the database. This means that XMLPersistence must be executed only if DBPersistence has failed and CheckRaise has succeeded.

We identified several dependencies among aspects at SJPs. If there is no explicit language support for expressing the dependencies, they have to be implemented as workarounds in the realization of aspects. This has generally a negative impact on adaptability and reusability. In this chapter, we argue that there is a need for introducing new operators for expressing composition of aspects at shared join points. These operators must be capable of expressing both ordering among aspects and conditional execution of aspects.

5.2.3 Software Engineering Requirements

In the previous section, we presented the requirements from the aspect interference viewpoint. In this section, we list software engineering requirements that may play an important role in the quality of programs.

Modularization of composition specifications

From a software engineering perspective, not only the orthogonality of operators but also the structure and modularization of composition specifications play an important role. In particular, new dependencies are introduced since the specifications need to refer to specific join points, advices and aspects.

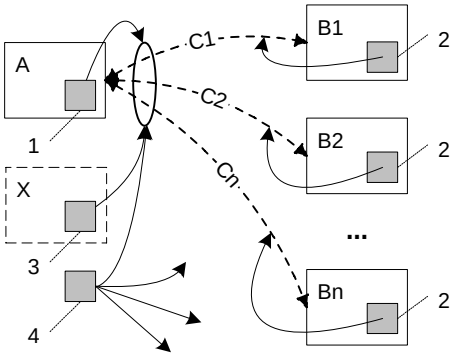

Figure 5.3 *Four alternative modularizations of constraint specifications; A, X and Bi are aspect specifications, Ci are composition specifications, and the grey squares (1 to 4) indicate alternative specification loci.*

Figure 5.3 illustrates a situation where between the *aspect A* and a series of *aspects B1* to *Bn*, the composition specifications *C1* to *Cn* apply, respectively. The figure shows four alternative modularizations of the composition specifications; each of these is shown as a grey square, labelled with a different number. We will discuss each of these numbered alternatives briefly:

1. A combined specification of *C1* to *Cn* is embedded in the specification of the *aspect A*; consequently, this aspect will depend on (refer to) *B1* to *Bn*. The introduction of a new aspect, say *Bm*, can either be handled automatically by the use of an open-ended specification (as will be discussed in section "Evolvability: Supporting open-ended specifications"), or it may require an additional effort to modify the corresponding specification of the *aspect A*.
2. The composition specification is partitioned and the corresponding specifications are located in *B1* to *Bn*, respectively; as a result, each of these aspects *Bi* will now depend on *A*. A newly introduced aspect, say *Bm* must then incorporate the composition specification *Cm*.

A critical issue in the above two cases is that the *aspects A* and *Bi* now include knowledge about how they depend on each other. In certain cases, this may be exactly what is required, but for example if the two aspects come from different (third-party) libraries, this is not desirable.

3. The composition specification is represented in a separate module (labeled *X* in the figure); In this case, the *aspect* specifications do not depend on each other. *X* can be either defined as a dedicated module for describing the composition of aspects, or it is a part of another module (e.g. an *aspect* or a class). Obviously, *X* will now depend on both *A* and *B1* to *Bn*. Changes to any of these may require an update to *X*. This allows for localizing composition specifications in a set of dedicated modules, if desired.
4. All the composition specifications are collected in one global module (c.f. a configuration file); this is a special case of alternative (3), and has the same dependency issues. In this case, all composition specifications are collected in a single location, which makes it easier to get an overview. However, scaling up to a large system will be more difficult, as the module consequently becomes larger. Obviously, each

change to the structure of the system requires a potential revision of this global module.

Based on this analysis, we conclude that it is not desirable to offer a solution which satisfies only a single case; AOP languages should offer a rich set of language mechanisms for composition specifications so that the programmers may choose the right modularization alternative for their problem.

Soundness: Identifying inconsistencies
An important design consideration is that programmers should be warned if their specification is not sound. A specification is sound when it contains no inconsistencies. This is especially important if the complete specification is made up from several sub-specifications defined at different locations. For example, creating circular relationships is a typical error that can occur in such a case. When a programmer creates a new composition specification, he or she must be warned if the new specification is inconsistent with other specifications.

Evolvability: Supporting open-ended specifications
Open-ended specifications in this context ensure that a specification is resilient to changes. Open-endedness may appear in the following forms:

1. The specification directly refers to an abstraction (a language element) that is not (yet) defined. In this case, open-endedness means that the specification is still correct and usable, even though some abstractions that the specification refers to have not been yet defined. Clearly, this requires a well defined meaning for the 'undefined' case.
2. The specification indirectly (by defining a number of selection criteria) refers to a set of potential abstractions. In this case, open-endedness means that, if a new abstraction appears in the environment and satisfies those criteria, it will be in the set of actual abstractions designated by the specification.

If developers use open-ended specifications for composing *aspects* in SJPs, both types of specifications will be able to handle *aspects* that are referred to, but not yet present in the system.

Lack of specification: non-deterministic execution
If there is no complete specification of the execution order, but sequential execution is required, the question remains in what specific order to execute the actions. Various types of 'default' orders can be defined; however, there is another alternative that should be considered.

The lack of specification may originate from two sources: uncertainty and omission (programmers merely forgot to specify order). Whatever the case may be, the order of the execution of aspects will not be clarified. A non-deterministic execution order between aspects, for which the order was not specified for some reason, can express the lack of ordering information. This means that each time the control flow reaches the join point, aspects are selected randomly for execution. In this way, non-determinism can express the uncertainty in the execution order. It is important that the programmers are warned about the non-deterministic execution so that they can specify a particular order if necessary.

There are domains (e.g. distributed and concurrent systems) where non-determinism can be favourable. However, non-determinism involves important implementation issues such 'real' random selection and performance.

5.3 Constraint Model

The problem of shared join points is general to AOP languages. For this reason, we propose a generic solution model that can be possibly built into various AOP languages. The aim of this section is not to present a formal foundation, but to illustrate the approach in an intuitive and concrete, but language-independent way. This requires a set of assumptions about AOP languages, which are presented in the subsection Basic Entities. The rest of this section presents the *composition constraints* and other means to specify composition of aspects at SJPs.

Our proposed model for composing aspects at SJPs is based on declarative specifications of constraints. Constraints define dependencies between actions. We distinguish between three main categories of constraints: *structural constraints, ordering constraints* and *control constraints*. Structural constraints specify what actions have to be or cannot be mutually present at a shared join point. Ordering constraints specify a partial order upon the execu-

tion of a set of actions. Control constraints specify conditional execution of actions.

5.3.1 Basic Entities

In this section, we outline the key elements of AOP models, that we consider relevant to our purpose. In order not to be too restrictive, it is important to make only a few assumptions about these entities. In particular, we focus on *join points* and *advices*[7].

Join Points

AOP languages have different means to designate join points. Thus, the range of the possible join points that can be designated varies from language to language. We do not make further assumptions about the designators. We just assume that there are certain events (e.g. calling a method) in the execution of a program where aspectual behavior can be executed.

Actions

In our model, the aspectual behavior (i.e. *advice*) that can be executed at join points, is abstracted under the concept of *action*. An action has a name that is used for identification. Advices will be mapped to actions when we adapt the constraint model to a particular aspect-oriented language. We consider the mapping of *after* and *before advices* as trivial cases. *Around advice* will be mapped to two actions; a before action and an after action[8].

An action may have a result value[9]. For the purposes of our model, we only allow Boolean result values. These typically indicate a success (true) or a failure (false) of the action. A key reason for this restriction to Boolean values is that it guarantees uniform interfaces between the actions and constraints. Allowing for more freedom in choosing result types would create undesired coupling, since constraints would become dependent on –the compatibility of– the result types of the actions. For instance, in the case of the example problem where persistence was required, the action that is responsible for updating the database will indicate a failure if it cannot connect to the database for some

7. These two entities have been identified among the main ‘ingredients’ of AOP languages [7].
8. Note that this mapping does not cover all the detailed semantics of around advice but it is sufficient to reason about the ordering of advice executions.
9. A result value is independent from the original return type of an advice.

reason. If an action does not have a result value, we use by default the void value for this purpose. In our model, the result values of actions will be used to express certain behavioral dependencies among actions at the same joint point.

By default, every action assigned to the join point will be executed, unless specified otherwise. The execution of actions is sequential[10], that is, only one action executes at a given time. An action may not execute multiple times. How to handle this is considered a language-design issue. In the absence of ordering constraints, the execution order of the actions is undefined[11]. Typically, a fixed order can be determined at compile-time, and be applied for each execution. Alternatively, a random order of actions may be generated for each execution; this can result in a non-deterministic execution order.

5.3.2 Structural Constraints

Structural constraints aim at specifying what actions have to be or cannot be mutually present at a shared join point.

Inclusion and Exclusion of Actions

There are two types of structural constraints; their definitions are the following:

include(x,y) – the presence of action x (i.e. x is actually woven) at the join point implies that action y has to be *present* at the same join point as well.

Note that the constraint include is unidirectional; it exerts its effect only in one direction. If action y is present at compile time, action x can be absent or present. For instance, include(x,y) does not imply that x should (or not) be present at the join point if y is present.

The constraint include has a transitive property: include(x,y); include(y,z) implicitly implies the include(x,z) constraint.

10.Parallel execution is an orthogonal issue; if synchronization between (actions executing in) multiple threads is needed, this is not a different problem from regular issues of thread-safe code. In this chapter, we focus on the sequential execution of aspects. Parallel execution of actions at shared join points is outside the scope of this work. In particular, we have not encountered any motivation for exploring this further.

11.In this case, the programmer should be warned about possibly unspecified orderings.

exclude(x,y) – the presence of action x (i.e. x is actually woven) at the join point implies that action y has to be *absent* at the same join point. If action y is present action x has to be absent, otherwise the contraint is not satisfied. This means that the presence of any of the actions mutually excludes the presence of the other action .

Hence, the constraint exclude is bidirectional. For instance, exclude(x,y) imply that y cannot be present at the join point if x is present, and the other way around, x cannot be present at the join point if y is present already.

Conflicts among Structural Constraints

We have identified two types of conflicts that can arise in the specification of structural constraints:

Straight Conflict

Two or more constraints have opposing statements in the specification. For instance, the following two constraints are in straight conflict: include(x,y); exclude(x,y). That is, one constraint states that y should be present at the join point, while the other one states the opposite.

Due to the transitive property of the include constraint, there can be more complicated cases that lead to straight conflicts in the specification. As an example, consider the following specification: include(x,y); include(y,z); exclude(x,z). The source of conflict is that include(x,y); include(y,z) implies include(x,z) which is the opposite of exclude(x,z).

Reverse Conflict

Two or more constraints have reverse statements in the specification. As an example, consider the following two constraints that are in reverse conflict: include(x,y); exclude(y,x). In this specification, the first inclusion constraint states that the presence of x needs the presence of y as well. However, the second exclusion constraint states that the presence of y forbids the presence of x at the join point.

Again, due to the transitive property of include there may be more complicated cases that cause reverse conflict in the specification. As an example, consider the following specification: include(x,y); include(y,z); exclude(z,x). The source of

conflict is that include(x,y); include(y,z) implies include(x,z) that is in reverse conflict with exclude(z,x).

5.3.3 Ordering and Control Constraints

Ordering Constraints

Ordering constraints specify a partial ordering over actions. When several actions are superimposed upon the same join point, all these actions are assumed to execute once, in an unspecified order. This implies that there can be many possible valid orderings. By applying ordering constraints, the number of possible orders can be decreased. For example, assume that four aspects are superimposed on the same join point, as shown in section 5.2. Without any ordering constraints, the number of possible execution orders is 4!=24. To be able to specify ordering, we need to introduce an ordering constraint.

Constraint pre

A pre constraint between two actions specifies that the execution of constrained action should precede the execution of another action at the SJP:

pre(x,y) – The followed order of actions is such that x should never be executed after the execution of y. Hence, y should be executed only after x has been executed at this join point[12]. (The two actions do not have to follow each other directly; other actions can be executed between them.)

We use Table 5.7 to illustrate the definition of constraints that are applied to two actions, respectively *x* and *y*. The topmost row of the table shows the applied constraints. Let us now focus on the column of the constraint pre. The leftmost column lists the possible values (true, false and void) that the action x can have after its execution. The last item in this column is the special case when the action x has not been executed for some reason. According to the applied constraint and the return value of x, the remaining cells of the second column from left indicate if y is allowed to execute after the execution of x or not. We can see in this figure that the pre constraint is not influenced by the

12.In general, constraints do not allow for the execution of an action if the dependent action did not execute. In other words, we deal with *hard* constraints. To be able to specify open-ended constraint specifications, we introduce additional functions that are discussed in section 5.3.4.

return value: in each case y can be executed after x is executed. The last cell in this column shows that y is not allowed for execution if x did not run.

Table 5.7 *The execution semantics of the composition constraints: y:yes means that y can be executed according to this specification; y_{return}=R means that return value of y is substituted with R*

	pre(x,y)	cond(x,y)	skip(x,y,R)
x: true	**y: yes**	**y: yes**	**y:no, {y_{return} = R}**
x: false	**y: yes**	**y: no**	**y: yes**
x: void	**y: yes**	**y: no**	**y: yes**
x: did not run	**y: no**	**y: no**	**y: yes**

In Table 5.8 we illustrate how the pre constraint decreases the number of possible orders. We use the case that we have introduced in section 5.2.2. As a short hand notation, we show only the first letter of the name of an *action*. We assume that all four actions (C = CheckRaise, D = DBPersistence, M = MonitorSalary, X = XMLPersistence) are superimposed upon the same join point. In the middle column, we list the constraints applied, and correspondingly in the right column we list all the possible orders which are valid. In the first row (Case I.) we apply only one constraint specifying that DBPersistence should be executed before MonitorSalary. The last six possible orders of Case I are those cases where the execution of DBPersistence and MonitorSalary are interleaved with other actions (C and/or X).

Table 5.8 *The possible execution orders decrease as new constraints are added*

Case	*Constraints*	*Possible Orders*
I.	**pre(D,M).**	***DM***CX, C***DM***X, CX***DM***, ***DM***XC, X***DM***C, XC***DM*** ***D***C***M***X, ***D***CX***M***, C***D***X***M***, ***D***X***M***C, ***D***XC***M***, X***D***C***M***

Table 5.8 *The possible execution orders decrease as new constraints are added*

Case	*Constraints*	*Possible Orders*
II.	pre(D,M), **pre(D,X).**	C***DMX***, C***DXM***, ***D***C***MX***, ***D***C***XM***, ***DMX***C, ***DM***C***X***
III.	pre(D,M), pre(D,X), **pre(C,D).**	***CDMX, CDXM***

In Case II., we add a new pre constraint, which specifies that DBPersistence should precede XMLPersistence as well. By applying two ordering constraints, the number of valid orders are reduced to six in this case. In the third row (Case III), after applying three constraints, there are only two alternatives left. Here, only the order between MonitorSalary and XMLPersistence is not fixed.

Control Constraints

Control constraints express conditional execution dependencies between actions. The general form of a control constraint is the following: *Constraint(Condition, ConstrainedAction)*. The *Condition* is represented by an action or a Boolean expression built up from actions with logical connectors (AND, OR, NOT). Control constraints use the return value of the executed actions for constraining the execution of *ConstrainedAction*.

Constraint Cond

The cond constraint specifies that an action is conditionally executed depending on the return value of another action. The definition of the cond constraint is the following:

cond(x,y) – Action y can execute only if x returns true. That is, y will not execute in case of the following four conditions: (1) if x returns false; (2) if x returns void; (3) if x has not been executed, or (4) if x is not present at the join point.

For the cond constraint, a Boolean return value is desired. Hence, if strong typing is applied to the return values of *actions* and the arguments of constraints, the void case can be avoided. We have deliberately included the

return value void as a legitimate case to make the system more flexible and applicable to a wide range of languages: (a) void is the default return value if a programmer does not indicate failure of an action; (b) the return value void can be intentionally used to indicate that the action results neither in success nor in failure.

The column of cond(x,y) in Table 5.7 illustrates the meaning of the cond constraint: y can execute only if x succeeded (i.e. x returned true). Note that when x did not execute, cond does not allow for the execution of y.

Table 5.9 *Using the* cond *constraint*

Case	*Constraints*	*Possible Cases*
IV.	**pre(D,M),** **pre(D,X),** **cond(C,D).**	***C = true*** **> *CD*MX, *CD*XM** ***C = false*** **> *C*MX, *C*XM** ***C = did not run*** **>** MX, XM

Table 5.9 demonstrates the effect of the cond constraint. In Case IV, we have changed the third constraint of Case III to cond(CheckRaise, DBPersistence). Depending on the return value of CheckRaise, there are two sets of possible orders. When CheckRaise returns true (the first row of *Possible Cases*) the possible orders are the same as the one of the pre constraint. However, when the return value of CheckRaise is void or false (the second row of *Possible Cases*) DBPersistence will not be executed. The third row "C did not run", shows that both CheckRaise and DBPersistence do not execute in this case.

Constraint Skip

The skip constraint specifies that the execution of an action may be skipped, based on the result of the logical expression. The definition of the skip constraint is the following:

skip(x,y,R) – The execution of y is skipped and y marked as 'executed' with the return value R, only if x yields true.

R substitutes the original return value of y if y is skipped. For example, R can be true, false or void, but an arbitrary logical expression can also be used to express the return value.

In Table 5.7, we show the behavior of skip: y is skipped only if x has been succeeded (i.e. x was true). In addition, the return value of y is substituted with R.

Table 5.10 *Using the* skip *constraint*

Case	*Constraints*	*Possible Cases*
V.	**pre(D,M), cond(C,D), skip(D,X,F).**	***C∧D = true > CDM{X←F}*** ***C∧!D = true > CDMX, CDXM*** ***C = false ∧ D=did not run > CX***

In Table 5.10, the first row under the cell *Possible Cases* shows that when both CheckRaise and DBPersistence succeed, XMLPersistence is skipped as if it has returned a false value. In the middle row, CheckRaise succeeds but DBPersistence fails, so XMLPersistence is executed. The third column on the right hand side shows that XMLPersistence will also be executed in the absence of DBPersistence.

Note that other possible scenarios may occur for both control constraints. We have chosen only those scenarios that we considered important for the illustration of the behavior of control constraints. With the cond constraint the execution of an action is controlled on the basis of information that originates from the *past*. Basically, the cond constraint formulates a guard condition on the execution of the constrained action. Using the skip constraint it is possible to "skip" the execution of an action that and mark it is an executed one with a given return value. Although it is possible to introduce additional constraints, based on the example cases that we have carried out, we believe that the three constraints pre, cond and skip are powerful enough for expressing a large category of conditional constraints.

Composition Rules for Multiple Constraints

The constraints discussed so far, are to a large extent orthogonal to each other. However, when multiple constraints apply to the same action, certain rules must be considered to resolve the composition of constraints.

Precedence of Constraint Types
If different types of constraints apply to the same action, e.g. skip(x, z, true); cond(y, z) the constraints are evaluated in a given order according to their type. The precedence order of the three constraints is the following (starting with highest priority): pre, skip, cond. It is important to note that when a new type of constraint is introduced, its relative precedence has to be determined according to this list.

Composition of Constraints
Control constraints are by default conjoined; an action can be executed only if none of the constraints applied to it forbids its execution. For example, in a set of constraints, if there is a cond constraint that does not allow for execution, the action to which it applies cannot be executed. As an example, consider the following pair of constraints: cond(x,z); cond(y,z). Since both constraints are applied to z, in order to execute z both x and y have to be true. In fact, the above mentioned pair of constraints can be rewritten into the following one: cond(x $\wedge$ y, z). On the other hand, note that the execution of z can be skipped and marked as executed by an additional skip constraint, since the skip constraint has a higher precedence than the cond constraint.

If complex Boolean expressions are used in cond constraints, they are composed with AND logic as well. Consider the following example, where two cond constraints with different Boolean expressions are applied to the same action: cond(a $\vee$ b, z); cond(!c, z). These two constraints can be rewritten in the following manner: cond((a $\vee$ b) $\wedge$!c, z).

Multiple Skips
In case of multiple valid skip constraints with different substitution values, a run-time conflict occurs, since it is ambiguous which value should be used for substitution. As an example, consider the following pair of constraints: skip(x,z,True); skip(y,z, False). Since both x and y can result in true after their executions, it is not obvious if the return value of z should be substituted with true or false. Since this problem can be determined only after x and y have been executed, we indicate this by a runtime-conflict when it is necessary. Note that there can be simple conflict situations that can be determined statically as well. For instance, skip(x,z,True); skip(x,z, False) is a statically detectable conflict, since different substitution values are used with identical conditions.

Cascading Constraints
The sequential composition of actions through constraints can have cascading effects. For instance, consider cond(A, B); pre(B,C) as an example. If B is not executed it implies that C will not be executed either.

5.3.4 Hard and Soft Specifications

Both ordering and control constraints introduced previously represent 'hard' forms of specification. This means that the semantic of a constraint does not allow the execution of the *constrained action* if any action that is part of a *condition* is not present at the join point. That is, the semantics of constraints aim at ensuring the presence of an action; this may be important for the sake of safety and correctness.

However, a 'soft' form of specification is also beneficial from the perspective of evolvability and maintainability. Soft specifications can 'tolerate' the absence of an action; hence, they can handle situations where the action is referred to in the specification but not present in the system yet (or anymore). This feature can be important to provide open-ended specifications. In the following section we will show soft and hard converter functions that can be used within the scope of constraints to achieve soft specifications.

Soft Converter Functions
Soft converter functions can imitate that an absent action was actually present at a shared join point and executed with a given return value. We propose three types of converter functions, differing in the return value that they provide:

1. %(action) – returns void value, short notation: %action
2. %t(action) – returns Boolean true value
3. %f(action) – returns Boolean false value

Listing 5.11 *Soft converter functions*

Example usage:

cond(%t(x),z) – If action x is not present it will be interpreted as if it was executed and returned true value. In this way, the constraint will behave 'normally' and execute z even if action x is not present.

pre(%y, z) – If action y is not present, it will be interpreted as if it was executed and returned void value. In this way, the constraint will allow the execution of z even if action y is not present. If the soft converter function is not applied, the absence of y will not allow the execution of z based on the semantic of pre (see Table 5.7).

Note that the scope of a converter function is the constraint in which it was applied. This means that other constraints will not see this conversion and different constraints can apply different converter functions.

A language mapped to the constraint model can support open-ended specifications to provide default constraint specifications that are evolvable (i.e. less fragile to changes when aspects are added to, and removed from a project). For instance, every unit that corresponds to an action and takes part in a pre constraint can be 'soft-converted' with the % function, unless there is another specification that overrules this[13].

Hard Converter Function

We consider it important to provide safe specifications when it is needed. The hard converter function is default converter function used in constraint specifications:

- #(action), short notation: #action

Example usage:

pre(#y, z) – if action y is not present, the ordering constraint 'breaks down' and does not allow for the execution of z either. That is, pre(#y,z) equals pre(y,z).

5.3.5 Summary

Practically, each constraint language narrows down further the possible composition and execution of advices. As an overview of the whole constrain-

13.For instance, the declare precedence construct of AspectJ uses also open-endedness in its specification: the weaver does not 'complain' if an aspect referred by the specification is actually not woven.

ing process, the following summary describes the steps of the enforcement of constraints:

1. **Structural constraints** First, based on the structural constraint specifications, the weaver signals when an aspect requires or excludes the presence of other aspects.
2. **Ordering constraints** Based on the ordering constraint specifications, the weaver generates a possible order of the superimposition of advices (cf. actions) and weaves the advices based on this order.
3. **Control constraints** Based on the control constraint specifications, the weaver performs a sequential, conditional execution of advices depending on the execution of the related advices (in runtime).

5.4 Dependency Graphs and Algorithms

In this section we describe how a given set of control constraints applied to the same join point are enforced. This involves two steps: (1) generating a valid execution order (which may be done statically), and (2) managing execution according to the control constraints. We introduce the notion of *dependency graphs* as a representation of the set of constraints.

5.4.1 Dependency Graph

A dependency graph (Figure 5.4 illustrates an example) consists of nodes that represent actions, and directed edges that represent the constraints between two actions. Edges always point from the dependent node to the node on which it depends. Edges have labels to denote the type of the constraint. A dependency graph always has a root node, denoted by a dashed rounded rectangle. Dashed edges point to the root node, from all the actions that are allowed to execute first; i.e. actions that have no preceding actions. They represent a pre constraint that is assumed for these actions by default.

Figure 5.4 shows the dependency graph of Case I. In this case only one pre constraint has been specified between MonitorSalary and DBPersistence. Thus, there are three possible starting actions in Case I.

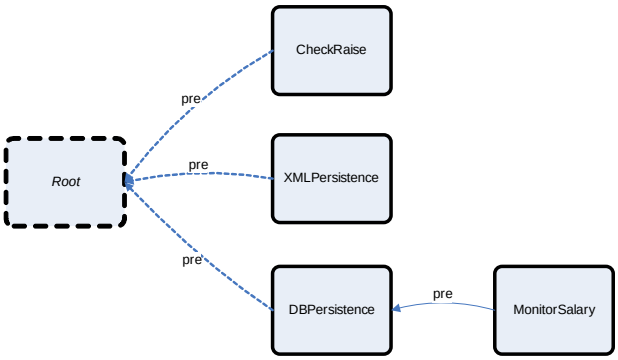

Figure 5.4 *Dependency graph of Case I.*

As another example, Figure 5.5 represents the dependency graph of four constraints that specify the composition of aspects as required in the section 5.2.2. There is only one possible starting action, CheckRaise. DBPersistence executes only if CheckRaise succeeds (denoted by the dotted arrow, and cond(C,D) between CheckRaise and DBPersistence). XMLPersistence will execute only if DBPersistence does not execute (or fails) *and* CheckRaise succeeds (denoted by the dotted arrow, and skip(D,X, void) between DBPersistence and XMLPersistence). In all the other cases XMLPersistence will not execute. There is a pre constraint between the two aspects, since it was required that MonitorSalary must not execute after DBPersistence. The order between MonitorSalary and XMLPersistence was not specified; hence, any of these two can execute first.

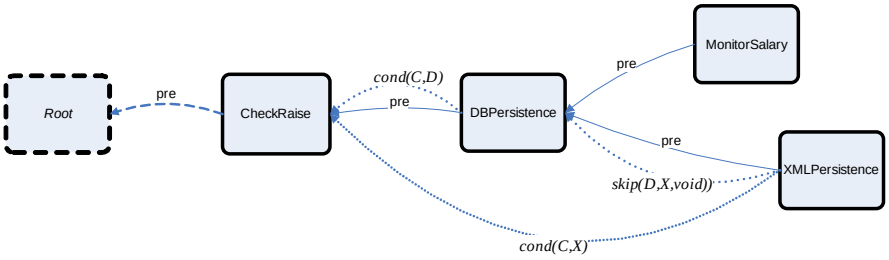

Figure 5.5 *The dependency graph of Case V.b*

5.4.2 Algorithm for Ordering Actions

Given a set of constraints between actions at a shared join point, one or more possible execution orders should be generated. This can be achieved by traversing the dependency graph, where all constraints have been mapped to *pre* constraints; all other semantics of constraints are dealt with by the execution-managing algorithm[14].

The algorithm that we show performs a topological sort [4] of nodes, including cycle detection. Instead of explaining the structure of the algorithm in detail, we use a simple example graph to show in an intuitive way how the traversal takes place and the order is generated. To this aim, Figure 5.6 demonstrates a traversal of a simple graph built up from the constraints pre(C,D), pre(C,X), pre(X,M) and pre(D,M). This a possible ordering of the scenario that we discussed in section 5.2.2 where C = CheckRaise, D=DBPersistence, X=XMLPersistence and M=Monitoring.

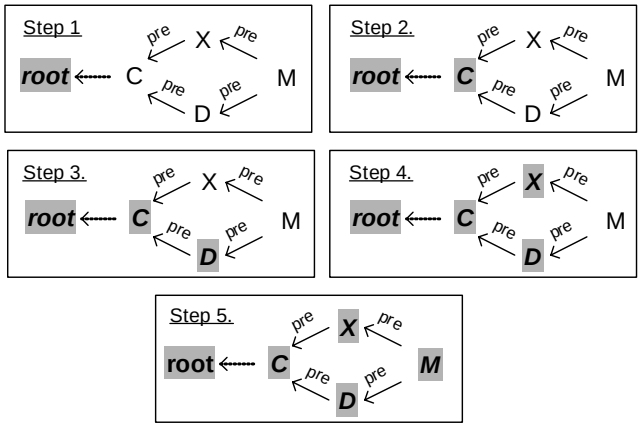

Figure 5.6 *An example for traversing a graph*

To generate an order, the traversal algorithm looks for a node for which all parents have been visited. (A node *p* is a parent to node *n* if there is an edge pointing from *n* to *p*.) In step 1, the root node is the only one that satisfies that condition. Thus, the root node is selected as the *current node*. When a node is successfully selected as the current one it is added to the end of the queue that represents the execution order. Hence, the root node will be the first item in this

14. Section 5.5 presents the details of the applied algorithms.

queue. In the second step, the node C is selected as the current node, since this is the only node that has only visited parents (the root node has been visited and placed in the execution queue). The node C will be added to the end of the queue. In the third step, there are two nodes, X and D, that have only visited parents. If more than one node meets the criteria for the selection there are more possible execution orders that are valid to the given constraints. In this case the algorithm randomly chooses one of them and gives a warning that no unique order can be determined. In this example we assume that D will be selected and added to the queue[15]. For the next step two candidate nodes remain. M cannot be selected, since one of its parents is still unvisited (i.e. X). In contrast, X has only one visited parent. Therefore, X will be added as the next item in the queue. In the last step, both parents of M have been visited, so M can also be selected and added to the queue.

Since there are now no more nodes to visit, the algorithm terminates. The traversal will also terminate if none of the remaining –unvisited– nodes is suitable for selection. In this case the graph has at least one cycle, which is caused by circular references among the constraints applied. The traversal algorithm detects this situation and can subsequently show all the cycles by listing the actions involved for each cycle.

5.4.3 Algorithm for Managing Execution

Once the execution order has been determined, the actions should be performed, but only if all relevant control constraints are taken into consideration. In order to show how this run-time process takes place we will use the Employee example: the ordering algorithm has returned the following order: CheckRaise, DBPersistence, XMLPersistence, MonitorSalary. Further, we assume that all the actions are assigned to the join point; that is, there is not any absent action. We recommend to see Figure 5.5 to follow the description of the execution.

By default, an action is *executable* if it is present at the join point. At the implementation level, the *executable* state indicates both whether an action can be executed and whether the constraints of the action can be enforced. The

15.Note that we would need an extra pre(D,X) constrain to provide a correct, non-deterministic specification for our scenario.

enforcement of a constraint or the execution of the action may change this state to *non-executable* or *executed*.

In the example of Figure 5.5, the execution starts with CheckRaise. There is no action that CheckRaise depends on, so it can be performed after checking if it is *executable*. We assume that CheckRaise executes successfully and returns true. DBPersistence is the next action in the queue and it depends on CheckRaise via a cond constraint. Before enforcing a constraint, first a check is performed to determine if the current action is executable. Since DBPersistence is executable, the cond constraint can be checked. This means that the return value of CheckRaise is evaluated and DBPersistence remains executable, since the value is true. We assume that DBPersistence executes successfully and returns true. XMLPersistence is the current action after the execution of DBPersistence. XMLPersistence has two constraints: a skip constraint with DBPersistence and cond with CheckRaise. The algorithm selects the constraint with higher precedence to evaluate first. In the example, first the skip constraint is selected because the skip constraint has a higher precedence than cond. The evaluation of the expression with DBPersistence results in true value. This implies that the execution of XMLPersistence will be skipped; it will be marked as *executed* with void return value. Another consequence is that the cond constraint will not be checked, since XMLPeristence is not executable anymore. Finally, MonitorSalary is the last item in the queue. Having no control constraint, this action will simply be executed.

In another scenario, assume that XMLPersistence is the current action to be executed again. Further, assume that CheckRaise failed and as a result of this, DBPersistence has not been executed. When the skip constraint is checked first, XMLPersistence still remains executable. However, when the cond constraint is checked, XMLPersistence will no longer be executable because CheckRaise failed.

This scenario shows the importance of the *precise* specification of constraints. Without the latter cond constraint, XMLPersistence would be executed in this scenario, which is not the expected behavior of the composition.

5.5 Algorithms

This section presents the pseudo-code of the algorithms that realize the enforcement of behavioral and structural constraints that we introduced in the previous sections. To describe these algorithms, we extended the notation of [4] in the following manner:

- We used object-oriented message calls on certain variables (e.g. *c.enforce()*). We applied this notation in the pseudo code where the execution of a method has no relevance to the description of the algorithms.
- We introduced the use of comments in pseudo-code.

5.5.1 Algorithm for Order Generation

Table 5.12 presents the input variables and declarations in the algorithms related to ordering:

Table 5.12 Input variables and declarations in the algorithms related to ordering

Variables, functions	*Description*
Nodes	set of the available nodes in the graph
root	the root node
Order	a list of nodes that contains the actions to be executed in their order
ENQUEUE(List, anElement)	function that adds an element to a FIFO list
DEQUEUE(List) :: anElement	function that removes and returns an element from a FIFO list
PARENT-NODES(node):: List	function that returns all parent nodes[1] of node

1. A node p is a parent to node n if there is an edge 'pre' pointing from n to p.

The pseudo-code of the algorithm that generates an order of actions is presented in Listing 5.13. As we wrote in section 5.4.2, this algorithm performs a topological sort of a directed graph that represents the ordering specification

of actions. The execution of the algorithm was demonstrated through an example case in section 5.4.2.

```
0)  GENERATE-ORDER(Nodes, root)
1)    current ← root
2)    while (true)
3)      do if current ≠ NIL
4)           then
5)             ENQUEUE(Order, current)
6)             Nodes ← Nodes \ {current}
7)           else
8)             if  Nodes ≠ ∅
9)               then
10)                return Order
11)              else
12)                print "Circular reference detected in the ordering specification"
13)                return NIL
14)       current ← SELECT-OPEN-NODE(Nodes, Order)
15)
16) SELECT-OPEN-NODE(Nodes, Order)
17)   Candidates ← {}
18)   for each current ∈ Nodes
19)     do if PARENT-NODES(current) \ Order = ∅
20)          then
21)            ENQUEUE(Candidates, current)
22)
23)   if Candidates = ∅
24)     then
25)       return NIL
26)     else
27)       /* The ordering is non-deterministic if |candidates| is greater than 1 */
28)       return DEQUEUE(Candidates)
```

Listing 5.13 *Order generating algorithm*

5.5.2 Interpreter Algorithm for Behavioral Constraints

Table 5.14 presents the input variables and declarations in the algorithms related to the enforcement of behavioral constraints:

Table 5.14 Input variables and declarations in the algorithms related to the execution of actions

Variables, functions	*Description*
OrderedNodes	list of nodes that contains the actions to be executed in their order
ConstraintTypes	list of constraint types in their precedence order
ENQUEUE(List, anElement)	function that adds an element to a FIFO list
DEQUEUE(List) :: anElement	function that removes and returns an element from a FIFO list
HEAD(List) :: anElement	returns an element from a FIFO list

Listing 5.15 presents the pseudo code of the algorithm that enforces the behavioral constraints pertaining to the actions of a shared join point. The execution of the algorithm was demonstrated through an example case in section 5.4.3. This algorithm is an interpreter that has the following generic steps to execute an action: first, the constraints of the action are ordered based on their precedence. Second, conflict detection rules, such as the multiple-skip rule, are performed to detect possible runtime conflicts among the constraints. After these two steps, the constraints are enforced one by one similarly to a shirt circuit (minimal) evaulation. Finally, the constrained action will be executed, if it is still executable after the enforcement of all constraints.

```
0)  EXECUTE-ACTIONS(OrderedNodes)
1)    /* the first node is always the root node and it does not contain any action */
2)    current ← DEQUEUE(OrderedNodes)
3)    while (OrderedNodes ≠ ∅ )
4)      do current ← HEAD(OrderedNodes)
5)         currentAction ← current.getElement()
6)         Constraints ← currentAction.getConstraints()
7)
8)         /* STEP 1: ordering the enforcement of constraints based
9)           * on their precedence*/
10)        Constraints ← ORDER-CONSTRAINTS(Constraints)
```

```
11)             /* STEP 2: detecting possible runtime conflicts, e.g. skip-skip */
12)             DETECT-CONFLICTS(current, Constraints, OrderedNodes)
13)
14)             /* STEP 3: enforcement of each constraint one by one */
15)             for each c ∈ Constraints
16)                 do if currentAction.isExecutable()
17)                         then
18)                             c.enforce()
19)                         else
20)                           break
21)
22)             /* STEP 4: execution of the action if it is still executable */
23)             if currentAction.isExecutable()
24)                then
25)                   currentAction.execute()
26)
27)             DEQUEUE(OrderedNodes)
28)
29)  ORDER-CONSTRAINTS(Constraints)
30)    /* ContraintTypes is a list of contraint types in their precedence order */
31)    for each ct ∈ ContraintTypes
32)       do for each c ∈ Contraints
33)            if c.type() = ct
34)               then
35)                  ENQUEUE(OrderedConstraints, c)
36)
37)  DETECT-CONFLICTS(current, Constraints, Nodes)
38)     /* Iterate over each type of constraints and detect possible conflicts
39)      * based on the current node, constraints and the actual set of nodes */
40)     for each ct ∈ ConstraintTypes
41)        do ct.detectConflict(current, Constraints, Nodes)
```

Listing 5.15 *Algorithm for managing the execution of actions*

5.5.3 Detecting Conflicts Among Structural Constraints

As we discussed in section 5.5.3, there are two types of conflicts that may occur in the specification of structural constraints. To detect these conflicts, we use a simple technique that uses a directed graph representation of structural constraints and an algorithm that searches for a path between two nodes of the graph.

In the graph representation of structural constraints, a constraint include(x,y) is represented by a directed edge between the nodes x and y. Figure 5.7 illustrates a simple example graph that is built up from the following constraints: include(a,b), include(b,c), include(b,d).

In this graph representation, a structural conflict is detected for a constraint exclude(x,y) if there is a path of include edges either from the node x to y (cf. straight conflict), or from the node y to x (cf. reverse conflict).

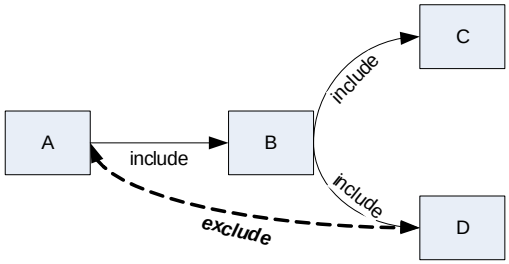

Figure 5.7 *An example graph representation of the structural constrains* include(a,b), include(b,c), include(b,d) *and* exclude(a,d)

For instance, a straight conflict will be detected for the constraint exclude(a,d), since there is a path between the nodes a and d in the example graph.

Listing 5.17 presents the pseudo code of the algorithm that performs the conflict detection in the specification of structural constraints. Table 5.16 presents the input variables and declarations of this algorithm.

Table 5.16 Input variables and declarations in the algorithms related to the conflict detection among structural constraints

Variables, functions	*Description*
Graph	a set of nodes and edges that represents the specification of structural constraints
ENQUEUE(List, anElement)	function that adds an element to a FIFO list
DEQUEUE(List) :: anElement	function that removes and return an element from a FIFO list

```
0)  DETECT-STRUCTURAL-CONFLICTS(Graph)
1)    for each edge ∈ Graph.getEdges()
2)       do if edge.getLable = "exclude"
3)             then
4)                left ← edge.getLeftNode()
5)                right ← edge.getRightNode()
6)
7)                /* detecting straight conflict */
8)                if FIND-INCLUDEPATH(left, right) = true
9)                   then
10)                     print "structural conflict for exclude(" +left+ "," +right+ ")"
11)
12)               /* detecting reverse conflict */
13)               if FIND-INCLUDEPATH(right, left) = true
14)                  then
15)                     print "structural conflict for exclude(" +right+ "," +left+ ")"
16)
17) FIND-INCLUDEPATH(source, target)
18)   OpenNodes ← {} ; Path ← {}
19)   current ← source
20)   while ((current ≠ NIL) AND (current ≠ target))
21)      do ENQUEUE(Path, current)
22)         current ← SELECT-OPEN-NODE(current, target, OpenNodes, Path)
23)   return (current ≠ NIL)
24)
25) SELECT-OPEN-NODE(current, target, OpenNodes, Path)
26)   /* Expand the set of open nodes with the children of the current node*/
27)   for each child ∈ CHILDREN-INCLUDE-NODES(current)
28)      do if ((child ∉ OpenNodes) AND (child ∉ Path))
29)            then
30)               ENQUEUE(OpenNodes, child)
31)
32)
33)  /* if the target node is in the set of open nodes
34)   * it is returned immediately to optimize the execution time */
35)   if target ∈ OpenNodes
36)      then
37)         return target
```

```
38)  /* if there are no open nodes left... */
39)  if OpenNodes = ∅
40)     then
41)        return NIL
42)     else
43)        return DEQUEUE(OpenNodes)
```

Listing 5.17 *Algorithm for detecting structural conflict*

5.6 Integration with AOP Languages

In this section, we will show the application of the concepts of the constraint model to concrete AOP languages. As we pointed out before, the constraint model is intended to be a succinct representation of the core concepts for controlling the interaction among aspects. Hence, it does not address programming language issues such as comprehensibility. It is rather intended as a model that can be adopted by AOP languages.

This section is structured as follows: first, we extend the join point concept, as it is available in most AOP languages. Then, we use the extended join point construct to integrate the constraint model with AspectJ and Compose*. We revisit the example that we introduced in the problem analysis section and show how the extended version of AspectJ and Compose* can resolve the identified problems.

5.6.1 Extending Join Points with Properties

Most AOP languages provide reflective information about the *current* join point by representing the join point as a first-class entity. The 'instance' of the join point can be accessed within the body of the *advice* that is being executed when the join point is reached. For example, in AspectJ, the type JoinPoint represents the concept of join point. The variable thisJoinPoint is an instance of that type and it can be used only within the context of *advices*. The type Joinpoint in AspectWerkz [1], Invocation in JBoss [5], and ReifiedMessage in Compose* [2] serve the same purpose.

To implement the conditional execution of aspects (i.e. cond constraint) and other concepts of the constraint model presented in section 5.3, we have extended the interface of type join point with new operations. These operations allow for placing and retrieving extra information into and from an instance of

the join point – this extra information may originally not pertain to the join point itself. In this way, the join point will act as a communication channel among the aspect instances that are sharing the same join point. Thus, aspect instances being executed on the same join point can exchange information among each other through the extended join point interface. In addition, the extra information placed in the join point can also be recognized and maintained by a weaver to direct the weaving process.

Extra Information: Properties

The extra information is represented in the form of properties. A property is a *key-value* pair that belongs to the join point during the execution of advices. The *key* is the fully qualified name of the property: a fully qualified representation where the property was created (the namespace and the name of an aspect and its advice), plus the identifier of the property itself. For example, the *value* is a fully qualified reference to a constant defined in Java. The structure of properties is defined as follows:

Definition

```
Property := <Key; Value>
Key := <Namespace.Aspect.Advice.Identifier>
Value := Fully Qualified Constant References in Java
```

Example

```
<Persistence.update.isSucceeded; Boolean.True>
```

Manipulation of Properties

In general, properties can be manipulated by two parties: the weaver and programmers. Before or after the execution of an advice the weaver can create, access, change or release a property related to the join point. We refer to the properties recognized by the weaver as *built-in* (*application independent*) properties. Programmers can also use built-in properties to direct the weaver. Built-in properties are independent of particular applications; typically, they are used by the weaver for maintaining standard interactions among aspects. We consider the conditional execution of aspects as an example of such an interaction. Programmers can also create their own properties and manipulate them within advices. We refer to the properties created by programmers as *user-defined* properties. User-defined properties are typically application specific properties. In this case, a user-defined property realizes a common parameter passing mechanism among aspects to exchange information.

5.6.2 Integration with AspectJ

Before we adapt the constraint model to AspectJ, we need to carry out two simple extensions to the language:

Named Advices

As we mentioned above, every property has a fully qualified name for two reasons: to be able to trace back to the origin (i.e. advice) of the property and to provide a unique name for the property. For this reason, the advice-construct of AspectJ needs to be extended with an identifier[16].

Extending the Join Point Interface

To be able to handle properties, the org.aspectj.lang.JoinPoint interface needs to be extended with the following methods:

void createProperty(String propertyId, Object value) throws PropertyExists – creates a property with the given value. If the property already exists, the method throws an exception.

Object getProperty(String propertyName) throws AmbiguousPropertyIdentifier – returns the value of the given property. The propertyName is either the fully qualified name, or only the identifier of the property. If the property with the given identifier or fully qualified name does not exist, the method returns a null value. When only the identifier is used as propertyName and there are more properties with the given identifier, the method looks up and returns the one that is in the default namespace. (That is, it looks up the property that is created in the current aspect & advice). If there is not such a property, the method throws an AmbiguousPropertyIdentifier exception.

void setProperty(String propertyName, Object value) throws AmbiguousPropertyIdentifier – sets the value of the given property. The look up strategy is the same as described at the method getProperty. If the given value is null the property is removed.

16.A number of AOP languages (e.g. AspectWerkz, JBoss, Compose*) already support the identifier of the construct that represents the superimposed behavior. (Typically, this construct is called advice in AOP). However, this does not apply to AspectJ, where advices are unnamed. To keep the backward compatibility of weaver, the name of the advice is an optional syntax element. However, properties can be created only within named advices.

Listing 5.18 illustrates the use of these extensions by a simple example. Within a named advice (checkRaise) a property (isSucceeded) is stored with a given value. The weaver will read this value whenever the advice checkRaise is used as a condition.

```
1) public aspect EnforceBusinessRules{
2)    after checkRaise(Employee p, int l):
3)       MonitorSalary.salaryChange(p,l){
4)          …
5)          thisJoinPoint.createProperty('isSucceeded',
6)            Boolean.True);
7)          …
8)    }}
```

Listing 5.18 *Placing a property into a join point in AspectJ*

Adopting the Constraint Model in AspectJ

Before discussing how AspectJ can adopt the constraint model, we have to mention that AspectJ has already introduced the declare precedence construct to order the execution of *advices* at shared join points. For this reason, we do not provide a mapping from AspectJ to the ordering constraints in our approach.[17] Consider the following significant characteristics of the constraint model:

Granularity of actions: Advices are mapped to the actions of the constraint model. A built-in property called isSucceeded is introduced to indicate the success or failure of an advice. This built-in property can be set by the above described operations, as shown in Listing 5.18. To enforce conditional constraints, such as cond, the weaver uses the property isSucceeded of each advice that is used in a *condition* of a control constraint. It is not mandatory for programmers to set isSucceeded in each advice. If isSucceeded is not set for an advice but the advice is used in a condition, the weaver takes the void case (neither success nor failure)[18] by default.

Specification of constraints: A new construct is introduced in AspectJ to define specifications of control constraints. This construct has the following syntax:

17.Even though AspectJ has class-level precedence rules.

18.It is important to note we write the Boolean property into the join point and do not modify the original return value of an advice.

declare constraint [on pointcut]: list of constraint statements. A set of *constraint statements* is introduced, which aim at providing the desired control constraints for a join point designated by *pointcut*. If the clause on pointcut is not used the contraint specification is global to every join point. We list the constraint statements along with their mapping to the constraint model:

Control constraints

(x and y represent advices)

x if y;	⇔ *cond(y, x);*
skip x with const if y;	⇔ skip(y, x, const);

Structural constraints

(x and y may represent both advices and sets of advices, see details below)

x includes y;	⇔ *include(x, y);*
x excludes y;	⇔ *exclude(x, y);*
x m_includes y;	⇔ *include(x, y); include(y, x);*

Designation of actions: In general, the arguments of the constraint statements (x and y) designate advices, which can be specified according to the template namespace.Aspect.advice. For structural constraints, the arguments can designate a set of possible advices, which means that the constraint statement is repeated over the elements in the Cartesian product of the argument(s). For example, the arguments of an include constraint statement can be resolved as follows:

{a1, a2} includes {a3, a4} ⇔ *include(a1, a3); include(a1, a4);*
include(a2, a3); include(a2, a4);

This is equivalent to four *include* constraints with each of the possible combinations of advices a1 to a4. In effect, this illustrates that the constraint statements can express crosscutting constraints.

Modularization of specifications: In AspectJ[19], the constraint specification, similarly to other declare constructs, is modularized by aspects. Note that it is not necessary to place a constraint specification in an aspect that is referred by

19.This is a proposal for an extension to AspectJ.

the specification itself; any aspect can contain arbitrary constraint specifications.

```
1) public aspect ApplicationConstraints{
2)  declare constraint on
3)   call(void Employee.increaseSalary(..)):
4)    DBPersistence.update if EnforceBusinessRules.checkRaise;
5) }
```

Listing 5.19 *An example constraint specification in (extended) AspectJ*

Listing 5.19 shows an example of a constraint specification. It specifies that the advice update of the aspect DBPersistence executes only if the advice checkRaise of the aspect EnforceBusinessRules has succeeded.

Example Revisited

In Listing 5.20, we revisit the second step of our scenario (section 5.2.2). In this listing, we show how the extended version of AspectJ can realize the composition of DBPersistence and CheckRaise, without introducing the problems we have identified in its original AspectJ version. In the aspect CheckRaise we have made three modifications: (line 2) the code that was responsible for resetting the Boolean variable has been removed; (line 4) the advice that is responsible for checking the salary has been named as checkRaise; (line 14-16) instead of the Boolean variable that was used for the workaround of conditional execution, the property ísSucceeded has been introduced to indicate the success or failure of checkRaise. The realization of DBPersistence (regarding the update functionality) has been modified in two places: (line 26) the advice that was responsible for updating the database has been named update; (line 27) the code that was responsible for the conditional execution has been removed. Naturally, it is necessary to express the conditional execution between DBPersistence and CheckRaise. This is done in the constraint specification at (line 36). As we wrote before, control constraints do not specify the execution order of advices; this also has to be provided to achieve the correct composition of aspects.

```
1)  public aspect CheckRaise pertarget(target(Employee) ){
2)    /* removed maintenance code */
3)
4)    after checkRaise(Employee person, int l):
5)      MonitorSalary.salaryChange(person,l){
6)       Manager m=person.getManager();
7)      if ((m!=null)&&(m.getSalary() <= person.getSalary()) ){
8)        …
9)        //Undo
10)      person.decreaseSalary(l);
11)       //setting Boolean for conditional execution
12)      /* _isValid = false; */
13)      thisJoinPoint.createProperty("isSucceeded",
14)         BooleanConstants.False);
15)     } else {
16)        thisJoinPoint.createProperty("isSucceeded",
17)           BooleanConstants.True);
18)     }
19)   }
20)}
21)
22)public aspect DBPersistence
23)   pertarget (target(PersistentObject)){
24)     …
25)    after update(PersistentObject po): stateChange(po){
26)    /* if (CheckRaise.aspectOf((Object)po).isValid()){ */
27)         System.out.println("Updating DB...");
28)         po.update(po.getConnection());
29)    /* } */
30)}}
31)
32)public aspect EmployeeConstraints{
33)     declare precedence:
34)        EnforceBusinessRules, DBPersistence;
35)     declare constraint on
36)        call(void Employee.increaseSalary(..)):
37)          DBPersistence.update if
38)            EnforceBusinessRules.checkRaise;
39)}
```

Listing 5.20 *Realization of the second requirement in our scenario using the extended version of AspectJ*

Note that we have removed all code that was related to the workaround of conditional execution. The remaining code now represents clearly the intended responsibilities of aspects, since the conditional execution is realized by the weaver and it is not tangled with the affected aspects. The interaction between the aspects is expressed in the form of a declarative specification, which is much closer to the design, as opposed to the tangled realization. Besides, the two aspects have become independent from each other, since they do not contain references to each other anymore. As a result, there is a low coupling between these aspects; they can be developed and maintained independently.

5.6.3 Integration with Compose*

The constraint model is generic in the sense that it can be adopted by different AOP languages. For example, we have also provided an integration of the constraint model with Compose* in a way that is similar to AspectJ: The join point type of Compose* (the ReifiedMessage class) has been extended to handle properties, and we introduced a pre-defined property (named isSucceeded) to map filtermodules to actions. The mapping has been realized using similar steps as we have discussed for AspectJ.

Named Filtermodules

In Compose*, filtermodules represent the language unit of the superimposed behavior. Since filtermodules are already named, we do not have to deal with this issue.

Join Point Interface

To be able to handle properties, we also need to extend the join point type of Compose*, similarly, as we did for AspectJ. In Compose*, the composestar.runtime.FLIRT.message.ReifiedMessage type is responsible for representing the join point. We need to implement the same methods that we implemented for AspectJ: createProperty, getProperty and setProperty. These methods can be used within *advice types* (ACT), as it is illustrated by an example in Listing 5.21. The CheckRaise filtermodule (line 2) represents the superimposed behaviour. In fact, this filtermodule realizes the salary-checking feature of the step 3 of the scenario in section 5.2.2. However, the check of salary is realized in a bit different way than it was done in AspectJ, in the orig-

inal scenario. Instead of checking the salary after the execution of increaseSalary, we do the check before the execution in an ACT of Compose*.

```
1) concern EnforceBusinessRules{
2)   filtermodule CheckRaise{
3)     internals
4)       CheckRaiseACT cact;
5)     inputfilters
6)       m: Meta = { [increaseSalary] cact.check }
7)   }
8)
9)   superimposition{
10)    selectors
11)      businessClasses = { C | isClassWithName(C,'Employee')
}
12)    filtermodules
13)      businessClasess <- CheckRaise;
14)    }
15) }
16)
17) public class CheckRaiseACT{
18)    public boolean isValidSalary(int level){…}
19)
20)    void check(ReifiedMessage msg){
21)       int salaryLvl = msg.getArgument(0);
22)
23)       if (!isValidSalary(salaryLvl)){
24)          /* indicating the failure of the aspect
25)           * & skipping the original method*/
26)          msg.createProperty("isSucceeded",
27)             Boolean.False);
28)          msg.reply();
29)       } else {
30)          /* indicating the success of the aspect */
31)          msg.createProperty("isSucceeded",
32)             Boolean.True);
33)          msg.resume();
34)       }
35)     }
36)   }
37)}
```

Listing 5.21 *Example revisited in Compose**

This different implementation strategy is due to the fact that currently there is no construct in Compose* that clearly corresponds to the after-advice of Aspect. However, the ACT construct of Compose* has a similar semantics as the around advice of AspectJ. By using ACT (cf. “around advice”), our example is still suitable to illustrate the use of properties and conditional execution of filtermodules. The CheckRaise filtermodule is superimposed on the Employee class in the superimposition specification (line 9-15). In the superimposition specification, the selector businessClasses is defined to designate the Employee class and then, in the filtermodules part, CheckRaise is bound to this selector. Filters, placed in the CheckRaise filtermodule after the keyword inputfilters in line 5, process the intercepted messages to the instances of the classes on which the filtermodule is superimposed (i.e. the class Employee in this case). Here, only one filter is defined: a Meta filter that will match on every message named increaseSalary. The meta filter reifies the message matched and passes it as a parameter in a call to the check method on an instance of CheckRaiseACT (lines 20-36). In the check method, large part of the code is the realization of the business logic, except lines 25-26 and 31-32. In these lines, by using the createProperty method, we place a property (named isSucceeded) into the reified message representing the actual join point. In this case, the fully qualified name of the property will be EnforceBusinessRules.CheckRaise.isSucceeded. This property is read by the weaver whenever the filtermodule CheckRaise is used as a condition.

The Approach

We discuss the most important characterizations of our approach:

Granularity of actions: Filtermodules are mapped to the actions of the constraint model, because they modularize the behaviour superimposed upon the join points. Similarly to what we did for AspectJ, a built-in property called isSucceeded is introduced to indicate the success or failure of a filtermodule. This built-in property can be set within an *advice type* (ACT) by the above described operations as it is already shown in Listing 5.21. In addition, it is not necessary to use the meta-filter & ACTs for the manipulation of properties. Properties can also be manipulated directly from a dedicated filtertype, the Property filter. To enforce conditional constraints, such as cond, the weaver uses the isSucceeded property for each filtermodule that is used in a *condition* of a control constraint. It is not mandatory for programmers to set isSucceeded in

each filtermodule. If isSucceeded is not set for a filtermodule but the filtermodule is used in a condition, the weaver takes the void case (neither success nor failure)[20] by default.

Specification of constraints: The keyword constraints is introduced in the superimposition specification to describe constraint specifications. This keyword is followed by one or more constraint statements that are bound to a selector in the following manner: selector <- constraint statement. The selector designates a set of structural join points as a local scope of the constraint statements. This allows for specifying different constraints between the same filtermodules that are superimposed on different join points. Compose* defines a set of constraint statements, which aim at offering the power of the composition constraints with an obvious intuitive meaning. We list the statements along with their mapping to the constraint model:

Ordering constraints

(x and y may represent both filtermodules and sets of filtermodules, see details below, at the subsection Designation of actions):

x before y;	⇔ *pre(x, y);*

Control constraints

(x and y represent filtermodules):

x if y;	⇔ *cond(y, x);*
x skipif y with const;	⇔ *skip(y, x, const);*
x ordIf y;	⇔ *pre(y, x); cond(y, x);*
x ordSkipif y with const;	⇔ *pre(y, x); skip(y, x, const);*

Structural constraints

(x and y may represent both filtermodules and sets of filtermodules, see details below, at the subsection Designation of actions):

x includes y;	⇔ *include(x, y);*
x excludes y;	⇔ *exclude(x, y);*
x m_includes y;	⇔ *include(x, y); include(y, x);*

Control constraints do not deal with ordering; although, the filtermodules used by control constraints typically need to be ordered as well. For this reason, we have defined two new control constraint mappings: ordIf and ordSkip. These

20.Note that we do not talk about the return value of an advice here. The original return value of an around advice is not affected; we place the property into the join point instance.

constraint statements are for the sake of convenient use. By applying these constraints, for each filtermodule that appears in the condition, an ordering constraint is automatically created between the constrained filtermodule and the filtermodules that are used as conditions.

Designation of actions: In general, the arguments of the constraint statements (x and y) designate filtermodules, which can be specified according to the Compose* notation: namespace.Concern.filtermodule. For structural and ordering constraints, the arguments can also designate a set of possible filtermodules, which means that the constraint statement is repeated over the elements in the Cartesian product of the argument(s). For example, assume that the arguments of an include constraint statement can be resolved as follows:

{a1, a2} includes {a3, a4} ⇔ *include(a1, a3); include(a1, a4);*
include(a2, a3); include(a2, a4);

As we showed in the mapping to AspectJ, this is equivalent to four *include* constraints with each of the possible combinations of advices a1 to a4.

Modularization of specifications: In Compose*, the constraint specification is modularized by concerns. Note that it is not necessary to place a constraint specification in a concern that the specification itself refers to; any arbitrary concern can contain arbitrary constraint specifications. Listing 5.22 shows an example constraint specification: an ordIf relationship is defined between the filtermodules Update of DBPersistence and CheckRaise of EnforceBusinessRules. This means that both the *cond* and *pre* constraints are applied between these filtermodules when they are superimposed on class Employee (designated by the selector EnforceBusinessRules.businessClasses).

```
1) /* a separate concern definition for
2)  * application-specific constraints */
3) concern ApplicationConstraints{
4)    superimposition{
5)       constraints
6)          EnforceBusinessRules.businessClasses <-
7)            DBPersistence.Update ordIf
8)               EnforceBusinessRules.CheckRaise;
9) }}
```

Listing 5.22 *An example constraint specification in Compose**

In Listing 5.23, we revisit the third step of our scenario (section 5.2): we show, focusing on the use of composition constraints, how Compose* can realize the update functionality of DBPersistence and the salary checking functionality, implemented by the CheckRaise aspect in Listing 5.21. In our solution, the salary checking functionality is realized by the CheckRaise filtermodule of EnforceBusinessRules, and the CheckRaiseACT advice type. The implementation of these units has already been presented in detail in Listing 5.21. In Listing 5.23, the update functionality of DBPersistence aspect is implemented in a similar way (line 2): the Update filtermodule uses a meta filter to intercept the increaseSalary message, then it reifies the message and passes it as a parameter to the method update of DBPersistenceACT (line 18). Both the CheckRaise and Update filtermodules are superimposed on Employee; hence, they will do filtering on the same join point, when an instance of Employee receives the increaseSalary message. The conditional execution is defined in an independent concern named ApplicationConstraints (in Listing 5.22): the ordIf relationship defines two constraints between the Update and CheckRaise filtermodules: (a) the superimposition order of the Update and CheckRaise filtermodules is such that the CheckRaise filtermodule will process first the intercepted message and then, the Update filtermodule; (b) the Update filtermodule can be executed only if CheckRaise succeeded. This means that filtermodule Update will be executed only if isSucceeded has been set to true in CheckRaise.

Note that we achieved the same characteristics of code that we got in the revisited AspectJ examples: DBPersistence and EnforceBusinessRules can be formulated and maintained independently from each other; besides, the interaction (i.e. conditional execution) between them is formulated in an independent module, in the form of declarative specification.

5.7 Implementation

Figure C.1 in Appendix C presents the architecture of Compose*, the realization of Composition Filters on .NET platform. The constraint model and proposed language mechanisms were implemented within the Filter Composi-

tion and Checking (FILTH) module. The grammar of the proposed constraint language can be found in section A.11 in Appendix A.

```
1) concern DBPersistence{
2)   filtermodule Update{
3)      internals
4)         DBPersistenceACT db_act;
5)      inputfilters
6)         m: Meta = { [increaseSalary] db_act.update }
7)   }
8)
9)   superimposition{
10)    selectors
11)       businessClasses = { C |
12)          isClassWithName(C,'Employee')};
13)    filtermodules
14)       businessClasess <- Update;
15)   }
16)}
17)
18)public class DBPersistenceACT{
19)  void update(ReifiedMessage msg){
20)    /* let's fire the message */
21)    msg.proceed();
22)    PersistentObject po =(PersistentObject)msg.getTarget();
23)
24)    /* code copied from the after advice */
25)    System.out.println("Updating DBMS...");
26)    po.update();
27)    …
28)  }
29)}
30)
31)
```

Listing 5.23 *An overview of the implementation of the running example with Compose**

5.8 Related Work

Composition of aspects at shared join points is a common problem, which has been –partially– addressed by several AOP languages. In the following, we

examine some of them with respect to the requirements that we identified in section 5.2.

In AspectJ [8], the order between advices can be controlled by the declare precedence statement. The precedence determines the execution order of advices superimposed on the same join point, depending on the type of the advice. The precedence declaration can be placed either in the aspect that defines the advice, or in other independent aspects; this allows most of the modularizations discussed in section 5.2.3. Circular relationships among aspects are detected only if they are superimposed on the same join point. The precedence is defined at the level of aspects, which implies that different pairs of advice of the same two aspects cannot have different precedence. As in most other AOP languages, in AspectJ, conditional executions are not supported. However, to the best of our knowledge, among the current AOP languages, AspectJ is the one that supports the identified software engineering requirements to the largest extent.

Constantinides et. al. [3], emphasizes the importance of the 'activation order' of aspects that have been superimposed on the same join point. In their framework, called Aspect Moderator Framework, they propose a dedicated class, called *moderator*, to manage the execution of aspects at shared join points. The moderator class, as defined in [3], can express conditional execution of aspects, but it cannot specify partial ordering relationships between aspects. The implementation of the moderator class allows the activation of an aspect only if all the preceding aspects are pre-activated successfully. In our work, a conditional execution is defined between individual advice actions. In this way, the execution of an aspect does not depend on the order of other aspects. Note that since the application programmer can implement new moderator classes, it is possible to introduce other activation strategies; however, for certain cases, to define these strategies might not be straightforward in an imperative way as defined in Java. With the composition constraints we propose, the execution strategies are derived in a declarative way. Besides, extending the framework to support partial ordering relationships would allow for a more sophisticated way of the activation of aspects.

In JAC [11], wrappers are responsible for the manipulation of intercepted methods. A wrapper is implemented by a class that extends the class Wrapper.

The order of the wrappers that can be loaded into the system is handled in a global configuration file. In this file the wrapper classes are listed in their wrapping order. This means in JAC the wrapping order is global, whereas in our approach the order can be made specific to individual join points. JAC does not support the conditional execution between advices either.

EAOP [6] defines several operators that are comparable to our constraints. The *Seq* operator specifies an exact order of aspects. Unlike pre in our model, it does not allow for partial ordering. The EAOP operators Cond and Fst, are related to the Cond constraint of our model. However, in EAOP the composition *operators* are used *to construct* a composition of aspects, whereas in our model we use the *constraints to derive* a possible composition of aspects. The difference between the two approaches is that EAOP may require the reconstruction of the composition of aspect instances whenever a new aspect instance has to be included. In our model, by adding one or more new constraints, the composition of the new aspect is automatically derived. Further, in EAOP the specification of composition is not open-ended (it requires concrete aspect instances) and conflict analysis is not available, yet planned to be integrated in the tool.

The *connector* abstraction of JAsCo [13] allows for specifying the execution order of advices belonging to different hooks. In other words, the ordering specification is expressed on the level advices. Besides the ordering specifications, JAsCo allows for defining custom combination strategies using regular Java. Each concrete combination strategy implements the CombinationStrategy interface. A JAsCo combination strategy works like a filter on the set of applicable hooks at a certain point in the execution of the application. In JAsCo, it is possible to add/remove connector combination strategies dynamically. There are two important differences between the approaches of JAsCo and our constraint model: (1) combination strategies describe the composition of hooks in an imperative manner, in terms of Java code; (2) as opposed to our constraint model, the order of the application of combination strategies does matter in JAsCo. This means that the introduction of new hooks and combination strategies may require more effort to change the composition specification than using constraints. In addition, the imperative approach may render difficulties in checking the consistency of the composition specification, as compared to our approach.

5.9 Trade-off Analysis of Software Quality Factors

5.9.1 Comprehensibility

Using composition constraints, the dependencies between aspects are expressed explicitly; thus, the composition of aspects can be understood easily. Without constraints, these dependencies must be hard-wired into the body of aspects (and/or advices), which renders more difficulties in the understanding of the program.

It is also important to consider the modularization of composition constraints. If the specification of constraints is distributed over several modules (i.e. it is too fragmented), the comprehensibility of the program can decrease. On the other hand, if all constraints are centralized without sufficient organizational structure, this may scale up badly in terms of comprehensibility.

Briefly, composition constraints may have both positive and negative impacts on the comprehensibility of the composition of aspects, depending on their usage. For this reason, a language should support alternative modularizations (see section 5.2.3) of constraints. For example the AspectJ declare precedence construct allows this. Our model can be mapped to languages that support this.

5.9.2 Evolvability

A constraint specification - using the soft converter functions - may refer to aspects which are not necessarily present in the system. Whenever those aspects are defined by the developer and become present, the constraint specification will automatically apply to them. Thus, aspects can refer to other aspects which will be integrated with the system later. Furthermore, aspects can be removed from the system without modifying the constraint specification. For this reason, we call these specifications open-ended. This is an important property of our model, since aspects can be developed and deployed independently from each other.

5.9.3 Predictability

Without using ordering constraints, the execution order of advices is undetermined. For this reason, the application of control contraints have, in general, positive impact on predictability. However, the constraint skip may have nega-

tive impacts on runtime predictability: the possible conflicts can be detected among multiple skip constraints during compilation; however, the conflict itself can be detected only in runtime as we discussed in section 5.3.3.

Control constraints may have a slight negative impact on predictability when they are used with soft-converter functions. As we said in the previous section, aspects can refer to other aspects that can be integrated with the system later. Thus, a programmer can add a new aspect to the system without knowing what specifications apply to that aspect. Compiler and IDE support might help in preventing unintended compositions and avoiding conflicts.

On the other hand, using composition constraints, designers can ensure that when certain aspects are added later to the system, they will be integrated in a predefined way. This has a positive impact on the predictability of development process.

We introduced the notion of structural constraints to describe valid composition of aspects at shared join points. Besides, we are capable of detecting conflicts in the specification of both structural and control constraints. All of these features increases the predictability of the program specification.

5.9.4 Adaptability

A constraint specification can be placed in any aspect module of a system. A benefit of this approach is that control constraints statements in this manner can express application specific composition of independently developed aspects. This contributes in a positive manner to the adaptability the composition specification.

5.9.5 Modularity

Without explicit means to express the conditional execution between aspects, the programmer needs to use workarounds - in the form of variables and extra advices - that are not part of the original aspect specification. In addition, as we wrote in the previous section, aspects have explicit dependencies among each other that render difficulties in their reuse. Another benefit of applying control constraints and using the third modularization alternative of section 5.2.3 is that aspects contain only their intended responsibilities; hence, they have no

more direct references to each other. As a result, there is a *low coupling* between these aspects; they can be developed and maintained independently.

It is important to note that the reuse of aspects, e.g. through inheritance, should incorporate the reuse of constraints. This is a language design issue that is independent from our proposed model.

5.10 Conclusion

Shared join points are not a new phenomena, nor specific to any AOP language. To the best of our knowledge, SJP composition has not been explicitly analysed in-depth in the literature before. In particular, in the current approaches, we have encountered mostly ordering constraints, but little or no control constraints and structural constraints. In this chapter, we have first performed an extensive analysis on the issues that arise when multiple aspects are superimposed at a SJP. Based on this analysis, we identified a set of requirements that drove our design (section 5.2). As a generic solution, independent of any specific AOP language, we have proposed a constraint-based, declarative approach to specify the composition of aspects (section 5.3).

The proposed constraint specification can express the composition of aspects from different libraries, provided by third parties. This is important for large-scale systems, where a large number of aspects are involved in the development process. Unlike the other approaches, the composition is expressed in form of declarative specifications, rather than in some form of imperative code within methods. This declarative specification allows defining the composition of aspects already in the design phase.

We have implemented and tested the algorithms that are necessary to check the soundness of the constraint specification and detect possible runtime conflicts. By the underlying constraint model and conflict detection techniques we aimed at providing safe use for programmers.

We have proposed a mechanism to extend the concept of *join point* with the *property* construct. By this construct, aspects can exchange information among each other and control the weaver at shared join points. We claim that this extension is applicable to a wide range of aspect-oriented programming languages that offer an explicit join point type. Finally, to give an intuitive use

of the constraint model, we have provided two adaptations of it to two specific AOP languages, AspectJ and Compose* by using the extended join point type and a dedicated property.

5.11 References

[1] BONÉR, J. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)* (Mar. 2004), K. Lieberherr, Ed., ACM Press, pp. 5–6.

[2] Compose* project. http://composestar.sf.net.

[3] CONSTANTINIDES, C., BADER, A., AND ELRAD, T. An aspect-oriented design framework for concurrent systems. In *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)* (June 1999), C. V. Lopes, A. Black, L. Kendall, and L. Bergmans, Eds.

[4] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms (MIT Electrical Engineering and Computer Science)*. The MIT Press, 1990.

[5] DALAGER, C., JORSAL, S., AND SORT, E. Aspect oriented programming in JBoss 4. Master's thesis, IT University of Copenhagen, Feb. 2004.

[6] DOUENCE, R., AND SÜDHOLT, M. A model and a tool for event-based aspect-oriented programming (EAOP). Tech. Rep. 02/11/INFO, Ecole des Mines de Nantes, 2002.

[7] ELRAD, T., AKSIT, M., KICZALES, G., LIEBERHERR, K., AND OSSHER, H. Discussing aspects of AOP. *Comm. ACM 44*, 10 (Oct. 2001), 33–38.

[8] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *Proc. ECOOP 2001, LNCS 2072* (Berlin, June 2001), J. L. Knudsen, Ed., Springer-Verlag, pp. 327–353.

[9] NAGY, I., BERGMANS, L., AND AKSIT, M. Declarative aspect composition. In *2nd Software-Engineering Properties of Languages and Aspect Technologies Workshop* (Mar. 2004), L. Bergmans, K. Gybels, P. Tarr, and E. Ernst, Eds.

[10] NAGY, I., BERGMANS, L., AND AKSIT, M. Composing aspects at shared join points. In *Proceedings of International Conference NetObjectDays, NODe2005* (Erfurt, Germany, Sep 2005), A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, Eds., vol. P-69 of *Lecture Notes in Informatics*, Gesellschaft für Informatik (GI).

[11] PAWLAK, R. *AOSD with JAC*. PhD thesis, CNAM Paris, Dec. 2002.

[12] RASHID, A., AND CHITCHYAN, R. Persistence as an aspect. In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (Mar. 2003), M. Aksit, Ed., ACM Press, pp. 120–129.

[13] SUVÉE, D., AND VANDERPERREN, W. JAsCo: An aspect-oriented approach tailored for component based software development. In *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)* (Mar. 2003), M. Aksit, Ed., ACM Press.

Chapter 6

Conclusions

This chapter presents the contributions and conclusions of the research presented in this thesis, and suggests directions for future work.

Chapter 6 is structured as follows: section 6.1 emphasizes the contributions of this thesis. In section 6.2, we provide a summary about the language constructs that we introduced in this thesis. In this section, we also discuss how the introduced constructs are related to the language concepts of the reference model proposed in Chapter 2. Finally, section 6.3 indicates directions for future work.

6.1 Contributions

The contributions of the thesis are the following:

1. A reference model of aspect-oriented languages

Chapter 2 presents a reference model of aspect-oriented languages that captures their common and distinctive concepts. This reference model provides an overview of the state-of-the-art aspect-oriented languages and serves as a comparison framework to show their characteristic features. Furthermore, it exposes various design dimensions of aspect-oriented languages; i.e. those issues that have to be considered when one develops an aspect-oriented language.

2. An in-depth analysis of the role of design information within the context of aspect-oriented programming, and the integration of design information with aspect-oriented composition abstractions

Chapter 3 presents an in-depth analysis of the role of design information within the context of aspect-oriented programming. We motivate the need of referring to program elements based on their design information. We also demonstrated that the integration of design information with aspect-oriented composition mechanisms ('semantic composition') offers a means of coupling that is both manageable and powerful. The main benefits of the design information annotations that we introduce in chapter 3 are:

- The ability to select join points based on design information (which cannot be derived from the program itself).
- The proposed mechanism allows for choosing the appropriate locations to define annotations: within the code, co-located with the code or in an aspect.
- The use of design information makes aspects, especially pointcut expressions, less vulnerable to changes of the program. The reason is that they avoid dependencies of the pointcut expression upon the structure or the naming conventions of the program.
- The dependencies between program elements can be more precise and easier to understand by referring to the design intentions instead of structural, or syntactic patterns.

3. Evolvable advice-pointcut binding, and aspect specifications, founded on annotations and a generic, predicate-based query language

Chapter 4 presents an extensive analysis of the pointcut-advice binding mechanisms of various aspect-oriented languages. Based on the identified key properties (e.g. *loose coupling*, *many-to-many binding*), we designed an advice-pointcut binding concept that provides *associate access* to advices/aspects. In the new specification we applied queries, founded on a generic, predicate-based language, that allow for designating both the places - in terms of structural join points - where we want to weave, and the units (e.g. filter modules, methods, etc.) that we want to weave. As a result, we provided a binding mechanism that is more expressive and evolvable compared to the analysed binding mechanisms. By applying annotations (semantic properties), we improved the evolvability and comprehensibility of the advice-pointcut bindings, and aspect specification to a greater extent.

4. A constraint-based, declarative approach to specify the composition of aspects at shared join points

Chapter 5 presents an analysis on the issues that arise when multiple aspects are superimposed at the same join point. Based on this analysis, we identified a set of requirements towards expressing the composition of aspects. As a generic solution, independent of any specific AOP language, we proposed a constraint based, declarative approach to specify the composition of aspects at shared join points. The proposed constraint specification can express the composition of aspects from different libraries, provided by third parties. This is important for large scale-systems, where many aspects are involved in the development process. The composition is expressed in the form of declarative specifications, rather than imperative code within methods. We implemented and tested algorithms that are necessary to check the soundness of the constraint specification and detect possible runtime conflicts. The underlying constraint model and conflict detection techniques aim at providing a predictable specification for programmers.

5. Extending join points with the property construct to provide a mechanism by which aspects can exchange information with each other, and control the weaver at shared join points

In the second part of Chapter 5, we extended the concept of join point type with the *property* construct to provide a mechanism by which aspects can exchange information with each other, and control the weaver at shared join points. This extension is applicable to a wide range of aspect-oriented programming languages that offer an explicit join point metadata type.

By using the extended join point type and a dedicated property, we integrated the previously proposed constraint-based approach in the AOP languages AspectJ and Compose*.

6.2 Overview of the Introduced Language Constructs

This thesis evaluates the software composition mechanisms of current aspect-oriented languages, and proposes novel extensions to them so that programs written in these languages exhibit better quality.

In Chapter 3, we propose an expressive pointcut language founded on a generic, predicate-based language that can *designate structural join points* based on annotations in Compose*, which is indicated by (1) in Figure 6.1. In addition, we extend the matching expression of Compose* with the ability of referring to annotations. In this way, messages, as *behavioral join points,* can also be designated based on annotations attached to the corresponding shadow points.

In Chapter 3 and 4, we propose a language construct for the *introduction of annotations* in Compose*. To designate structural join points for the introductions, we apply the fully expressive pointcut language of Compose*. As a result, annotations can be introduced to program elements based on the existence of other annotations. We call this technique derivation of annotations. By supporting the derivation of annotations, dependent annotations (and complete annotation hierarchies) can be automatically superimposed.

In Chapter 4, we propose a new advice-pointcut binding specification construct, indicated by (2) in Figure 6.1. This construct is a specialization of the *separate binding specification* concept of the reference model. In the new binding construct, we apply queries to indirectly designate filtermodules (cf. *aspects*) for superimposition, based on their properties. The new binding specification uniformly presents the superimposition of various subjects, such as filtermodules and annotations; hence, we call this feature weaving subject polymorphism.

In Chapter 5, we propose a constraint specification language to specify the composition of advices at shared join points. Its ordering constraints correspond to the *advice ordering* concept of the reference model, indicated by (5) in Figure 6.1. Its behavioral and structural constraints, indicated by (6) and (7) in Figure 6.1, correspond to the *customizable advice composition* and respectively, *aspect-aspect composition* concepts of the reference model.

We also propose to extend the concept of *join point type* with properties, indicated by (3) in Figure 6.1. Properties provide a parameter passing mechanism between aspects: properties can be written into a join point instance and read by aspect instances to exchange information among them.

Figure 6.1 illustrates the relationships between the proposed language constructs and the concepts of the reference model

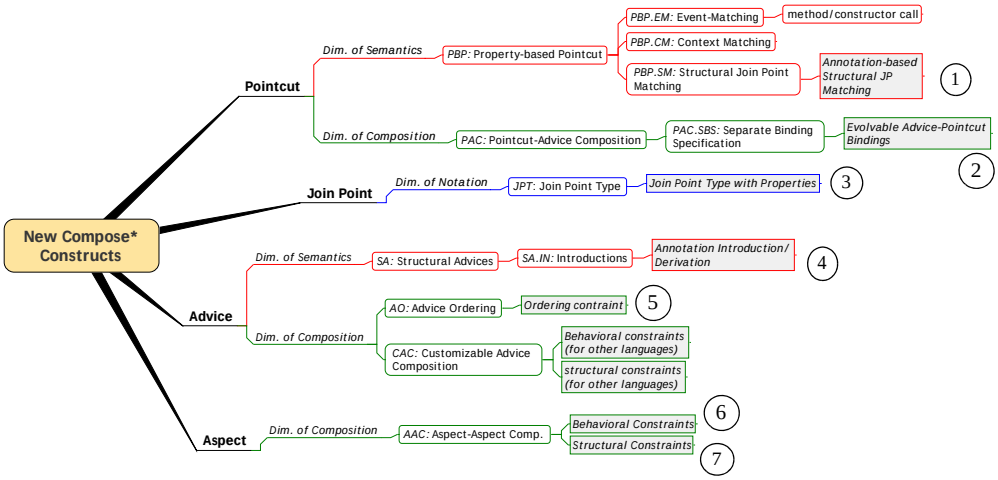

Figure 6.1 *The relationships between the proposed language constructs and the concepts of the reference model*

6.3 Future Research

As we discussed in Chapter 2, a construct of a particular aspect-oriented language may fulfil the role of more than one language concepts of the reference model and vice versa. That is, there is often no one-to-one mapping between the constructs of a language and the reference model. One of our future works is to provide mappings to the reference model from various aspect-oriented languages, besides the ones that we provided in Chapter 2. The motivation behind this work is to make observations on (1) how a concept of the reference model is realized through the constructs of a particular language, and (2) how a construct of a particular language can realize multiple concepts of the reference model. This information could help us to reflect on the design of aspect-oriented languages, and understand how certain language constructs may influence positively or negatively the quality of aspect-oriented programming languages.

Regarding the use of annotations, disciplined programming is required to keep the correct design information associated with the appropriate program elements. We believe this problem can be improved through the language

constructs that we proposed in Chapter 4. However, it is unavoidable that certain semantic properties have to be specified by the software engineer. We have illustrated how superimposition and derivation can be used to attach semantic properties. In addition, we plan two ways to address this issue: (1) by investigating design-level support and the automatic derivation of annotation specifications from stereotypes in UML diagrams; (2) by searching for techniques (e.g. control and data flow analysis) that can automatically derive certain common annotations.

Other potential future work is about (a) the ability to apply the notion of semantic composition to more composition techniques than the superimposition mechanism (that was the main subject of study of Chapter 2), or (b) the exploitation of semantic composition for the purpose of modelling product lines and variability management.

In Chapter 5, we presented a composition model for composing aspects at shared join points. One of our future works is to investigate interactions between aspects that are not necessarily woven at the same join points, and to design models for their composition. To realize certain interactions between aspects, we extended the concept of join points types with the property construct. We are interested in the further application possibilities of the property construct at shared join points to implement more complex interactions, and to detect possible interaction patterns between aspects.

Appendix A

The Grammar of Compose*

This appendix describes the grammar of Compose*. Standard EBNF ISO/IEC 14977 is used to specify the grammar. Each rule of the grammar defines a non-terminal symbol. The defining symbol in a rule is "::=". In the right hand-side of a rule, the following elements of the EBNF can be used:

- [ ] : specifies an elements that is optional
- ( )* : specifies an element that can be repeated zero or more times
- ( )+ : specifies an element that can be repeated one or more times
- ’ ’ : specifies a string literal
- a | b : specifies an alternative to a rule
- a-LIST: this expression is always substituted with a (’,’ a)*

A.1 Concern Definition

```
Concern                 ::= 'concern' ConcernName
                            ['(' FormalParameters ')']
                            ['in' PackageReference] '{'
                              (FilterModule)*
                              [SuperImposition]
                              [Implementation]
                            '}'

FormalParameters        ::= FormalParameterDefinition-SEQ
```

FormalParameterDefinition ::= Identifier-LIST ':' Type

A.2 Filtermodule Definition

FilterModule ::= '**filtermodule**' FilterModuleName '{'
[Internals]
[Externals]
[Conditions]
[MethodDeclarations]
[InputFilters]
[OutputFilters]
'}'

Internals ::= '**internals**' (Identifier-LIST ':' Type ';')*

Externals ::= '**externals**' (Identifier-LIST ':' Type
['=' InitializationExpression] ';')*

InitializationExpression ::= FilterModuleElementReference

Conditions ::= '**conditions**' (ConditionDecl)*

ConditionDecl ::= ConditionName ':' ConditionReference ';'

MethodDeclarations ::= '**methods**' MethodDeclaration*

MethodDeclaration ::= MethodName '('
[FormalParameterTypeDef-SEQ] ')'
[':' ReturnType] ';'

FormalParameterTypeDef ::= [Identifier-LIST ':'] Type

InputFilters ::= '**inputfilters**' GeneralFilterSet

OutputFilters ::= '**outputfilters**' GeneralFilterSet

A.3 Filter Definitions

```
GeneralFilterSet            ::= GeneralFilter (
                                FilterCompositionOperator
                                GeneralFilter)*

FilterCompositionOperator   ::=  ';'

GeneralFilter               ::= FilterName ':' FilterType
                                ['(' ActualFilterParameters ')'] '='
                                '{' [FilterElements] '}'

ActualFilterParameters      ::= Value-LIST

FilterElements              ::= FilterElement
                                (ElemCompositionOperator
                                FilterElement)*

ElemCompositionOperator     ::= ','

FilterElement               ::= [ConditionExpression
                                ConditionOperator]
                                MessagePatternSet

ConditionOperator           ::= '=>' | '~>'

ConditionExpression         ::= ConditionLiteral | '!' ConditionLiteral |
                                '(' ConditionExpression ('|' | '&')
                                ConditionExpression ')'

ConditionLiteral            ::= ConditionName| 'True' | 'False'

MessagePatternSet           ::= '{' MessagePattern-LIST '}' |
                                MessagePattern

MessagePattern              ::= SignatureMatching SubstitutionPart |
                                NameMatching SubstitutionPart |
                                DefaultMatchAndSubstitute

SignatureMatching           ::= '<' MatchPattern '>'
```

NameMatching ::= '[' MatchPattern ']' |
Quote MatchPattern Quote

DefaultMatchAndSubstitute ::= MatchPattern

SubstitutionPart ::= MatchPattern

MatchPattern ::= [Target '.'] Selector

Target ::= Identifier | 'inner' | '*'

Selector ::= MethodName ['(' [Type-SEQ] ')'] | '*'

A.4 Superimposition

SuperImposition ::= '**superimposition**' '{' [SelectorDefinition]
[ConditionBinding]
[MethodBinding]
[FilterModuleBinding]
[AnnotationBinding]
[Constraints]
'}'

A.5 Selector definitions

SelectorDefinition ::= '**selectors**' (SelectorName '=' '{'
VarName '|'
PrologBody '};')*

VarName ::= UpperCase (LowerCase)*

PrologBody ::= PrologFun-LIST

PrologFun ::= ConstString ['(' [Arg-LIST] ')']

Arg ::= PrologFun | PrologVar | PrologList |
ConstNum

PrologVar ::= '_' | VarName

PrologList ::= '[' ']' | '[' ListElems ']'

ListElems ::= [Arg-LIST] ['|' (PrologList | PrologVar)]

A.6 Common Binding Information

CommonBindingPart ::= Selector WeaveOperation

Selector ::= ConcernElementReference

WeaveOperation ::= '<-'

A.7 Condition Bindings

ConditionBinding ::= '**conditions**' (CommonBindingPart ConditionNameSet ';')*

ConditionNameSet ::= '{' ConditionName-LIST '}' | ConditionName-LIST

ConditionName ::= FilterModuleElementReference | FilterModuleElementReferenceStar

A.8 Method Bindings

MethodBinding ::= '**methods**' (CommonBindingPart MethodNameSet ';')*

MethodNameSet ::= '{' MethodName-LIST '}' | MethodName-LIST

MethodName ::= FilterModuleElementReference ['(' [Type-LIST] ')'] | FilterModuleElementReferenceStar

A.9 Filtermodule Bindings

FilterModuleBinding ::= '**filtermodules**' (CommonBindingPart FilterModuleSet ';')*

FilterModuleSet ::= '{' ConcernElementReference-LIST '}' | ConcernElementReference-LIST

A.10 Annotation Bindings

AnnotationBinding ::= '**annotations**' SelectorRef '->' AnnotationSet ';'

AnnotationSet ::= concernReference-LIST | '{' concernReference-LIST '}'

A.11 Constraints

Constraints ::= '**constraints**' (CommonBindingPart ConstraintElementSet ';')*

ConstraintStatementSet ::= '{' ConstraintStatement-LIST '}' | ConstraintStatement-LIST

ConstraintStatement :: = OrderingStatement | ConditionalStatement | StructuralStatement

OrderingStatement ::= '{' OpenEndedStatement-LIST '}' Identifier {' OpenEndedStatement-LIST '}' | OpenEndedStatement Identifier OpenEndedStatement

OpenEndedStatement ::= ('%'|'#') FilterModuleReference

ConditionalStatement ::= Expression Identifier FilterModuleReference ['**with**' (TRUE|FALSE|VOID)]

PrimitiveElement ::= OpenEndedStatement |
'(' Expression ')'

Expression ::= NegationExpression
(('**and**'|'**or**') NegationExpression)*

NegationExpression ::= ('**not**')* PrimitiveElement

StructuralStatement ::= '{' FilterModuleReference-LIST '}'
Identifier
{' FilterModuleReference-LIST '}' |
FilterModuleReference Identifier
FilterModuleReference

A.12 Implementation

Implementation ::= '**implementation**'
ImplementationDefinition

ImplementationDefinition ::= '**by**' ClassName ';' | '**in**'
SourceLanguage '**by**'
ClassName '**as**' FileName '{' Source '}'

Source ::= (?)*

A.13 References

ConcernElementReference ::= [ConcernReference '::'] Identifier

SourceLanguage ::= Identifier

ClassName ::= ConcernReference

ConcernName ::= Identifier

ConcernReference ::= [PackageReference '.'] ConcernName

ConditionName ::= Identifier

ConditionReference ::= FilterModuleElementReference |
ConstructorReference

ConstructorReference ::= ConcernReference '()'

FilterModuleElementReference ::= [[ConcernReference '::']
FilterModuleName ':'] Identifier

FilterModuleElementReferenceStar::= [[ConcernReference '::']
FilterModuleName ':'] '*'

FilterModuleName ::= Identifier

FilterModuleRefence ::= ConcernElementReference

FilterName ::= Identifier

FilterType ::= ConcernReference

MethodName ::= Identifier

PackageReference ::= (PackageName '.')* PackageName

PackageName ::= Identifier

ReturnType ::= ConcernReference

SelectorName ::= Identifier

SelectorRef ::= ConcernElementReference

Type ::= ConcernReference

Value ::= Identifier | Number

A.14 Commonly used elements and definitions

"Elem-LIST" ::= Elem (',' Elem)*

"Elem-SEQ"	::= Elem (';' Elem)*
Identifier	::= (Letter | Special) (Letter | Digit | Special)*
ConstString	::= LowerCase (Letter|Digit|Special)* | '" (?)* '"
ConstNum	::= ['-'] (Digit)+ ['.' (Digit)+]
Number	::= (Digit)+
FileName	::= Quote (Letter | Digit | Special | Dot)* Quote
Letter	::= LowerCase | UpperCase
LowerCase	::= 'a'..'z'
UpperCase	::= 'A'..'Z'
Special	::= '_'
Quote	::= '"
Digit	::= '0'..'9'

A.15 A Template for Specifying Concerns

```
concern MyConcern ;
{ // you may repeat several filtermodule declarations
        filtermodule MyFilterModule {
                internals       // declare used internal (per instance) objects
                externals       // declare used external (global) objects
                conditions      // declare used conditions here
                methods         // declare used methods here
                inputfilters    // define the inputfilters (composed by ';')
                outputfilters   // define the outputfilters
        }
        superimposition {
```

```
            selectors          // selects classnames with queries
            conditions         // bind the listed conditions to the context
                                  specified by selectors
            methods            // bind the listed methods to the context
                                  specified by selectors
            filtermodules      // superimpose the list filtermodules at the
                                  locations
            annotations        // bind the listed annotations to the context
                                  specified by selectors
            constraints        // ordering of the filters
        }
        implementation by 'assemblyname.dll';
    }
```

Appendix B

The Selector Language of Compose*

This appendix describes the predicates that can be used in the *selector* construct of Compose* presented in Chapter 3 and 4. The meaning of a predicate, if it is not intuitive, is described after a column symbol.

B.1 (.NET) Language units

- `isNamespace(Namespace).`
- `isInterface(Interface).`
- `isClass(Class).`
- `isAnnotation(Annotation).`
- `isField(Field).`
- `isMethod(Method).`
- `isParameter(Parameter).`
- `isType(Type) :- isInterface(Type).`
- `isType(Type) :- isClass(Type).`

- *`isConcern(Concern).`*
- *`isFilterModule(FilterModule).`*

B.2 Properties of program elements

- `isNamespaceWithName(Interface, InterfaceName).`

- `isInterfaceWithName(Interface, InterfaceName).`
- `isInterfaceWithAttribute(Interface, AttributeName).`

- `isClassWithName(Class, ClassName).`
- `isClassWithAttribute(Class, ClassAttribute).`

- `isAnnotationWithName(Annotation, AnnotationName).`
- `isAnnotationWithAttribute(Annotation, AnnotationAttrib-ute).`

- `isFieldWithName(Field, FieldName).`
- `isFieldWithAttribute(Field, FieldAttribute).`

- `isMethodWithName(Method, MethodName).`
- `isMethodWithAttribute(Method, MethodAttribute).`

- `isParameterWithName(Parameter, ParameterName).`
- `isParameterWithAttribute(Parameter, ParameterAttribute).`

- ```
  isTypeWithName(Type, TypeName) :-
    isClassWithName(Type, TypeName).
  ```
- ```
  isTypeWithName(Type, TypeName) :-
    isInterfaceWithName(Type, TypeName).
  ```

- ```
  isTypeWithAttribute(Type, TypeAttribute) :-
    isClassWithAttribute(Type, TypeAttribute).
  ```
- ```
  isTypeWithAttribute(Type, TypeAttribute) :-
    isInterfaceWithAttribute(Type, TypeAttribute).
  ```

- *`isConcernWithName(Concern, ConcernName).`*
- *`isFilterModuleWithName(FilterModule, FilterModuleName).`*

B.3 Relations between program elements

B.3.1 Namespace relations

- `namespaceHasInterface(Namespace, Interface).`
  `: Namespace contains Interface.`
- `namespaceHasClass(Namespace, Class).`
  `: Namespace contains Class.`
- `namespaceHasType(Namespace, Type).`
  `: Namespace contains Type.`
- `isParentNamespace(ParentNS, ChildNS).`
  `: ParentNS is the parent namespace of ChildNS.`

B.3.2 Type relations

- `typeHasAnnotation(Type, Annotation).`
  `: Type has Annotation attached.`
- `typeHasAnnotationWithName(Type, Annotation).`
  `: Type has an annotation with the specified name attached.`
- `isSuperType(SuperType, SubType).`
  `: SubType directly inherits from SuperType.`
- `typeHasMethod(Type, Method).`
  `: declaration Type contains Method.`

B.3.3 Class relations

- `classHasAnnotation(Class, Annotation).`
  `: Class has Annotation attached.`
- `classHasAnnotationWithName(Class, AnnotName).`
  `: Class has an annotation with the specified name attached.`
- `classHasMethod(Class, Method).: the Class has Method.`
- `classHasField(Class, Field).: the Class has Field.`
- `isSuperClass(SuperClass, SubClass).`
  `: the SubClass directly inherits from SuperClass.`
- `classImplementsInterface(Class, Interface).`
  `: Class implements Interface.`

- `inherits(Parent, Child).`
  `: Child is in the inheritance tree of Parent (e.g. any subclass of Parent).`
- `inheritsOrSelf(Parent, Child).`
  `: Child is in the inheritance tree of Parent or Parent == Child (e.g. Parent and all subclasses).`

B.3.4 Interface relations

- `isSuperInterface(SuperInterface, SubInterface).`
  `: the SubInterface directly inherits from SuperInterface.`
- `interfaceHasAnnotation(Interface, Annotation).`
  `: the Interface has Annotation attached.`
- `interfaceHasAnnotationWithName(Interface, AnnotName).`
  `: the Interface has an annotation with the specified name attached.`
- `interfaceHasMethod(Interface, Method).`
  `: the Interface declaration contains Method.`

B.3.5 Method relations

- `methodReturnClass(Method, Class).`
  `: the Method returns a result of type Class.`
- `methodReturnInterface(Method, Interface).`
  `: the Method returns a result of type Interface.`
- `methodReturnType(Method, Type).`
  `: the Method returns a result of type Type.`
- `methodHasParameter(Method, Parameter).`
  `: the Method has the specified Parameter.`
- `methodHasAnnotation(Method, Annotation).`
  `: the Method has Annotation attached.`
- `methodHasAnnotationWithName(Method, AnnotName).`
  `: the Method has an annotation with the specified name attached.`

Method parameters are not ordered, but can be selected by their name. It is (currently) not possible to select parameters by order (e.g. the first argument of a method).

B.3.6 Field relations

- fieldClass(Field, Class).
  : the class (type) of a field - not to be confused with the Type that the field is a part of (classHasField).
- fieldInterface(Field, Interface)
  : the interface (type) of a field.
- fieldType(Field, Type): the type of a field
- fieldHasAnnotation(Field, Annotation).
  : the Field has Annotation attached.
- fieldHasAnnotationWithName(Field, AnnotName).
  : the Field has an annotation with the specified name attached.

B.3.7 Parameter relations

- parameterClass(Parameter, Class).
  : the class (type) of a parameter.
- parameterInterface(Parameter, Interface).
  : the interface (type) of a parameter.
- parameterType(Parameter, Type): the type of a parameter.
- parameterHasAnnotation(Parameter, Annotation).
  : the Parameter has Annotation attached.
- parameterHasAnnotationWithName(Parameter, AnnotName).
  : the Parameter has an annotation with the specified name attached.

B.3.8 Concern and Filtermodule relations

- *concernHasAnnotation(Concern, Annotation).*
  : Concern has Annotation attached.
- *concernHasAnnotationWithName(Concern, AnnotName).*
  : Concern has an annotation with the specified name attached.

- *filterModuleHasAnnotation(FilterModule, Annotation).*
  : FilterModule has Annotation attached.
- *filterModuleHasAnnotationWithName(FilterModule, AnnotName).*
  : FilterModule has an annotation with the specified name

attached.

- *concernHasFilterModule(Concern, FilterModule).*
  : Concern has FilterModule.

Appendix C

The Architecture of Compose*

This appendix gives a brief introduction to the architecture of Compose*. An overview of the architecture is presented in Figure C.1.

C.1 The Architecture Layers of Compose*

The architecture of Compose* consists of four distinguishable parts:

- **Visual Studio Plug-in** The Compose* compilation process is triggered by a Visual Studio.NET plug-in in the Visual Studio.NET environment.
- **Compose* Compile-Time** This layer consists of modules that take care of the compilation of Compose* projects. The compile-time layer of Compose* is responsible for the compilation of the sources that contain *language independent* concern specifications. All the information gathered and analyzed by these modules is stored into a central data store called *Repository*.
- **Compose* Adaptation** The adaptation layer is responsible for the generation of artefacts that act as input to the Run-Time layer. The

generated artefacts are *platform-specific*. Each platform requires the implementation of its corresponding adaptation layer.

- **Compose* Run-Time** The run-time layer is an interpreter that is responsible for the execution of compiled Compose* projects.

These four layers, presented in Figure C.1, perform the compilation and execution of a Compose* project. In the following sections, we give a description about the functionality of each layer.

C.1.1 Visual Studio Plug-in

This layer is presented by the 'IDE' block in Figure C.1. The plug-in is responsible for the configuration of all components involved in the actual compilation; however, it performs several activities before it starts the actual compilation and runtime processes. In this layer, it is determined what files are part of the project, what language they are written in and which compiler should be used to compile them. The phase of compilation is initiated from this plug-in as well.

C.1.2 Compose* Compile-Time

This layer is presented by the 'Compiler' block in Figure C.1. The Compile-Time layer consists of several modules that are responsible for extracting and processing information from the given concern sources, and storing them into the repository. The most important activities in this layer are the following: parsing the concern sources, resolving references, consistency checking of filtermodules. We explicitly mention three modules of this layer, illustrated in Figure C.1, as they are the realization of the language concepts that were discussed in Chapter 3 and 4.

- **SANE (Superimposition ANalysis Engine)** The superimposition analysis engine calculates, for each concern specification, the join points where the filtermodules should be imposed. This information is attached to all the imposed objects or concerns in the repository.
- **LOLA (LOgic LAnguage)** This module is responsible for interpreting the superimposition selectors. Selectors are expressed by a logic

language and consist of (possibly complex) combinations of constraints on program element properties and relations. Language mechanisms that were introduced in Chapter 3 and 4 are realized in the SANE and LOLA modules.

- **FILTH (FILTer composition and cHecking)** SANE and LOLA determines the structural join points where the filtermodules are imposed on based on the given selectors of a project. FILTH is responsible for processing the constraints specifications, e.g. providing the possible orderings of filtermodules that superimposed on the same join point. The language mechanisms that were introduced in Chapter 5 are realized in FILTH.

C.1.3 Compose* Adaptation

This layer is presented by the 'Adaptation Layer' block in Figure C.1. This layer consists of modules that are responsible for extracting and processing information from any input file (e.g. a concern implemented in a concrete programming language, a .NET assembly, etc.) that is platform-specific.

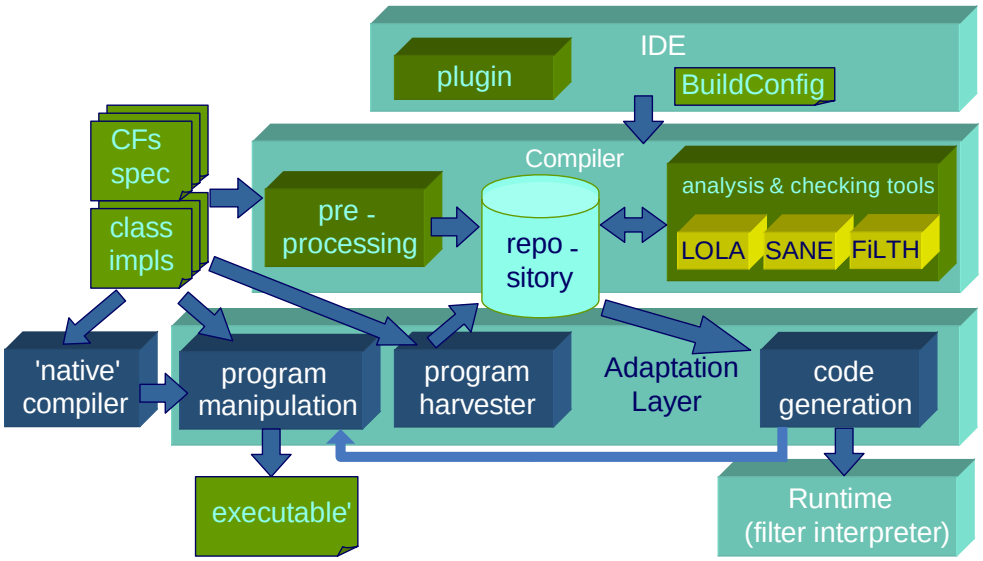

Figure C.1 *An overview of the Compose* architecture*

The most important activities in this layer are the following: compilation of language dependent concerns ('native' compiler block), extracting meta-information from target code (program harverster block), generation and manipulation of the target code, for instance, the hook instrumentation for the corresponding join points in the byte code (program manipulation and code generation blocks). This layer can also generate standalone executable modules that do not require an additional interpreter for the execution of filters.

C.1.4 Compose* Runtime

This layer is presented by the 'Runtime (filter interpreter)' block in Figure C.1. The responsibility of this layer is to provide runtime execution of Composition Filters. It creates an ObjectManager for each object as needed, and runs the appropriate filter code with it whenever a join point is reached. The execution of the interpreter is triggered by the hooks that were inserted into the target code during the hook instrumentation in the adaptation layer.

Samenvatting

Aspect-georiënteerd programmeren is een aanpak van software ontwikkeling welke nieuwe mogelijkheden biedt voor het scheiden van verschillende onderwerpen ('concerns'). Aspect-georiënteerde talen bieden abstracties aan voor de implementatie van onderwerpen waarvan de modularisatie niet kan worden bereikt met traditionele programmeertalen. Dergelijke onderwerpen worden doorgaans aangeduid als 'crosscutting concerns'. Het is algemeen geaccepteerd dat het op de juiste manier scheiden van verschillende concerns positieve effecten heeft op kwaliteitseigenschappen zoals hergebruik en aanpasbaarheid. De in software van elkaar gescheiden concerns moeten vervolgens op een zodanige manier worden samengesteld dat de totale software op coherente wijze aan de programma-eisen voldoet. We noemen taalmechanismes die gescheiden concerns samenstellen 'compositie-mechanismes'. Dit proefschrift evalueert de software compositie-mechanismes van de huidige aspect-georiënteerde talen vanuit het perspectief van software kwaliteitsfactoren zoals evolueerbaarheid, begrijpelijkheid, voorspelbaarheid en aanpasbaarheid. Op basis van deze studie worden in het proefschrift nieuwe uitbreidingen op de huidige aspect-georiënteerde talen voorgesteld waardoor programmas indien ze in deze talen worden geprogrammeerd, van hogere kwaliteit zijn.

Er zijn een aanzienlijk aantal aspect-georiënteerde talen geïntroduceerd ten behoeve van het modulariseren van crosscutting concerns. Vanzelfsprekend hebben deze talen zowel gemeenschappelijke alsmede onderscheidende eigenschappen. In dit proefschrift wordt een referentiemodel voorgesteld dat tot doel heeft de gemeenschappelijke en onderscheidende concepten van aspect-georiënteerde talen te representeren. Dit referentiemodel vormt een basis om de belangrijke karakteristieken van de state-of-the-art AOP talen te begrijpen en helpt ons om verschillende AOP talen te vergelijken. Daarnaast laat het referentiemodel zien welke zaken moeten worden bekeken bij het ontwikkelen van een nieuwe aspect-georiënteerde taal.

In dit proefschrift worden de vier voornaamste aspect-georiënteerde concepten, te weten 'join point', 'pointcut', 'advice' en 'aspect', elk geanalyseerd, resulterend in de identificatie van een aantal potentiele problemen in diverse

AOP talen. Op basis van deze analyse, worden uitbreidingen van de bestaande concepten, dan wel nieuwe concepten, geintroduceerd die de problemen addresseren.

In de huidige aspect-georienteerde talen selecteert een pointcut een aantal join points in een programma op basis van lexicale informatie zoals de namen van programmaelementen. Echter, dit reduceert de aanpasbaarheid van software, aangezien er teveel informatie wordt gebruikt in de specificatie welke hard-coded, en vaak implementatie-specifiek is. We stellen dat dit probleem kan worden beperkt door aan programmaelementen te refereren door middel van hun semantische eigenschappen. Een semantische eigenschap beschrijft bijvoorbeeld het gedrag van een programmaelement, of het bedoelde effect. We formuleren een aantal eisen voor de juiste toepassing van semantische eigenschappen in aspect-georiënteerd programmeren. We bespreken hoe semantische eigenschappen kunnen worden gebruikt voor de toepassing van aspecten in een programma, en hoe semantische eigenschappen kunnen worden toegevoegd ('geïntroduceerd') aan programmelementen. Hiertoe stellen we taalconstructies voor welke 'semantische compositie' mogelijk maken: de compositie van aspecten met die elementen van een ('base') programma, die aan bepaalde semantische eigenschappen voldoen.

De huidige advice-pointcut binding constructies van AOP talen bevatten expliciete afhankelijkheden naar advices en aspecten. Het gevolg hiervan is dat aspect specificaties minder evolueerbaar zijn en meer onderhoudsintensief zijn tijdens de ontwikkeling van een systeem. We tonen aan dat dit kan worden geaddresseerd door referenties naar advices and aspecten associatief te maken, in plaats van expliciete afhankelijkheden. Hiertoe stellen we voor om in advice-pointcut bindings de expliciete referenties te vervangen door een selectietaal welke het mogelijk maakt om te refereren middels de syntactische en semantische eigenschappen van de te selecteren elementen (dwz. Advices en aspecten). We laten ook zien hoe semantische eigenschappen kunnen worden gebruikt om herbruikbare en aanpasbare aspect abstracties te construeren.

Aspect-georiënteerde talen maken het mogelijk om gedrag -in de vorm van advice- toe te voegen aan een gespecificeerde verzameling joinpoints. Het is mogelijk dat niet slechts één, maar meerdere advices aan hetzelfde join point worden toegevoegd. Zulke "shared join points" kunnen een aantal problemen

veroorzaken, zoals het bepalen van de executievolgorde en het uitdrukken van afhankelijkheden tussen aspecten. We presenteren een gedetailleerde analyse van het probleem, en identificeren een aantal eisen waaraan mechanismen voor de compositie van aspecten op shared join points moeten voldoen. We introduceren een algemeen, declaratief model voor het definiëren van constraints op de mogelijke composities van aspecten (op shared join points). Door gebruik te maken van een uitbreiding op het gangbare join point concept, kunnen we laten zien hoe ons voorstel kan worden toegepast in concrete aspect-georiënteerde programmeertalen.

Het proefschrift laat tevens zien hoe de voorgestelde taalconstructies kunnen worden geïntegreerd in de aspect-georiënteerde taal Compose*. Teneinde de voorgestelde constructies te evalueren, wordt een kwalitatieve analyse met betrekking tot diverse software engineering eigenschappen, zoals evolueerbaarheid, modulariteit, voorspelbaarheid en aanpasbaarheid, uitgevoerd.