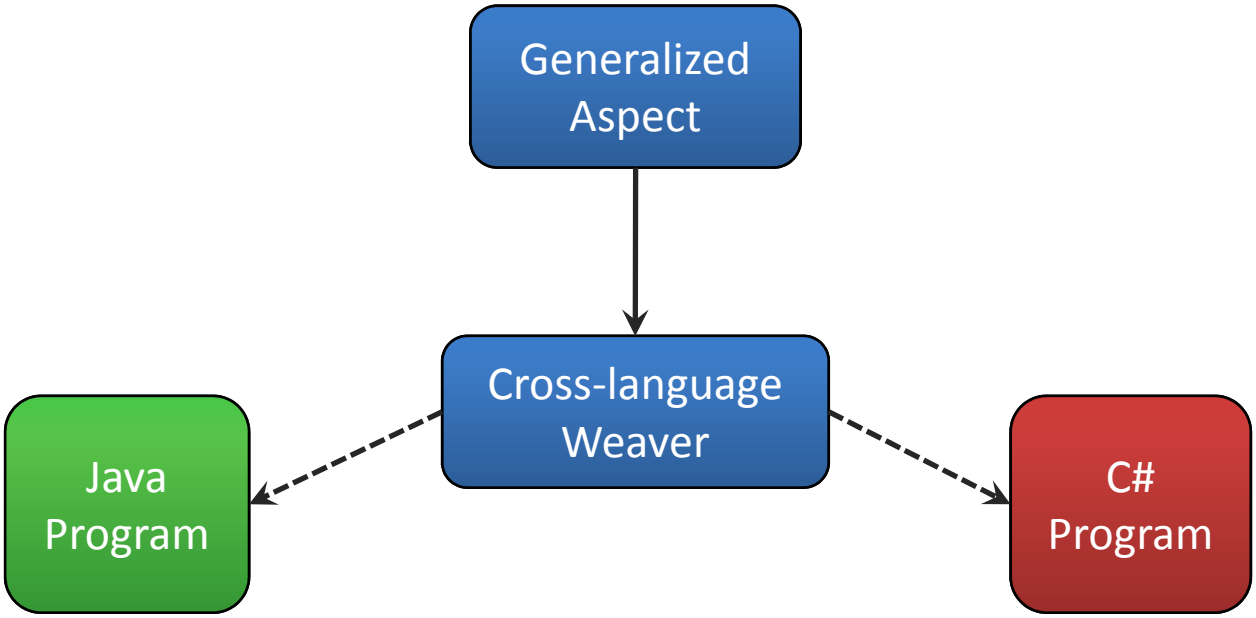
Creating CLARA

The Cross-Language Reusable Aspect-language



A Master Thesis by group *d620a*:
Brian Jørgensen, Eckhart Pedersen & Tinus Norstved
Created during SW10 at Aalborg University, Spring 2007

Aalborg University
Department of Computer Science, Frederik Bajers Vej 7E, DK-9220 Aalborg East

Title:
Creating CLARA - The Cross-Language Reusable Aspect-language

Project period:
Software Engineering 10,
Spring 2007

Project group:
D620A

Group members:
Brian G. Jørgensen,
qte@cs.aau.dk

Eckhart T. K. Pedersen,
corner@cs.aau.dk

Tinus Norstved,
tinus@cs.aau.dk

Supervisor:
Bent Thomsen,
bt@cs.aau.dk

Abstract

This report documents the design and implementation of our cross-language AOP language which can weave reusable aspects into programs written in any of several OOP target languages. Our analysis discusses AOP language features and our generalized OOP model, which is the basis for our language. In order to achieve cross-language capability we base our pointcut language on the logic-meta-programming based JTL and create a generic advice language. We demonstrate the language with two aspect examples on both Java and C# implementations of simple programs. We conclude that we have successfully shown the plausibility of creating a cross-language AOP language, but that several features require more work and further analysis.

Copies: 7
Pages: 79
With Appendicies: 93
Delivered: June 6th, 2007

Aalborg Universitet
Institut for Datalogi, Frederik Bajers Vej 7E, 9220 Aalborg Øst

Titel:
Creating CLARA - The Cross-Language Reusable Aspect-language

Projektperiode:
Softwareingeniør 10,
Foråret 2007

Projektgruppe:
D620A

Gruppemedlemmer:
Brian G. Jørgensen,
qte@cs.aau.dk

Eckhart T. K. Pedersen,
corner@cs.aau.dk

Tinus Norstved,
tinus@cs.aau.dk

Vejleder:
Bent Thomsen,
bt@cs.aau.dk

Synopsis

Denne rapport dokumenterer design og implementation af vores tværsproglige AOP sprog, der kan weave genbrugbare aspekter ind i programmer skrevet i et af flere OOP sprog. Vores analyse diskuterer forskellige AOP sprog-konstruktioner samt vores generaliserede OOP model, som er grundlaget for vores sprog. For at opnå tvær-sproglig understøttelse baserer vi vores pointcut sprog på det logisk-meta-programmerings baserede JTL, og designer et generisk advice sprog. Vi demonstrerer vores sprog ved hjælp af to eksempel aspekter for både Java og C# implementationer af to simple programmer. Vi konkluderer at vi har vist at det er muligt at lave et tvær-sprogligt AOP sprog, men at flere af vores konstruktioner kræver videre arbejde og analyse.

Kopier: 7
Sider: 79
Med bilag: 93
Afleveret: 6. juni 2007

Preface

This master thesis is written by group *d620a* and documents the effort done as part of a 10^{th} semester Software Engineering project at the Department of Computer Science, Aalborg University. It was written during spring 2007 under the Programming Technology group and the theme for our project is Aspect-Oriented Programming.

The thesis documents the development of a cross platform reusable Aspect-Oriented Language called CLARA. We assume that the reader is familiar with Aspect-Oriented Programming through reading either our previous report [13] or other papers on the subject.

As part of our implementation we make use of Java Tools Language (JTL) and we wish to thank the authors of JTL for their support with its integration into our project.

References in this report are marked by [x] where x is a number which refer to an item found in the bibliography.

The accompanying CD contains:

- A copy of the thesis and our previous report in PDF format.
- Source code and binary version of our weaver.
- Source code and binary versions of all example implementations discussed in Chapter 6.
- A Readme file with further information.

Signatures

Brian G. Jørgensen

Eckhart T. K. Pedersen

Tinus Norstved

Contents

1 Introduction **1**

2 Analysis **5**
2.1 Aspects 5
2.2 Pointcuts 6
2.3 Advices 9
2.4 Generalized OOP Model 12
2.5 Summary 17

3 Language Design **19**
3.1 Overview 19
3.2 Using the language 20
3.3 Aspects 21
3.4 Native Interfaces 22
3.5 Fields 23
3.6 Methods 24
3.7 Pointcuts 25
3.8 Advices 30
3.9 Policies 33
3.10 Advice Language 35
3.11 Summary 41

4 Weaver Design **43**
4.1 Weaver Process 43
4.2 Weaver Components 45
4.3 Using the Weaver 48

5 Weaver Implementation **51**
5.1 Java Support 51
5.2 C# Support 52
5.3 Missing Features 54

6 Demonstration of Language and Weaver **57**

6.1 Example One - Bank Application 57
6.2 Example Two - Figure Editor 64
6.3 Summary 67

7 Future Work **69**
7.1 Pointcut Language 69
7.2 Advice Language 71
7.3 Advice Types 72

8 Conclusion **75**

Bibliography **76**

A Acronyms **81**

B Demonstration: Bank Application Output **83**

C Demonstration: Bank Native Interface **85**

D Grammar **87**

CHAPTER 1

Introduction

Aspect-Oriented Programming (AOP) was conceived at the Palo Alto Research Center (PARC) and stemmed from a desire to cleanly capture complex design structures[22]. With the release of AspectJ 0.1 the goal was restated as the intent to capture crosscutting concerns, and this has since been described as being the purpose of AOP. AspectJ has since its release been the dominant AOP implementation and is often considered synonymous with AOP.

A crosscutting concern is a piece of functionality which has its implementation spread over different locations in the code. One example of a crosscutting concern is authentication. Several parts of the code may require that the user is logged in and has the correct privileges to access those parts. Implementing authentication in each method requires addition of nearly identical code in all the methods where authentication is desired. Maintaining the code is problematic, because it requires that all relevant methods are updated when changes are made, in order to ensure consistent behavior. AOP provides a solution to this problem by enabling developers to place the concern in its own module, so maintenance only has to be performed in one place.

AOP introduces new terminology: aspects, join-points, pointcuts, and advices. Aspects are modules on the same level as classes in Object-Oriented Programming (OOP), and their purpose is to modularize a crosscutting concern. Aspects address crosscutting concerns with advices, a concept similar to methods, which are executed whenever control reaches one of the locations in the program where the concern needs to be addressed. The individual locations in the code where an advice can be executed are called join-points. Pointcuts are used to specify a certain set of join-points, at which an advice should be executed. Pointcuts are usually specified as an enumeration of signatures or a signature pattern of the program elements that should be addressed.

In "Tool Support for AspectDNG - Creating an Advice Wizard for simpler AOP development" [13] we concluded that pointcuts are the cause of most of the problems that developers have with AOP. In the report we described a number of alternative AOP implementations and concluded that they are very sim-

ilar to AspectJ. Only one of the implementations contained a different way of specifying pointcuts, namely AspectDNG, which relied on XPath queries over an XML representation of the target program. Queries in AspectDNG were however still based on signature patterns. Our solution to the problem of specifying pointcuts was to provide tool support to create and modify pointcuts, in order to make pointcuts easier to create and understand.

During our work with the project we found several papers in the literature that criticized AOP, questioning the proposed benefits of AOP in regards to the software quality properties. Software quality properties are a set of properties that every software project should consider when creating their software. They describe a requirement that is either prevalent during the development of the software or later on when the project is revised. The bulk of the papers discusses three of these properties: evolvability, reusability, and understandability.

Evolvability refers to: "*the set of activities that are performed to ensure that the software continues to meet the requirements in a cost effective way*[12]. AOP affects evolvability because aspects are dependent on certain assumptions made in their respective pointcuts. These assumptions are in place because of the signature based pointcut language that are inherent in almost all of the AOP implementations. Altman et al. [2], Steimann [27], and Tourwé et al. [28] all say the same: *AOP hinders software evolution*, because a change in the base code might break some of the assumptions made in the pointcuts. The high coupling between aspects and the base code has been dubbed *the fragile pointcut problem* by Koppen and Stoerzer [18].

Reusability is a property that in McCall's Quality Factors is defined as [23]:

> The extent to which a program [or parts of a program] can be reused in other applications - related to the packaging and scope of the functions that the program performs.

Kniesel and Rho [17] claims that: "*Aspects are not reusable*". They argue that due to the signature based pointcut language, aspects addressing a universal concern (ie. a concern that is prevalent in other projects as well) cannot simply be moved to another project.

The understandability property dictates that the software should be easy to understand. AOP affects understandability because it is not possible to see that

aspects are applied to a program unless you are using an Integrated Development Environment (IDE) with support for Aspect-Oriented Software Development (AOSD). Thus understandability is hard to achieve with respect to giving a complete system overview. However aspects also improve understandability because they ensure that the classes only contain what is needed to implement their concern, and not the concerns that are crosscutting. Alexander and Biemann [1] argue that even though aspects are able to extract and modularize concerns from the base code, signature based pointcuts nullify this advantage. The problem is that it is often difficult to specify a proper signature pattern for all the methods where the concern must be applied. This can be overcome by appending a suffix to the name of all those methods, which then can be used in the pointcut. This solution however causes another problem, since methods now have to follow a specific naming convention the concern is now again visible in the code.

All of the problems above are related to the pointcut language and several papers have suggested that a different kind of pointcut mechanism, one which is based on both program behavior and structure, might solve these problem. Such a pointcut mechanism would actually be concern based, since it would focus on the properties of the concern, instead of relying on naming conventions and signatures. We believe that this idea of a concern based pointcut mechanism is a step towards the original intent of AOP, which was not about intercepting method calls according to naming conventions, but rather to address concerns that happen to crosscut an application.

In previous work[13] we discovered that most AOP implementations were created for the Java platform, and that the C# counterparts were very similar to AspectJ. We believe that this is the case because Java and C# are both OOP languages and have a similar structure. Due to their similarities a concern based pointcut language could be constructed to work on the shared set of features for both languages. If it were possible to make the other parts of an AOP language equally generic, it should be possible to create a whole AOP language that can work on multiple programming language platforms. This idea leads us to our thesis:

> It is possible to create an AOP language which allows creation of reusable aspects that can be weaved into programs written in multiple OOP languages, as long as they share a basic set of features and properties.

In order to prove this thesis we need to describe the generalized OOP model, create a pointcut language that can be used to reason about multiple target languages, and create an advice language that can be translated into multiple target language.

Overview

The circumstances for our thesis are discussed in Chapter 2 based on related work. In the chapter we analyze different parts of AOP languages in order to learn the current state of AOP and what features our language should contain. We also discuss a generalized OOP model in the analysis chapter and what implications this model has for our language.

Chapter 3 and Chapter 4 present the design of our language and our weaver. In the language design chapter we describe how our language features, such as our pointcut mechanism, are designed. In the weaver design chapter we describe how the weaver is designed in order to achieve support for multiple target languages.

Chapter 5 discusses implementation details of our weaver. Chapter 6 contains two examples of aspects and how they are weaved into an application. The examples are used to demonstrate language features and to show where a concern based pointcut mechanism makes sense.

Chapter 7 discusses future work, including ideas that were discarded and further ideas on how to make good use of our pointcut language. Finally in Chapter 8 we present our conclusion.

CHAPTER 2

Analysis

In this chapter we intend to determine purpose and use of different AOP features and constructs by examining related work. This is done in order to learn how these features should be implemented in an AOP language to create a cross-language reusable AOP language. We also discuss our generalized OOP model for the target-languages we wish to include, in order to determine whether the model is good enough compared with a language specific model.

2.1 Aspects

In this section we discuss what features aspects can contain, as well as instantiation and inheritance mechanisms.

Aspects can contain more than pointcuts and advices. Many languages allow for adding fields and methods as well. Fields are often used to store a state or a configuration of some sort between the execution of an advice. Methods serve the same purpose as in OOP, to simplify the functionality in other methods (or in advices).

Aspect Instantiation

It is important how the aspects are instantiated since that will affect how the aspect members – fields and methods – act when the aspect is applied.

In some AOP languages the aspect is instantiated as a singleton object, where all fields and methods act as if they were static. Other languages – such as AspectJ – allow the developer to specify how the aspect should be instantiated. AspectJ support instantiation per target object, per caller object, per control flow, and as singleton. A description of the different instantiation methods can be found in [13].

Inheritance

AspectJ supports creating abstract aspects where pointcuts must be implemented in a concrete aspect inheriting from the abstract aspect. Abstract pointcuts can be used to apply an aspect to a concrete system, and specify what elements of the system that the aspect should crosscut. This feature is useful to achieve reusability, as the advice effect can be defined universally, and only the pointcut may need to be changed for different projects. In line with this reasoning AspectJ only allows pointcuts and methods to be overridden, but not advices.

2.2 Pointcuts

In this section we examine different forms of pointcut languages in order to determine which best satisfies our requirements of low-coupling between pointcuts and program structure, and increased understandability.

As discussed in the introduction, pointcuts must not be coupled to program structure and naming conventions, as this would impede program evolution. The pointcut language should instead be able to capture sets of join-points that are related to the concern that is addressed by the aspect. In order to achieve this, the pointcut language may need to analyze program behavior, such as execution- or data-flow.

Pointcut Languages

Several papers have suggested alternative pointcut languages, often based on the declarative programming paradigm. The following sections describe the different approaches that are currently described in the literature.

Signature Based Pointcuts

The signature based pointcut language used in e.g. AspectJ is a relatively simple form of pointcut language. Signature based pointcuts capture join-points based on basic structural properties such as namespace, class, return type, arguments, and names visible in the signature of an element. It is usually possible to use wildcards to specify patterns of names to capture multiple elements based on naming conventions. This means that pointcuts are tightly coupled

with their captured join-points, as simple refactoring may result in overmatching or accidental misses. This form of pointcut language can only be used to query program structure, and not interdependencies.

Although AspectJ mainly relies on signature based pointcuts, it has some few additional pointcut designators that can be used to reason about interdependencies between program elements. These include dynamic features like `cflow` or `withincode` to capture method calls only from within a certain context. Still, more complex relationships cannot be expressed in the AspectJ pointcut language.

Model Based Pointcuts

In order to avoid tight coupling between pointcuts and program structure, Kellens et al. [14, 15] propose to make pointcuts query a generated model of the program, instead of the program structure itself. The generated model is a classification of program elements, i.e. it consists of sets of program elements that share a common purpose or are otherwise related.

To capture these sets of program elements, their examples use the CARMA language, which is a logic language supporting Logic Meta Variables (LMVs) and features predicates that are similar to the pointcut designators found in AspectJ.

The CARMA language is still tightly coupled with the program structure, and the authors acknowledge that this shifts the problem from maintaining proper pointcut definitions towards maintaining a proper program model. In order to simplify this task they propose a tool support solution, which tries to discover mismatches as the program evolves. The tool examines the program elements within one category of the model and determines how many of the constraints associated with the category are satisfied by each of the contained program elements. If a program element is classified as belonging to a certain category, but does not satisfy all of its constraints it is flagged as a potential mismatch, and vice versa.

The authors also state that pointcuts should be independent of the actual implementation of the generated program model, i.e. the program model could be generated using a different language than CARMA. This means that using a different model language, which decreases the coupling between model categories and the program structure may reduce or remove the necessity for mismatch checks during program evolution.

Logic Meta Programming (LMP)

LMP has repeatedly been suggested as an alternative to current typical pointcut languages in the AOP literature. LMP is a meta-programming model that uses logic queries to reason about other programs. One common characteristic of this approach is the use of LMVs. LMVs can be used to specify that e.g. two references to a class within one query should both refer to the same class, thus providing more control over queries. For more information on these concepts, refer to [17].

Several different approaches towards the use of LMP in AOP have been suggested and are discussed in the following paragraphs.

CodeQuest [11] CodeQuest is querying tool for software, that can be used for AOP. It uses Datalog as its query language and supports the addition of new predicates to the query evaluator. At the time of writing, no public version of the tool is available[10].

LogicAJ2 [25] LogicAJ2 is the successor to LogicAJ, a language with the purpose of adding LMP to AspectJ. LogicAJ2 uses a different pointcut model, based on three simple predicates that are able to capture the entire spectrum of structures in Java. The new version also contains LMV. At the time of writing, only the first LogicAJ version is publicly available[24].

Alpha [21] Alpha is an aspect language that grants access to the execution trace, the syntax tree, the heap, the static type system and more. The reason for doing this, is that it should enable the addition of more modular pointcuts, that are able to cope with software evolution. At the time of writing, no public version is available[20].

JTL [8] Java Tools Language (JTL) is designed to be a universal and high-level query language for selecting program elements in Java programs. Its purpose is to serve other source code software tools. It is not specifically created with the intent of using it with AOP, but the language is similar to the other logic languages used in pointcuts. The authors have suggested using JTL for other languages – specifically C# – in [7]. A public version is available on the official JTL wiki[6].

2.3 Advices

In this section we present related work on advices. Advices can access various information from the base program, and several ways of accessing this information are presented in the first section. The second section presents a categorization of how advices interface with base-level programs. Finally we discuss whether advices are reusable.

Context Access

Existing AOP languages provide access to different amounts of context information inside an advice. The context information can be used to perform dynamic checks at run-time such as pre- and post-conditions, or be used to manipulate the computation.

If the advice crosscut a method or constructor call, or a field access, the target object can be accessed in a similar fashion as the executing object can be accessed through a `this` variable.

The Josh language[3] give access to the arguments through variables `$1`, `$2`, with one variable for each argument without a need to specify the argument binding. AspectJ requires the advice and pointcut to specify an explicit binding of arguments, for them to available inside the advice body. The use of LMV requires the same mechanism as what is used in AspectJ.

AspectJ and other AOP languages, such as the one presented by Kniesel and Rho [17], have a mechanism for handling lists of arguments for method and constructor calls. In the language presented by Kniesel and Rho, the argument lists can be forwarded to other methods. This allow the advice to work in a generic way.

The return value of the crosscut method or field access can be used to assert post-conditions, or to determine if other information should be updated. In AspectJ the return value is accessed by capturing the return value from the `proceed` call, and in Josh the return value is accessed using the `$_` variable.

Access to the execution flow or the call stack can be used to programmaticly determine if an advice should be executed, or to use the information in tracing or logging aspects.

Interaction with The Base-program

Dantas and Walker [9] present a harmless advice language, where advices do not interfere with the base-level program. Initially the authors wanted to create an advice that was unable to perform any operation that could possible harm the further execution of the program. They came to realize later that this sort of advice would be useless, and decided to allow I/O operations as well as changes to the termination behaviour of the computation. The authors argue that the harmless advices allow for local reasoning about the program behavior, and base-level programmers can remain oblivious to the aspects crosscutting the system. However, they also argue that since they deem it necessary to allow for operations such as I/O, this means that the advice is not completely harmless anymore.

Rinard et al. [26] presents a classification system for aspects and their advices. The idea is that an advice can be categorized based on its interactions with the base code. The authors describe two different categorizes, direct and indirect interactions. A direct interaction describe an action that can change the current control flow. An indirect interaction describes how an advice uses the data used by the target method.

Direct Interactions

The direct actions are the most interesting, because they define to which extend an advice affects the program flow. The following is a summary of the different types of direct interactions:

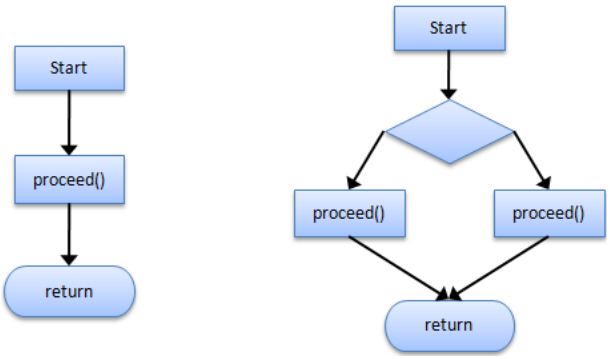


Figure 2.1: Interaction diagram: Augmentation

Augmentation An advice that always executes the target method is classified as having an augmentation interaction. The advice logic is executed before and after execution of the target method, but the target method is always reached as shown in Figure 2.1.

Narrowing An advice that only executes the target method if some condition is met is classified as having a narrowing interaction. If the condition is not met the target method is not executed and the advice exits either with an exception or by returning control to the caller as shown in Figure 2.2.

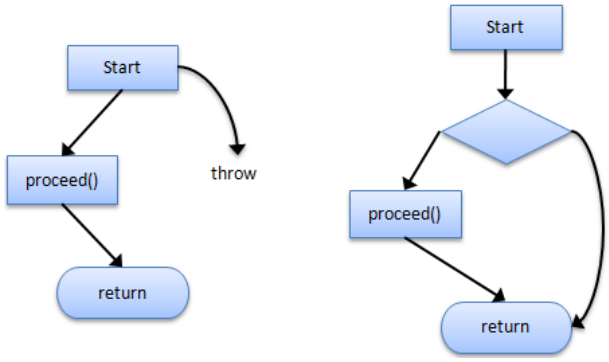


Figure 2.2: Interaction diagram: Narrowing

Replacement An advice that completely replaces the behavior of the target method, and where the target method is never executed, is classified as having a replacement interaction. Replacement advices use a static pointcut that checks for a certain condition, compared to the narrowing interaction where the condition is checked dynamically.

Combination If an advice is neither an augmentation, narrowing, or replacement advice, it is called a combination advice. The exact behavior of this form of advice cannot be summarily described or predicted.

Indirect Interactions

The indirect interaction between an advice a and a set of methods M is specified as the scope describing what distinct field accesses are made in a pair of $< a, M >$.

Table 2.1 summarizes what access operations are allowed in order for an advice-method interaction to be classified as one of the indirect interactions.

Interaction	Accesses <advice, method>
Orthogonal	<r,> <w,> <,r> <,w>
Independent	<r,r>
Observation	<r,w>
Actuation	<w,r>
Interference	<w,w>

Table 2.1: Accesses in indirect interactions. <r,w> indicates that the advice reads a field written by a method in that scope.

Using the classifications mentioned above, a harmless advice can be classified as either an augmentation or narrowing direct interaction, and a orthogonal, independent, or observation indirect interaction. Clifton and Leavens [4], Clifton et al. [5] also divides aspects or rather advice into two categories, spectators and assistants.

Reusability

If an aspect only contains code that makes use of the underlying framework and no project specific code it is reusable. The problem is that many aspects need project specific advice.

Kniesel and Rho [17] argue that the first generation of aspect languages do not support a high level of reusability, due to a lack of support for aspect genericity. They define genericity as "*the ability to concisely express aspect effects that vary depending on the context of a join point known at weave-time*". They propose integrating LMV into advices to further enhance the reusability of aspects. We discuss this further in the future works chapter (see Chapter 7).

2.4 Generalized OOP Model

In order to support multiple target OOP languages, a common set of OOP language features has to be identified. If the feature list allows for sufficient expressiveness in the AOP language, the list will be used in the design of the pointcut and advice languages. The identification analysis is preceded by a discussion of which of these features are commonly used by AOP systems. The initial analysis is based on a small set of OOP languages; Java, C#, Ruby, and Python. The four languages are represented on Tim O'Reilly's Programming Language Trends[19].

After we have identified the set of common properties in the four languages, we discuss how properties outside of the model affect the usefulness of the model. Finally we present a language that is not covered by the model.

2.4.1 OOP Features in an AOP Context

In this section we describe what properties the generalized OOP model must contain in order to provide sufficient information to be used in an AOP context.

Structural properties The structural properties can be used to restrict an aspect to a specific part of the target program. The structural properties consists of; The hierarchical structure of namespaces. Names and signatures of the set of classes, methods, constructors, fields, and constants. Access modifiers to differentiate between elements that are part of the public interface, or private implementation elements. Differentiating between public and private interfacces

Behavioral properties The behavioral properties of the program consists of the set of actions that can be performed inside a method and constructor. The actions include method calls, field accesses, and raising of exceptions.

Control flow properties The control flow properties describe the execution flow of a program at run-time. These properties can include information about the call stack and loop iterations, and be used to describe relationships among behavioral properties and control flow properties.

Constraints can be placed on the different properties to restrict what parts of the target program are affected by an aspect.

2.4.2 OOP Language Features

For each language feature, the similarities from the majority of the analyzed languages will be described, followed by a description of any irregularities.

Namespaces A namespace is used to group related classes into a single unit. The namespaces can be referenced by consumer code outside the namespace. The namespaces are placed in a name hierarchy, and they are identified by a unique name.

The namespace concept is named differently in the languages. In Java they are called packages, in C# namespaces, and in Ruby and Python modules.

The related items that can be grouped inside a namespace differ greatly among the languages; Java and C# namespaces can contain classes, interfaces, enums and respectively annotations or attributes. Ruby namespaces can contain classes, methods, fields, and constants. Ruby classes can contain methods, fields, and constants. Python namespaces can contain classes, methods, and statements.

In all of the four languages, elements do not have to be defined in a namespace, but can be implicitly defined in a default namespace.

Inheritance Inheritance allows derived classes to contain the same functionality as the class that they are derived from. This allows for greater reusability, as different classes with different functionality may at the same time share the implementation of common functionality. Java, C#, and Ruby all have single inheritance on classes, which means that classes in these languages can each only inherit functionality from a single class. In Java and C# interfaces can be used to simulate multiple inheritance. Python is the only language that has multiple inheritance, allowing classes to inherit functionality from several classes. Ruby has mixins that can be used as multiple inheritance.

Class Members Classes can contain a set of class members. All four languages can have methods, constructors, fields, and constants. Classes can be nested by placing a class definition inside another class – named inner classes.

Access Modifiers Access modifiers are used to define a public interface to consumer code for a module, and to hide implementation details. All languages have a mechanism for creating public and private methods. Python uses *name mangling*[1], where private methods have two underscores prepended to the name.

Elements can be specified as protected in Java, C#, and Ruby. Protected elements can be accessed by classes that extend the class, but not by consumers. In Java and C# access to elements can be restricted to only other members of the same package using respectively the package or internal access modifiers.

[1]Using a naming convention to specify a specific property of an element.

Typing The analyzed languages either have static or dynamic typing. All the languages have strong typing. Java and C# both have static typing, where the type is determined at compile time. Ruby and Python have dynamic typing, all using duck typing, where method calls are not targeted a specific type, but instead any object that contains a method with the specific signature.

Higher-Order Functions All the languages have some level of support for higher-order functions. Java uses anonymous inner classes to simulate anonymous methods, where the anonymous inner class must implement a specific interface. C# version 1 and 2 have support for delegates, which are function pointers based on a method signature. C# version 3 has lambda expressions.

Built-in Data Types The languages have a wide range of built-in data types, but they differ greatly from one language to another. All languages have string and 32-bit integer built-in data types. Java, C#, and Python have a boolean data type, but Ruby does not have a boolean data type. Instead Ruby considers anything except `false` and `nil` as true.[2] In Ruby, all numeric data types such as integers of different size and floating points are combined into a single numeric data type.

Exceptions All the languages have exceptions and support propagating exceptions up through the call stack from where it is raised until it is caught by an exception handler. Custom exceptions can be created by extending a root exception class.

Java support declared exceptions and run-time exceptions, where declared exceptions can only be thrown by methods that have declared that they can throw such an exception. Ruby have rescue exception handlers, that allow the associated code block to be executed again.

Concurrency All the languages have support for concurrency, either native support or through libraries. The concurrency models can differ greatly, and thus make it hard to find similarities between them.

Control Structures All languages have support for conditional branching such as if and switch statements, as well as different types of loops.

[2] `http://en.wikipedia.org/wiki/Boolean_datatype#Ruby`

2.4.3 Common Features in OOP Languages

After having analyzed the different OOP features in the four languages, we have created a list of the common characteristics. Some of the languages provide multiple possibilities for a specific feature, but we here only look at the common denominator.

Related classes are grouped into a hierarchical structure of namespaces and classes. A class can contain methods, fields, constants, and nested classes, that can have different public and private access modifiers.

All languages have support for higher-order functions, although the degree of support varies. All languages support four basic data types as built in data types. These data types are strings, integers, floating point numbers and booleans. Although Ruby does not acually have a boolean data type as such, all values can be interpreted as boolean types, with all values other than `false` and `nil` being true.

All languages have a mechanism to handle exceptions during execution. Custom exceptions can be declared by the developer, and exceptions can be thrown and caught. Exceptions can be propagated up through the call stack until they reache an exception handler or result in a runtime error.

2.4.4 Model Limitations

The four analyzed languages all have features that can not be mapped directly to the generalized model. If a language specific model of join-points is created for each of the languages, it could contain additional join-points compared to the generalized model. If important join-points are missing from the generalized model, but present in a language specific model, the generalized model would be less useful than a language specific model. If the missing join-points are very important concepts in the given language, this could mean that our language would be unusable to create aspects for programs written in that language.

The set of access modifiers supported by the four languages differ, and the generalized model only contain public and private modifiers, where the latter represents all modifiers that are not public. We have observed in the literature and aspect examples that when access modifiers are used, they are used to identify elements part of the public interface. Aspects intercept for instance

elements in the public interface and enforce pre and post conditions on the methods. This does not require the aspect to distinguish between protected or private elements, but only between public and not public elements.

In Ruby and Python it is also possible to place program elements outside of the general structure of the model, such as placing a function outside of classes and statements outside of methods. If a program contains much functionality of this type, using the current model would result in parts of the program not being addressable by aspects. Further analysis is needed to map these elements into a generalized model.

All of the features that are not contained in the generalized model are not supported by any of the major AOP implementations we have seen so far. Our model is thus at least as expressive as that of any of the existing implementations, and should be a good basis for an AOP language.

2.4.5 Languages that do not fit the Model

Not all OOP languages can properly be mapped into the generalized OOP model. Smalltalk is an example of such an OOP language. Smalltalk does not have a namespace mechanism, although some Smalltalk implementations – such as GNU Smalltalk – have their own namespace mechanism. Furthermore, exception handlers in Smalltalk can only be associated with classes, and are only searched for in the class where the exception is raised. It is thus not possible to propogate exceptions to callers. Since smalltalk includes too many deviations from our generalized OOP model, we believe that our language cannot be used to create aspects for smalltalk.

2.5 Summary

Our intention is to create a new AOP language, and in this chapter we examined whether the usual constructs in an AOP language could be improved upon. Most notable is the pointcut language, which we propose should be changed to use a logic based language. Having a logic language for pointcuts enables us to talk about concerns instead of signatures. Furthermore we propose adding different advice types, that differ in how they change the program flow. The purpose of adding more advice types is to implicitly improve reusability through improving readability and understandability.

One of the demands for our language is that is supposed to work on several OOP languages instead of being language specific. Therefor we created a generalized OOP model in order to demonstrate that it possible to find a set of common features for multiple OOP languages.

CHAPTER 3

Language Design

This chapter discusses our language design based on the requirements from our thesis in the introduction. The first two sections give an informal overview of the overall design-decisions and how our language is used, while the remainder of the chapter provides a more formal and detailed discussion of our language features.

3.1 Overview

While our language was designed from scratch, it builds on the existing ideas of AOP languages. As such we reuse the concept of aspects representing cross-cutting concerns, pointcuts are used to target specific parts of the base program, and advices are used to implement the desired behavior.

The difference to existing implementations lies in the details of the pointcut and advice concepts. In order to decrease the coupling between pointcuts and program structure, as well as meet the reusability requirement, pointcuts are based on a LMP query language. This makes it possible to reason about behavior and concepts, instead of only structure and signatures as in AspectJ. Similarly advices themselves are more reusable in our language as they are not written in the language of the specific base program, but instead are written in a new language that matches our generalized OOP model. Advices in this language should then be easily be translated into the target language, as long as the target language supports the same features as our generalized OOP model.

Since our advice language is kept simple it sets a certain limit for the power of expression in advices. In order to avoid this issue we decided to introduce a new concept called native interfaces, which make it possible to reference code written in the target language. An aspect that includes a native interface declaration can only be woven into the target project, if the developer provides a class that implements the specified interface. Native interfaces thus give full access to functionality of the target language, while their interface nature

still upholds the cross-platform and reusability requirement. Native interfaces are not suppose to be used by every aspect, they are only suppose to be used when the desired functionality is not possible to express in the aspect. Native interfaces are explained in further detail in Section 3.4.

In addition to making the language features more reusable in themselves, we want to enhance reusability by improving understandability. This means that we introduce new advice types based on the classification of advices in Section 2.3 in order to make it more clear what form of advice is used. Furthermore we introduce the policy feature which should make it easier to enforce policies.

Other common AOP features, such as introductions and control-flow based pointcuts, are not included in our language. While we recognize their uses, we feel that they are not required to support our thesis. Instead we wish our language design to be minimalistic in nature so that we we can focus on those features that are important to support our thesis. The minimalistic nature of our language will also become apparent in the following sections, which discuss our language features in further detail.

3.2 Using the language

This section describes how the language should be used to make best use of the reusability property. Since aspects are meant to be reusable, it should be possible to create aspect libraries that can be included in a development project. These aspect libraries should contain aspects that implement as much of the functionality as possible, but still allow for project specific customizations, we call these aspects abstract aspects.

In order to include a specific aspect into the project it would then be necessary to extend that aspect with a concrete implementation. The amount of required customization in a concrete aspect should be minimal and mostly consist of project specific bindings, such as specific class/method names or similar.

If an aspect contains a native interface definition, the developer must also provide an implementation of that interface before the aspect can be weaved into the target project. The weaver for our language should provide a way of generating an interface file for the specific target language, so that the user can implement that interface.

3.3 Aspects

Our language differentiates between abstract aspects and concrete aspects, as concrete aspects typically are project specific, while abstract aspects should only implement those parts that are not project specific. The syntax for our aspect declarations thus looks like this.

⟨aspect⟩ =
    'abstract' 'aspect' ⟨identifier⟩ '{' ⟨aspect-body⟩ '}' |
    'aspect' ⟨identifier⟩ 'extends' ⟨identifier⟩ '{' ⟨aspect-body⟩ '}' |
    'aspect' ⟨identifier⟩ '{' ⟨aspect-body⟩ '}'

The aspect-body contains a list of members, that can be pointcuts, advices, policies, native interface, fields or methods. Each of these elements is described in more detail in the following sections.

⟨aspect-body⟩=
    ⟨aspect-member⟩ |
    ⟨aspect-body⟩ ⟨aspect-member⟩

⟨aspect-member⟩=
    ⟨pointcut-declaration⟩ |
    ⟨advice-declaration⟩ |
    ⟨policy-declaration⟩ |
    ⟨native-interface-declaration⟩ |
    ⟨field-or-method-declaration⟩

Inheritance

Only abstract aspects can be extended, and the extending aspect must be a concrete aspect. Inheritance between two concrete aspects or two abstract aspects is not supported. The inheritance mechanism merges all members of the concrete and the abstract aspects. Any member in the concrete aspect that is also present in the abstract aspect always replaces the implementation from the abstract aspect. It is thus not possible to redirect calls to the abstract aspect. This approach was chosen because we did not want to add any features that we were not sure would be useful. If the need for deeper leveled inheritance hierarchies or a super-aspect concept arises at a later time, these features could always be added later.

Aspect Instantiation

Aspects are always instantiated as singletons and it is not possible to specify other instantiation mechanisms. This approach was also chosen due to our goal of keeping the language design minimalistic. While other instantiation mechanisms can be useful, they are not essential to support our thesis and can be added in future iterations of the language design.

3.4 Native Interfaces

The first aspect member we are going to describe is the *native interface*. The purpose of this member is to ensure that the expressive power of aspects is not limited to features in our advice language, while still ensuring that the aspect can work on different languages. Since our advice language is generic, it is not possible to invoke anything that is language specific in aspects. For instance, in an authentication aspect it would not be possible to display a login dialog which requires a language specific implementation. However, by using a native interface we can do precisely that.

The native interface is just like a typical OOP interface containing only method declarations, with the exception that it is declared in the aspect instead of a separate file. The interface is placed in the aspect file since the interface is directly required in that aspect, and this ensures that the interface definition is always present together with the aspect. The syntax for native interface declarations is:

⟨native-interface-declaration⟩ =
    'NativeInterface' '{' ⟨native-interface-member-list⟩ '}'

⟨native-interface-member-list⟩ =
    ⟨native-interface-member⟩ |
    ⟨native-interface-member-list⟩ ⟨native-interface-member⟩

⟨native-interface-member⟩ |
    ⟨identifier⟩ ⟨identifier⟩ '(' ⟨method-argument-list⟩ ')' ';'

Example

As an example, consider an authentication aspect that needs to display a login dialog if the user tries to access certain parts of the application. This aspect could contain the following native interface declaration.

```
1 NativeInterface {
2     bool authenticate();
3 }
```

The interface should be read just as any normal interface, meaning that it defines one method called *authentication* which returns a boolean. Before the aspect containing the native interface declaration can be woven into the target project, the interface must be implemented. In order to implement the interface, the weaver is used to generate a interface declaration in the target language. The following example shows how this may look with Java as the target language.

```
1 interface Authentication_NativeInterface {
2     boolean authenticate();
3 }
```

After implementing the interface in the target language, the aspect can be woven into the target program, and all references to the native interface are transformed into references to the class implementing the interface. In the above example the authenticate method on the interface could thus be used to bring up a login dialog, which returns boolean specifying whether the login was successful or not, as shown here.

```
1 isAuthenticated = NativeInterface.authenticate();
```

3.5 Fields

Fields can be used to store state between advice executions, and can be declared with or without a default value. If a default value is specified, it will be assigned when the aspect is initialized. The syntax for fields is as follows.

⟨field-declaration⟩ =
    ⟨identifier⟩ ⟨identifier⟩ '=' expression ';' |
    ⟨identifier⟩ ⟨identifier⟩ ';'

Listing 3.1 show an example of how a field can be declared, and referenced from an advice.

```
1 aspect FieldTest {
2   int count = 0;
3   pointcut targetPointcut[M]: "class 'java.lang.String members[M] M.method";
4
5   advice hits: around targetPointcut {
6     count = count + 1;
7     print("Method hits: " + count);
8   }
9 }
```

Listing 3.1: Declaration and use of an aspect field.

3.6 Methods

Methods can be used as auxiliary methods to simplify the logic of advices, and function similar to methods in OOP classes with arguments and a return value. Methods can be invoked as statements in the body of other methods or in advices, or as part of an expression.

Method declarations have the following syntax

⟨method-declaration⟩ =
    ⟨identifier⟩ ⟨identifier⟩ '(' ⟨method-argument-list⟩ ')' '{' ⟨method-body⟩ '}'

⟨method-argument-list⟩ =
    ⟨method-argument⟩ |
    ⟨method-argument-list⟩ ',' ⟨method-argument⟩

⟨method-argument⟩ =
    ⟨identifier⟩ ⟨identifier⟩

Abstract aspects can, in addition to concrete method declarations, also contain abstract methods, where the implementation is postponed to a concrete aspect. Abstract methods can be referenced from within concrete methods and advices in the abstract aspect.

⟨abstract-method-declaration⟩ =
    'abstract' ⟨identifier⟩ ⟨identifier⟩ '(' ⟨method-argument-list⟩ ')' ';'

Listing 3.2 show an example of how an abstract aspect (`MethodCache`) can declare an abstract method (`getKey`), which is used in the advice `cacheResult`. The concrete aspect (`MyMethodCache`) contains an implementation of the method.

```
1  abstract aspect MethodCache {
2      abstract string getKey();
3
4      advice cacheResult: around targetPointcut {
5          string key = getKey();
6          ...
7      }
8  }
9
10 aspect MyMethodCache extends MethodCache {
11     string getKey() {
12         return "mykey";
13     }
14 }
```

Listing 3.2: Declaration and use of abstract method.

Abstract methods and *native interfaces* have similar purpose, to delegate the implementation of functionality while providing a clear interface to that functionality. The difference is that abstract methods are implemented with our advice language, while native interfaces must be implemented with the target language. As the expressive power of our advice language grows, the need for native interfaces should diminish, while there likely always will remain some need for abstract methods.

3.7 Pointcuts

Our pointcut language is based on an already existing implementation in order to be able to focus on other parts of the language. We chose JTL because it is an LMP based language, and because it was designed to be a pluggable component. Incidentally it was also the only language that was publicly available of the languages we examined in Section 2.2. JTL also aims to enhance understandability by using a English-like syntax[8], which also makes it a good choice since enhancing understandability is one of our goals.

3.7.1 Pointcuts as JTL queries

Since we use JTL to evaluate pointcuts, pointcut expressions are the same as JTL queries. This section discusses syntax and semantics of JTL queries by showing how such queries are formed when running JTL as a command-line application. It should be noted that we do not discuss all of the features to be found in JTL, but only try to give a brief introduction. For more information on the usage of JTL we suggest reading *JTL - The Java Tools Language* [8].

Since JTL is a first order predicate language, the central concept in JTL queries are predicates. Predicates are used to reason about properties and can be used to determine the set of elements that have a certain property. A simple example is the `class` predicate which can be used to specify that the element to be found, also called the *subject*, is a class. A query in JTL is nothing more than the evaluation of a predicate itself using the run keywords as in the following example:

```
1 run method;
```

This query should return all methods that JTL know about, assuming that the `method` predicate has been defined. Predicates can be defined either as native predicates or as user defined predicates. Native predicates are predicates that are defined as part of the evaluation engine itself, while user defined predicates are predicates that the user has defined by combining several native or other user defined predicates. Native predicates usually define atomic properties which cannot be deduced by looking at other properties. The class property is such an atomic property, either an element is a class or it is not, the answer cannot be found by examining other properties, e.g. access modifiers. This means that the `method` predicate must be defined as a native predicate, because the user cannot define his own `method` predicate based on other predicates.

User defined predicates often match elements with a specific set of properties by combining several properties. For instance, in theory the following query should find all public methods that do not return a value.

```
1 pv_method := public void method;
2 run pv_method;
```

This query will not return anything however, since JTL requires a query predicates to specify a specific set of classes to operate on. This can be done by using a predicate specifying a class combined with predicate that reasons about the relationship of two elements, such as in the following example.

```
1 pv_method := public void method;
2 all_pv_method[M] := class members[M] M.pv_method;
3 run all_pv_method;
```

The first line defines the predicate that specifies that the subject is a public void method. The second line specifies that the subject of the `all_pv_method` predicate is a class, that the class contains member `M`, and that `M` is a public void method. Incidentally, `M` is what is called a LMV, as it is both a logic variable, and a meta variable (as in meta-programming). By specifying the LMV `M` in

the argument list of the `all_pv_method` predicate, JTL knows that the value returned by the query should not be the subject of the predicate, which is a set of classes, but instead those elements that match `M`. In this case the set of elements would consist of all public void methods in any of the classes that JTL knows about. This set can be further restricted by specifying a specific class or set of classes to work on, as in the following example.

```
1 pv_method := public void method;
2 all_pv_method[M] := class is[/java.lang.String] members[M] M.pv_method;
3 run all_pv_method;
```

The `/` denotes that the following is a string containing the name of a class. Regular expressions can also be used, as in the following example:

```
1 pv_method := public void method;
2 all_pv_method[M] := class 'java.lang.St?* members[M] M.pv_method;
3 run all_pv_method;
```

Instead of using the `is` predicate, the above example makes use of a shorthand notation using `'`. Regular expressions in JTL are similar to other well known regular expressions, except for using the `?` instead of the `.` symbol to denote "any character". The logical or operator can also be used in queries. The following example finds all elements that are either a field or method in the `java.lang.String` class:

```
1 field_or_method := [ method | field ];
2 all_pv_method[M] := class 'java.lang.St?* members[M] M.field_or_method;
3 run all_pv_method;
```

3.7.2 Pointcut Declaration Syntax

We use the JTL predicate syntax for out pointcut expressions, with some minor changes. Pointcuts must begin with the pointcut keyword and all pointcuts must have exactly one argument which corresponds to the join-point. The syntax thus looks like this:

⟨pointcut-declaration⟩=
'pointcut' ⟨identifier⟩ '[' ⟨argument⟩ ']' ':=' '"'⟨pointcut-expression⟩'"'

Pointcuts must explicitly state which LMV represents the join-point or set of join-points in order to make it more clear what the join-point is. If we allowed some pointcuts to use the implicit subject as the join-point it might be slightly more difficult to understand what the join-point is in each pointcut.

3.7.3 Native Predicates

Section 2.4.1 describes three distinct sets of properties that our generalized OOP model must contain in order to be useful in an AOP context. In order to make use of these different properties the pointcut language needs to provide predicates that can be used to reason about these properties. This means that we need predicates to reason about structural properties, behavioral properties as well as control-flow properties. However, since control-flow properties are relatively complex we decided to ignore these in our first version. The following sections describe those predicates that we have found to be necessary.

Structural Predicates

Structural predicates are used to reason about the structural relationships between entities. They are still more powerful then signature based pointcuts such as provided by AspectJ, since together with LMVs they can be used to express complex structural relationships. The following list of predicates allows us to reason about all of the parts in our generalized OOP model.

X.int subject X is of type int

X.float subject X is of type float

X.bool subject X is of type bool

X.string subject X is of type string

X.void subject X is of type void

X.class subject X is a class

X.method subject X is a method

X.field subject X is a field

X.constructor subject X is a constructor

X.constant subject X is a constant, i.e. an immutable field

X.member subject X is a class member, only returns false for class elements that are not nested inside another class.

X.private subject X is visible in the declaring class only

X.public subject X element is visible in all classes

X.static subject X element is static

C.extends[C′] subject class C extends the class C′ given as argument

M.overrides[M′] subject Method M overrides the Method M′ given as argument

C.members[M] subject class C contains a member which is the argument M

M.member_of[C] subject M is a member of the class C given as argument

Behavioral Predicates

Behavioral predicates make it possible to reason about more than pure structural behavior. For instance, they can be used to implement synchronization mechanisms which update some state when a specific set of fields has been written to. The following list of predicates allows us to reason about all of the behavioral properties that our generalized OOP model provides.

M.reads[F] subject method or constructor M reads field F

M.writes[F] subject method or constructor M writes to field F

M.invokes[M′] subject method or constructor M invokes method M′

M.creates[T] subject method or constructor M creates an object of type T

Comparison Predicates

Sometimes it may be necessary to determine whether a given element has a given type or is equal (or different) to another element, comparison predicates make this possible. We thus include the following comparison predicates.

X.is[T] The type of X is type T

X.is*[T] The type of X or any of its super classes is type T

X.eq[X′] subject element X is the same element as argument X′

3.8 Advices

Similar to other AOP languages, our language features no *before* and *after* advice types, as these can easily be emulated using around advices. A deviation from most AOP languages is that our language requires that advices are named because this makes it possible to refer to a specific advice, instead of just a signature. A well chosen name may also help with understandability.

In order to improve understandability further we decided to provide several versions of around advices based on the analysis of direct interactions in Section 2.3. We hope that stating explicitly how the advice interacts with the base program makes it easier to understand what the purpose of the advice is. In order to make this feature useful the weaver must ensure that each of the respective advice types can only be specified for advices that have the correct behavior.

We do not intend to include any advice type based on the analysis of indirect interactions, because they are more complicated to reason about for the developer. The direct interactions deal with control flow, and thus only result in two possible outcomes: either the advice method is called, or it is not. The indirect interactions deals with how advice possibly modify fields or other context around the advice method, and while this may provide useful information to the programmer it requires much more thought to determine whether the advice is important to look at or not.

The syntax for advice declarations is shown below.

⟨advice-declaration⟩=
    'advice' ⟨identifier⟩ ':' ⟨advice-type⟩ ⟨pointcut-identifier⟩ '{' ⟨advice-body⟩ '}'

Around advice

Around advices provides the superset of functionality for all the other advice types. The example below displays a typical around advice.

```
1 advice checkAuthentication: around target {
2     if (testAuthentication()) {
3         proceed();
4     } else {
5         NativeInterface.deny();
6     }
7 }
```

Harmless advice

The harmless advice is based on the harmless advice described by Dantas and Walker [9]. In semantic terms, this advice must contain a `proceed` statement in which the intercepted method is called and the advice should not be able to cause an error that would stop the execution of the intercepted method. This means that there are certain restrictions in a harmless advice, it should for example not be possible to use *native interfaces* at all, since we can not predict the outcome of such an action.

```
1 aspect Tracing {
2     pointcut allMethods[M]: "class members[M] M.method";
3
4     advice traceMethod: harmless allMethods {
5         print("entering method: " + ?M.Name);
6         proceed();
7         print("leaving method: " + ?M.Name);
8     }
9 }
```

The effect of having a harmless advice type, is that it safely can be ignored by the programmer concerned with the base code. A simple example is seen above, where the aspect is used for tracing. Note that our language does not contain a *print* statement – the example above is made to illustrate how we envision the harmless advice could be used.

Narrowing advice

The *narrowing advice* is an advice type, where the target method may or may not be called. Semantically this means that there is a chance that the advice may call proceed, depending on some condition. Normally this is easily done using an if-statement but we want to make it more explicit, and base it on a boolean condition. Therefor we propose the following structure of a narrowing advice:

⟨narrowing-advice-body⟩ =
'condition' '{' ⟨boolean-statement⟩ '}' 'successs' '{' ⟨advice-body⟩ '}' 'failure' '{' ⟨advice-body⟩ '}'

The example used in the around advice described before could thus look like the following:

```
1  advice checkAuthentication: narrow targetPointcut {
2      condition {
3          testAuthentication();
4      }
5      success {
6          proceed();
7      }
8      failure {
9          NativeInterface.deny();
10     }
11 }
```

The narrowing advice is all syntactical sugar, since it is suppose to be transformed into a normal around advice containing an if-then-else statement. Programmers need to be aware of narrowing advice, and we believe that it is easier to get an overview of the situation if all you have to evaluate is the boolean statement.

Based on our previous project on tool support for aspect-oriented programming[13], we envision that in the future this could be built into an IDE. Instead of just showing the advice name on mouse over, the actual condition could be showed to the user.

Replacement advice

The final advice type is called *replacement advice*, because it completely replaces the functionality of the base code with what there is in the advice. The idea behind the replacement advice is that the programmers can focus on the advice, instead of the methods that are being captured by the advice. A replacement advice never contains a proceed call, but can still be simulated by the around advice, since the only difference is that the advice should not contain a proceed call.

```
1  aspect ReplacementAspect {
2      pointcut legacyMethod[C] : "class is[/java.lang.System] members[M] M.name['
           out.println (_)];
3
4      advice replacementAdvice: replaces legacyMethod {
5          // fancy new print line method;
6      }
7  }
```

Advice Ordering

Advices order is determined by the order that they are written in aspects, the first advice being woven as the innermost advice, while aspects are woven in the same order as they are supplied to the weaver. We feel that this provides an acceptable level of control and understanding over advice ordering, while still keeping the language simple. Ideally we would want to include a language mechanism to explicitly specify ordering between advices, but since this is not an essential feature we decided not to include it.

Summary

Having more than one advice type grants the programmer a quicker and clearer understanding of what the advice actually does, which should make it easier to reuse aspects. All of the types above should be transformed into an around advice, but the only one that is semantically like an around advice, is the narrowing advice. The harmless and replacement advice are different, because they impose certain restrictions on top of the around advice, that normally is not present there.

3.9 Policies

Policies allow developers to specify certain constraints on the code, as such they share the same purpose as the *declare error* and *declare warning* constructs in AspectJ. The policies construct can only be used to declare errors, since we feel that a policy that is not enforced is useless. During development it may be useful to declare warnings instead of errors, but we do not consider this an essential feature. Each policy has a name to make it possible to refer to a specific policy and to improve understandability, similar to advices. The syntax for creating a policy is as follows:

⟨policy-declaration⟩=
    'policy' ⟨identifier⟩ 'forbids' ⟨pointcut-identifier⟩ ':' ⟨string⟩ ';' |
    'policy' ⟨identifier⟩ ⟨pointcut-identifier⟩ 'requires' ⟨pointcut-identifier⟩ ':' ⟨string⟩ ';'

In contrast to advices policies allow pointcuts to capture more than only methods, otherwise the functionality of policies would be severely limited. In the following we describe the two policies and how they differ from an example where we want to ensure that singleton classes have a private constructor.

Forbids Policy

The basic policy - the *forbids* policy - corresponds directly to the AspectJ *declare error* construct. This means that if the forbids policy matches anything it will produce a compiler / weaver error.

```
1 pointcut privateConstructor[M] := "private ctor";
2 pointcut singleton[C] := "class is[C] { public static method is[C] 'getInstance;
      }";
3 pointcut SingletonWithoutPrivateConstructor[C] := singleton { no
      privateConstructor};
4
5 policy noSingeltonsWithoutPrivateConstructor forbids
      SingletonWithoutPrivateConstructor : "A singelton class needs to have a
      private constructor";
```

The first pointcut in the example above defines that the class needs to have a private constructor. The second pointcut – *singelton* captures all the classes that are suppose to be singeltons (defined by having a `getInstance` method). Finally the third pointcut combines the two pointcuts in order to capture the classes that are suppose to be singeltons, but does not have a private constructor.

Requires Policy

The *requires* policy is syntactical sugar on on top of the *forbids* policy. It allows the developer to express that program elements with certain properties must also have some other properties. It is best explained from an example:

```
1 pointcut privateConstructor[M] = "private ctor";
2 pointcut singleton[C] = "class is[C] { public static method is[C] 'getInstance; }
      ";
3
4 policy SingeltonRequiresPrivateConstructor singleton requires privateConstructor
      : "A singelton class needs to have a private constructor";
```

The example uses the same two pointcuts as defined in the forbids example, but doest not require an extra pointcut to combine the two. Although the *requires* policy is no more expressive than the *forbids* policy, we believe that it makes easier to read and understand certain policies.

3.10 Advice Language

Our advice language covers all the of code that can be written inside methods and advices. As such it contains the typical elements of an OOP language. This includes data types, variable declarations and assignment statements and control structures. Finally our advice language differs in the way that values can be returned, especially the way return values from proceed calls are handled.

Data Types

In the first version of our language we have decided to support the data types shown in Table 3.1, based on the analysis in Section 2.4:

bool	int	float	string	object	HashTable	List

Table 3.1: Supported types in our language

These types are available in all of the OO-languages we have examined. The object data type in our language represents any type and requires no type casting. Instead it is implicitly cast to the correct type, since we did not want to include a typecasting mechanism in this version. *List* and *HashTable* are not known by the same name in the different languages, but they provide the same functionality. List is an data type that holds a collection of entities, much like an array, though with the possibility to change the size dynamically. List allows for the addition or removal of entities, looping over them in a foreach or while statement and accessing the members through an indexer. An example of how to use the list is given below.

```
1  List customers;
2
3  customers.add("john doe");
4  customers.add("jane doe");
5  customers.add("james doe");
6
7  // Removes "james doe" from the list
8  customers.remove(2);
9
10 // Prints "john doe" and "jane doe"
11 foreach(customers as customer) {
12     print(customer);
13 }
14
15 // Access the second element in the list. Prints: "jane doe"
16 print(customers[1]);
```

The HashTable is a data type that also holds entities that are saved with a unique key. The key can be used to fetch the entity, or to remove the entity from the collection. Furthermore it is possible to check whether a collection contains a certain entity by using its key. The hashtable allows for the addition of new entities by specifying a key and the entity and removing entities by using the key only.

```
1  HashTable customers;
2
3  customers.add("john", "john doe");
4  customers.add("jane", "jane doe");
5  customers.add("james", "james doe");
6
7  // Test if customers contains "jane"
8  if (customers.containsKey("jane")) {
9      print("Contains jane");
10 } else {
11     print("Does not contain jane");
12 }
13
14 // Removes "james doe" from the hash table
15 customers.remove("james");
16
17 // Prints: john doe
18 print(customers["john"]);
```

Both the List and the HashTable store content as object types. This is a decision we made to simplify the language, and may later be revised to include support for generics. These types do not represent every type that could be useful, but they are enough to showcase our ideas. Besides the types above, we support also support the *void* return type.

A complete list of operations available on the two data types List and HashTable are shown in Table 3.2 and Table 3.3.

List members

Member	Description
`void add(object E)`	Add entry E to list.
`void remove(int I)`	Remove entry at index I.
`int length()`	Get length of list.
`object [int I]`	Get entry at index I.

Table 3.2: All members on the List data type.

HashTable members

Member	Description
`void add(object K, object E)`	Add entry E with key K
`void remove(object K)`	Remove entry with key K
`int length()`	Get entry count of table
`bool containsKey(object K)`	Test if table contains entry with key K
`object [object K]`	Get entry with key K

Table 3.3: All members on the HashTable data type.

Operators

Our language supports a basic set of operators that are shown in Table 3.4.

Operator	Description
+	Addition and string concatenation
-	Subtraction
/	Divisions
*	Multiplication
||	Boolean or
&&	Boolean and
=	Assignment
==	Equals
!=	Does not equal
>	Greater than
<	Less than
>=	Greater or equal
<=	Less or equal

Table 3.4: List of operators

Variables

Variable declarations and assignments work the same way as in Java and C#.

⟨declaration-statement⟩ =
    ⟨identifier⟩ ⟨identifier⟩ ';'

⟨assignment-statement⟩ =
    ⟨identifier⟩ '=' ⟨expression⟩ ';'

⟨declaration-assignment-statement⟩ =
    ⟨identifier⟩ ⟨identifier⟩ '=' ⟨expression⟩ ';'

Control Structures

The language support three control structures; if statements, while and foreach loops.

If Statement

If statements uses an boolean expression to determine if the statement block inside the if statement should be executed. If the boolean expression evaluates to false, the statement block is not executed. An *if statement* can have a list of additional if statements – *else if statements* – that are only be checked if the original *if statement* did not execute its statement block. If neither the *if statement* or any of the *else if statements* are executed, the statement block in an *else statement* is executed.

The syntax of an *if statement* is:
⟨if-statement⟩ =
    'if' '(' ⟨expression⟩ ')' '{' ⟨method-body⟩ '}' |
    'if' '(' ⟨expression⟩ ')' '{' ⟨method-body⟩ '}' ⟨else-statement⟩

⟨else-statement⟩ =
    'else if' '(' ⟨expression⟩ ')' '{' ⟨method-body⟩ '}' ⟨else-statement⟩ |
    'else' '{' ⟨method-body⟩ '}'

While Loop

A *while loop* is a control structure, where a statement block is executed as long the boolean expression associated with the *while loop* evaluates to true. The boolean expression is evaluated before executing the statement block. After the statement block has been executed, the expression is evaluated again to determine if another iteration of the loop should be executed.

The syntax of a *while loop* is:
⟨while-statement⟩ =
    'while' '(' ⟨expression⟩ ')' '{' ⟨method-body⟩ '}'

Foreach Loop

A *foreach loop* is a control structure that iterates over a collection such as an argument list, List, or HashTable. As the *foreach loop* iterates over the collection, each item in the collection is assigned to a variable that can be used in the statement block inside the *foreach loop*.

The name of the collection that should be iterated over, and the name of the variable that each element should be assigned to, is specified in the declaration of the *foreach loop* as shown in the syntax:

⟨foreach-statement⟩ =
    'foreach' '(' ⟨identifier⟩ 'as' ⟨identifier⟩ ')' '{' ⟨method-body⟩ '}'

Listing 3.3 show an example of how a foreach loop iterates of the elements of an list.

```
1 List l;
2 l.add(1);
3 l.add(2);
4 foreach (l as e) {
5     NativeInterface.print("Element: " + e);
6 }
```

Listing 3.3: Iterating over a list collection.

Join-Point Context

All of available context information is accessible through a single context object, which has the same name as the LMV bound in the pointcut, in order to provide a consistent way of accessing join-point context. In order to make it more explicit when the context is accessed, the context object must be referenced with the `?` prefix. For instance, if the pointcut for a given advice binds the LMV *M*, it would be accessed as *?M* inside the advice body. The context object contains the following properties:

name The name of the method as a string.

arguments A list of all arguments of the target method. Each argument object again contains properties for value and type of the argument.

returnType The methods return type as a string.

className The name of the class where the method resides in as a string.

Name, arguments, return types and class name are also available in most other AOP languages. Since these properties may be useful in advices, for instance in the condition block of a narrows advice, we have chosen to include these in our language. The argument list is a read-only list, but is otherwise identical to the list data type. An example of how to use join-point context in an advice is given below.

```
1 foreach(?M.arguments as argument) {
2     string type = argument.type;
3     if (type == "int") {
4         int value = argument.value;
5     }
6 }
```

Returning Values

An advice returns a value with the same type as the one that the intercepted method has. If the intercepted method is called using the *proceed* statement, the return value from the intercepted method is stored in an object named `returnValue` on the LMV. If the developer does not wish to change the return value, then it is returned implicitly at the end of the advice without having used a return statement.

```
1 pointcut target := "class members[M] { public int 'getBalance }"
2
3 advice tracing: around target {
4     NativeInterface.log("Entering " + ?M.name);
5     proceed();
6     NativeInterface.log("Leaving " + ?M.name);
7 }
```

A value can be returned explicitly from an advice by using an *return* statement. The return statement can be used to return a new value if the intercepted method violates a post-condition, or a value can be returned before any *proceed* call if it is not necessary to call proceed.

```
1  pointcut target := "class members[M] { public int 'fib (int) }"
2
3  HashTable results;
4
5  advice fibCache: around target {
6      if (results.contains(?M.arguments[1])) {
7          return results[?M.arguments[1]];
8      } else {
9          int result = proceed();
10         results.add(?M.arguments[1], result);
11     }
12 }
```

If an abnormal state is reached inside an advice e.g. the pre or post conditions are not met, then it is possible to abort the execution and raise an exception by calling the *abort* statement. The abort statement aborts execution by throwing an runtime exception.

```
1  pointcut target := "class members[M] { public int 'getBalance }"
2
3  advice authentication: narrows target {
4      condition {
5          NativeInterface.authenticated();
6      }
7      success {
8          proceed();
9      }
10     failure {
11         abort();
12     }
13 }
```

An advice is required to return a value – implicitly by having a proceed call or explicitly – or call the *abort* statement in all code paths to ensure that the advice always exits correctly. If an advice does not return a value or exception on all code paths, the compiler gives an error.

3.11 Summary

The overall design of our language is minimalistic and focuses on the pointcut and advice concepts. While this means that some common AOP features such as introductions and control-flow were not included, our language is still a complete AOP language. The improvements that were made with regards to pointcuts and advices should enhance reusability and also make users want to reuse aspects. Future iterations of this language should likely concentrate on adding more features to the advice language, and thus making the language even more easy to use.

CHAPTER 4

Weaver Design

This chapter presents the design of the weaver and its sub-components. The first section presents an overview of the weaving process and what components are involved. The second section presents the design of the different components. Finally we present the design of the command line interface for the weaver executable and how it should be used by a developer.

4.1 Weaver Process

The aspect weaver is equivalent to a normal compiler with the exception that the weaver does not create a new executable program, but the output code is woven into an existing executable. The result is a modified version of the target program. In order to do this the weaver works the same way like a complete compiler, only with the added step of weaving the generated code into a target program.

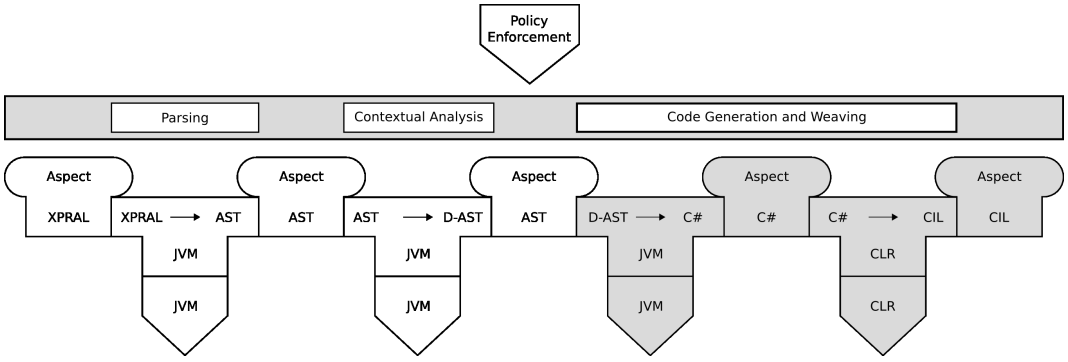


Figure 4.1: Tombstone diagram of weaver for the .NET platform with component names at the top. Grayed elements indicate platform dependent components.

The aspect source files go through different representations during the weaving process. Some of the representations are dependent on the platform that the aspect is to be weaved on. The diagram in Figure 4.1 is a combination of a tombstone diagram and a simple flow diagram, and it show the different

steps of the weaving process and what representations are used in the different steps. This process is described in more detail in the remainder of this section.

Syntactical and Contextual Analysis The first two steps of the weaving process is the syntactical and the contextual analysis, which we do not want to describe since it works just like in any other compiler. The end result after these two steps, which is a list of all the aspects, is sent to the next step.

Aspect Inheritance In the third step, all concrete aspects that extend an abstract aspect have the inherited members merged from the abstract aspect. Concrete and abstract aspects are merged instead of linked together because we assume that there will usually only be one implementation for each abstract aspect. Merging aspects will thus not waste much memory and perform better, in addition to being a simple solution. A list of concrete aspects is sent to the next step.

Policy Enforcement The fourth step is enforcement of policies. Before modifying the target program, all policies are evaluated. If a policy is violated the weaving process is aborted and reason for the policy violation is displayed. If no policies are violated, then the list of all concrete aspects is sent to the next step.

Aspect Weaving When all aspects have been parsed and checked against the contextual constraints and policies they are weaved into the target program. The actual weaving consists of two steps: weaving of the singleton aspect class into the target program, and weaving the advices into the methods they advise.

A singleton class is created for the aspect for the target language of choice, and then compiled. The singleton aspect class contains all of the methods and fields of the aspect. If a native interface has been declared, the singleton object also contains an object with the type of the native interface class. The fields, methods, and the native interface object can be accessed from within advices and methods through the `getInstance()` method on the singleton class. Once it is compiled it is then woven into the target program. The name of the generated singleton class is sent to the next step together with the list of advices and pointcuts in the current aspect.

For each advice in an aspect, the list of all join-points that the pointcut associated with the advice is found by evaluating the pointcut using JTL. For each join-point in the result list, an instance of the advice is weaved into the target method associated with the join-point.

In order to weave an advice, the target method in the target program is renamed by adding the `old_` prefix. If a method already exists with the `old_` prefix, a unique name is generated that is used instead. The advice method from the aspect class is then moved into the target program, replacing the original method. Any calls to the proceed method are finally changed to calls to the renamed original method.

4.2 Weaver Components

As described in the previous section, the weaver consists of a lexer, parser, contextual analyzer, combined code generator and weaver component. The components are realized as the classes shown in Figure 4.2. Since the weaver must handle multiple platforms, the weaver architecture is modularized, so that any step involving platform-dependent operations can use different components depending on the the target platform. This means that a new target platform can be added by only implementing the platform specific components and without having to change the platform independent components.

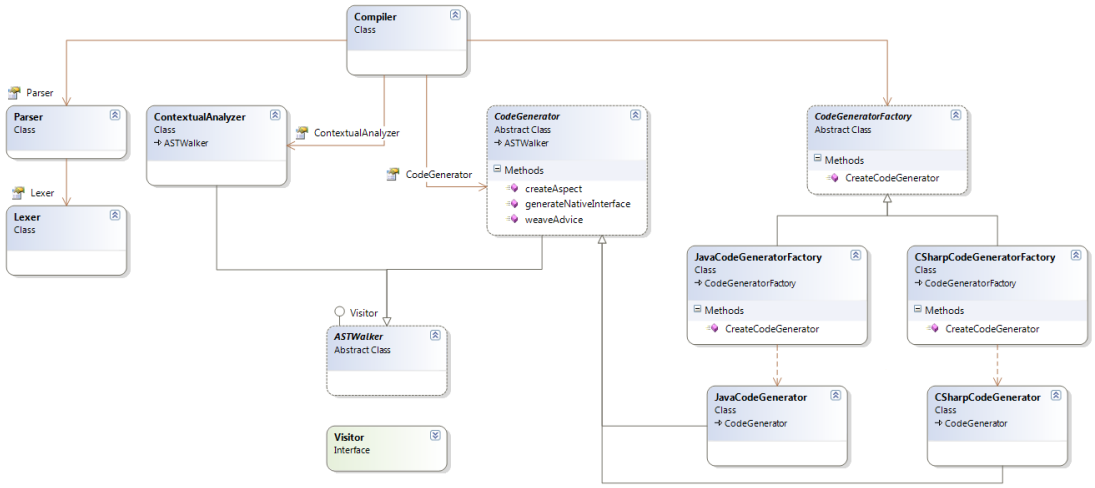


Figure 4.2: Weaver class diagram

The weaving process is controlled in the main method of the `Compiler` class, that uses the different components – parser, contextual analyzer, and code generator – as properties. The `Parser` class reads individual tokens with the help of the `Lexer` class and uses these to create an Abstract Syntax Tree (AST) of the input aspect.

The actual AST consists of several classes in the `AST` namespace – one class for each of the productions in the grammar. The `AST` namespace also contains the `Visitor` interface that is used to implement the visitor design pattern on top of the AST. The `Visitor` interface is implemented by the abstract `ASTWalker` class. The implementation in this class provides in and out methods for each node, and calls the visit method of each node and any sub nodes. The `ASTWalker` class in turn is extended by the `ContextualAnalyzer` and `CodeGenerator` classes.

Finally the `CodeGenerator` class is an abstract base class for the code generators for each of the target platforms. It provides a set of utility methods that are useful for code generation, e.g. a set of methods to determine the correct data types for identifiers. The actual code generation is described in the corresponding section for each supported platform in Chapter 5. An instance of the `CodeGenerator` class can be instantiated using the abstract class factory design pattern implemented by the `CodeGeneratorFactory` class and its derived classes – `JavaCodeGeneratorFactory` and `CSharpCodeGeneratorFactory`.

JTL Pointcut Evaluation

The JTL library is used as a library by the `Compiler` class to evaluate pointcut expressions in order to find elements violating a policy and methods intercepted by an advice. Since the JTL engine was written for the Java language, it requires some modifications in order to support multiple languages. We begin by describing how the original and unmodified version of JTL evaluates pointcuts, before discussing the required changes in order to make JTL capable of evaluating predicates for multiple languages.

Overview of the JTL Engine

In order to understand the modifications that need to be made to the JTL engine it is important to understand the overall architecture of the engine, as well as the workflow for the evaluation of predicates. This section discusses the evaluation of predicates in an unmodified release of JTL using version 0.1.13..

On startup JTL is supplied with an argument, specifying the input program. When JTL evaluates a query it first looks up all classes that are present in the input program and stores their fully qualified name in an array for later use. The predicate expression is then evaluated by evaluating each of the native predicates individually.

In order to evaluate a native predicate that does not operate on class names alone – such as the `is` predicate – it is required to determine the set of fields and methods of a given class, and often the instructions contained in the methods. This is done by creating a model of the class, containing representations of the fields and methods in it. In version 0.1.13 of JTL these class models consist of both Bytecode Engineering Library (BCEL) as well as Java reflection classes, because the first versions only used reflections and BCEL was only added later. The Java reflection classes are used to reflect over the general structure of the class, while the BCEL classes are used to reflect over instructions in methods and constructors.

The actual evaluation of a native predicate is performed by traversing the class models for all input classes. Matches are stored in a hash map for the predicate, with class or method signature as key and the value being a list of all the matches for the given class or method. Any program element that matches the type of the predicate is considered a match, not only those that match any supplied arguments. For instance, the `reads` predicate will store *all* field read instructions for a given method in the output list, not only those that match the supplied argument. Only when each of the native predicates has been evaluated individually are the exact matches for the query calculated, and the results returned.

Cross-platform Layer

Since the unmodified JTL engine uses a Java-specific model of classes in form of reflections and BCEL classes, it is not well suited to handle multiple platforms by default. In order to overcome this problem we need to introduce a platform independent layer between predicate evaluation and the generation of class models. The platform independent layer should consist of a package with abstract classes representing the various program elements, such as classes, methods, and instructions. The package should also contain an abstract class-factory class which can be used to obtain representations of specific classes. Each supported target platform then requires an implementation of these abstract classes, possibly using platform specific bytecode engineering libraries or by invoking external applications.

The generic platform-independent package should also contain a factory singleton class, providing access to and instantiating the correct class factory object based on the current target platform. The factory class should determine the correct target platform, based on a command line argument.

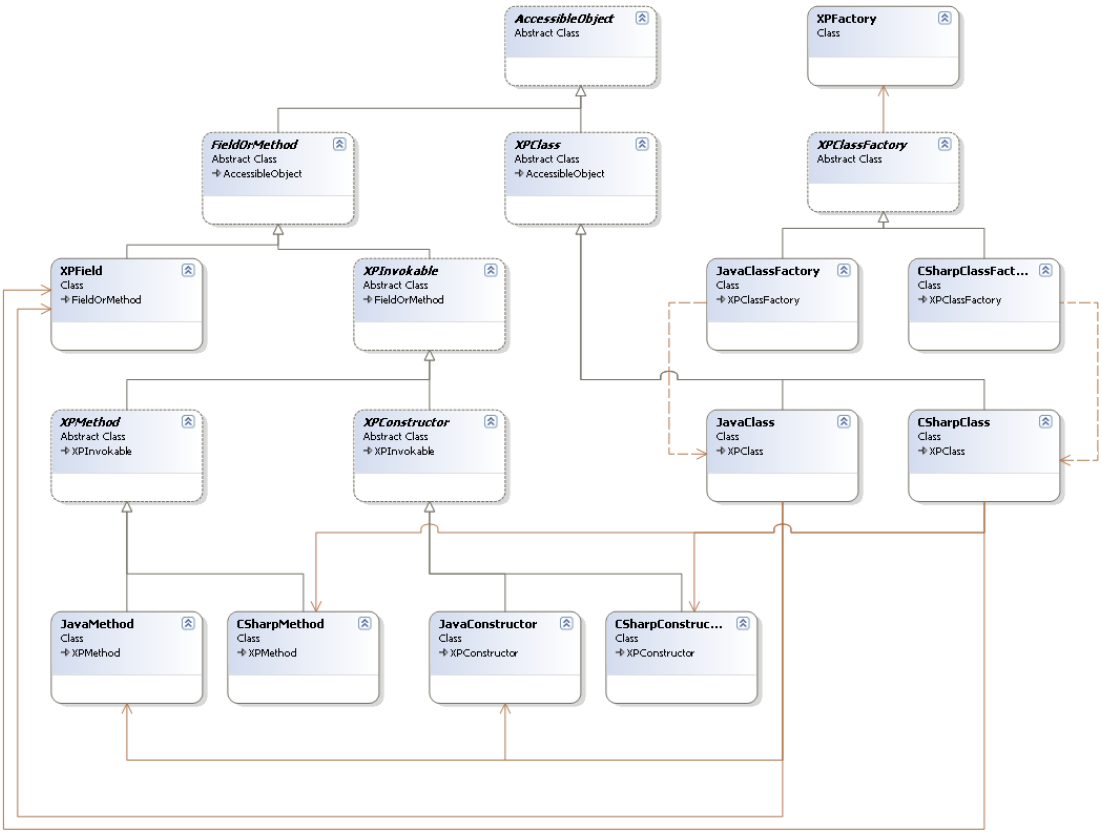


Figure 4.3: JTL class diagram

A simplified version of the class diagram for the cross-platform layer, which shows none of the instruction related classes, is shown in Figure 4.3. The class diagram illustrates the general process of providing abstract classes for different program elements and implementing them for each target platform. It also shows the design of the class factory pattern which is used to instantiate class objects.

4.3 Using the Weaver

The weaver must support two basic tasks: extracting a native interface declaration from an aspect, and weaving aspects into a target program. The following sections present the command line interface of the weaver executable, and examples of how the weaver is used to perform specific tasks.

Command-Line Interface

The command line interface of the weaver contains four switches that allow the user to specify the input to the weaving process. The general command line used to invoke the weaver is as follows:

```
java Compiler -language:<language-identifier> [OPTIONS] ASPECT_FILE
```

Where optional arguments can be any of the following:

-target:<target-file> Specifies the target program. In order to include several assemblies or class files, this argument may be specified multiple times.

-nativeclass:<native-class-file> This argument specifies the class name of the class implementing the native interface. The class must be contained in one of the files specified with the `target` argument.

-generatenativeinterface If this argument is specified the weaver will not actually weave any aspects, but instead extract a native interface declaration into source code format for the selected language.

The `-language` argument is obligatory, since it is required for both weaving of aspects as well as generating interface classes. Specifying an aspect file is also obligatory since the weaver can do nothing without an aspect.

Workflow Example

This section discusses an example of how the weaver can be used in the development process. Assuming that the developer is standing in the root of his Java-based target project, the developer can extract a native interface into a Java interface from an authentication aspect placed in the file `auth.xpa` by executing the following command:

```
java -jar weaver.jar -language:java -target:. -generatenativeinterface auth.xpa
```

This will extract the required native interface as source code and place it in the same directory. The developer can now create a new class implementing the interface. Once the interface has been implemented in the `NativeBridge` class, the developer can now use the weaver to weave the aspect in the following way:

```
java -jar weaver.jar -language:java -target:. -nativeclass:NativeBridge auth.xpa
```

In Chapter 6 we present more examples of how the user can be used.

CHAPTER 5

Weaver Implementation

This chapter presents the implementation of our weaver. The weaver is built like a compiler, with the difference that instead of compiling code to a new executable, the code is woven into an existing executable. We do not discuss those parts that are common in compilers, such as parsing and contextual analysis. Instead we focus on describing how we achieve support for both Java and C# as target languages.

It should be noted that as part of our weaver implementation we implemented the cross-language layer in JTL as discussed in Section 3.7. We also implemented all required native predicates that were not already present in the original JTL release.

5.1 Java Support

The Java implementation of the cross-platform package wraps the BCEL classes and redirects method calls to the wrapped BCEL objects. We upgraded to version 5.2 of BCEL which provides some additional functionality to traverse class and method hierarchies, making it somewhat easier to remove the dependency on the reflection classes.

The class factory loads classes only on demand and stores them in a hash map after creation, in order to prevent unnecessary class loading operations. When the class factory is instantiated the path to the system libraries is added to any classpath arguments the user may have provided, so that JTL is able to find program elements that are in the system library.

The set of input classes is determined by scanning the classpath for all classes. The classpath consists of any paths the user has specified when invoking the weaver, as well as the path to the system libraries.

Weaving

After the AST for the aspect has been generated, the first step is to generate source code for the aspect singleton class, containing fields, methods, and the native interface object, if one has been defined. The source code is compiled with the default Java compiler and placed in a package called `aspect`. In addition to the aspect class file, code for the native interface as well as the argument class is generated and placed in the `aspect` package.

After creating the aspect singleton class, the individual advices are woven in for each target method. Source code for the target method is created and placed in an empty class definition together with an empty proceed method. This class is compiled and loaded with BCEL in order to move the advice method into the target class file, which is also loaded with BCEL. The original method is renamed by adding `old_` as a prefix, and the advice method in inserted in its place with the name of the target method. Any calls to the proceed method in are replaced with calls to the renamed target method. If the target method is not a void method, a store instruction is added to store the result of the call in a new local variable, which is also loaded just before the return instruction. If there is no return instruction, or the user explicitly returns another value, the variable is ignored.

Finally, the modified class object is written to the original class file.

5.2 C# Support

Compared to the Java platform support, the support for the C# language has to go through an extra step in order to obtain the OOP model from the target program, and in order to weave the aspects into the target. The support for C# uses two external tools – written in C# – to reflect and manipulate .NET assemblies, instead of accessing the target program directly from the .NET modules written in Java. The `Mono.Cecil`[1] .NET class library is used in the two C# programs for both reading the OOP model and for performing the weaving of aspects into the target program.

[1] `http://www.mono-project.com/Cecil`

JTL Integration

As described in Section 4.2 JTL needs access to a list of all classes present in the input program as well as all system classes. The class list is read by executing the external C# program with the input classes as arguments.

The C# program creates an XML-document, and then appends an XML-node for each assembly. XML-nodes for classes are created based on information read from the assembly, and the classes methods and fields are added to the class node. Common Intermediate Language instructions are read from the method bodies and the opcode and operand are added to the method nodes.

After all classes in the input assemblies have been read, the system classes in the .NET framework are read as well. To increase performance, instructions are not read from methods in system classes.

When all classes are added to the XML-document, the text representation of it is written to standard out, so that the JTL code can read the document and parse it. Instances of JTL types for classes, methods, fields, and instructions are created based on the XML-document.

Weaving

Weaving of aspects into the target .NET assemblies begins with generating C# source code for a singleton class containing the aspect fields and methods. The C# source code is sent to a .NET program, which compiles the source code into a .NET assembly. The compiled .NET assembly is opened using the `Mono.Cecil` library, and the class is copied into the assembly of the target program.

For each method adviced by an advice, a temporary C# class is generated. The class contains a method containing the advice body with the same signature as the adviced method, and a method named `proceed` with the same signature. `proceed` statements are inserted as calls to the `proceed` method in the temporary class, with the methods arguments sent directly to the proceed method.

The generated C# class is compiled into a temporary assembly, which is opened using the `Mono.Cecil` library. The target method is found in the target assembly, and a copy is made of the method and inserted into the assembly named `old_MethodName` if the method is named "MethodName".

All instructions in the target method are removed, and the instructions from the advice method are inserted into the target method. The call instruction to the proceed method in the temporary class is changed to call the renamed target method.

When the advice has been woven into the target assembly, the assembly is saved.

5.3 Missing Features

This section discusses the features from our design which we have not yet implemented.

Harmless advice The harmless advice is currently not implemented. Since our current advice language does not offer any form for I/O, the harmless advice would still be useless if it were implemented, as it could not be used for any purpose at all. For instance, tracing is an example for a harmless advice, but since it is not possible to report the trace in any way, it would be useless.

Ruby and Python Support The generalized OOP model we discussed in Section 2.4 included Java, C#, Ruby, and Python. The implemented weaver only has support for the Java and C# languages, and thus support for Ruby and Python still needs to be implemented.

We actually have a partial implementation for Ruby, using JRuby[2], but we did not have time to finish the implementation and decided to not discuss it any further. We assume that Python support could be implemented using the Jython[3] class library, but have not tested it further.

In both the Ruby and Python implementation, the native interface declarations would need to be extracted into a class stub instead of an interface declaration since the two languages does not support interfaces. The respective implementations of the cross-language layer in JTL should be implemented to only read elements that can be mapped into our generalized OOP model.

Requires Policy The requires policy is not yet implemented due to time constraints.

[2] `http://jruby.codehaus.org`
[3] `http://www.jython.org/`

List and Hashtable The `List` and `Hashtable` data types could not be implemented due to time constraints

Foreach The `foreach` control structure was not implemented due to time constraints.

Weaving The current weaver implementation always uses the `old_` prefix to renamed advised methods. The implementation should adhere to the design and prevent naming collisions by using different prefixes, in case a method with the prefixed name already exists. In addition to this the weaver does not weave advices into the existing executables or assemblies, for instance the java weaver creates new class files instead of weaving into an existing jar file.

CHAPTER 6

Demonstration of Language and Weaver

This chapter presents two complete examples of how aspects are implemented in our language and how the weaver is used during the development workflow. Language independent parts of the examples are implemented on all the language platforms supported by the weaver. The description of the first example is organized to fit the process described in Chapter 4 - Weaver Design in order to make the process more clear:

- Development of the aspect
- Extracting the native interface
- Weaving the aspect

The second example is discussed in less detail in order to focus on important parts, which are the concern based nature of the aspect and its pointcut.

6.1 Example One - Bank Application

The first example is a bank application that is used by bank tellers to perform money transactions for customers that walk in from the street. The application allows the teller to deposit to or withdraw from an account, and transfer money to another account.

The class diagram in Figure 6.1 shows the classes of the bank application. The `Bank` class contains the database of customer accounts and user credentials for bank employees. The `Account` class identifies an account by its account number and stores the current balance. The account can be manipulated using the `Deposit` and `Withdraw` methods. The `User` class is an abstract class that identifies a bank employee by a username and password. The `Teller` and `Supervisor` classes are both concrete classes that extend the `User` class, and they are used to differentiate between what privileges the user has.

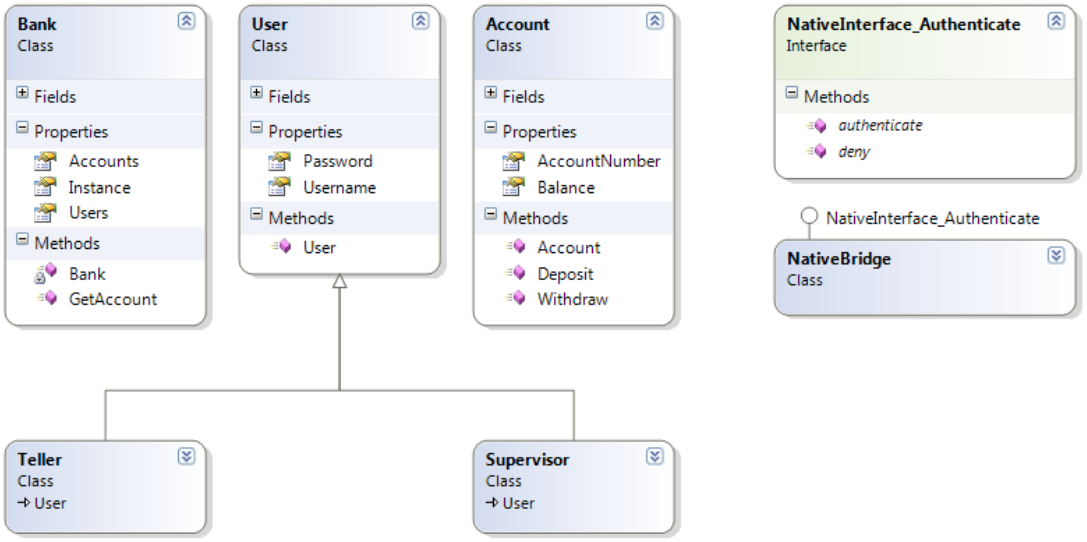


Figure 6.1: Bank application - Class diagram

An aspect is used to improve the functionality of the bank application. The *Authentication* aspect will be described in the following section. The aspect contains a native interface that is used to communicate with the bank application, and the interface is implemented by the `NativeBridge` class.

6.1.1 Authentication Aspect

To avoid fraud, an upper limit is placed on how much tellers can withdraw from a bank account. If the teller attempts to make a large withdrawal – above $1,000 – a supervisor has to the approve the transaction, otherwise the transaction is aborted.

This upper limit on withdrawal transactions is implemented using an authentication aspect – shown in Listing 6.1.

```
1  aspect Authentication {
2      NativeInterface {
3          bool authenticate();
4          void deny();
5      }
6
7      bool isAuthenticated = false;
8
9      pointcut withdrawMethod[C] := "method 'Withdraw member_of[C]";
10     pointcut targetMethods[M]:= "class 'BankApplication.Account is[C] members[M]
           M.withdrawMethod[C]";
11
12     bool askForAuthentication() {
13         if (NativeInterface.authenticate()) {
14             isAuthenticated = true;
15             return true;
16         }
17         return false;
18     }
19
20
21     advice largeWithdrawals: narrows targetMethods {
22         condition {
23            ?M.arguments[0].value < 1000 || isAuthenticated ||
                  askForAuthentication()
24         }
25
26         success {
27             proceed();
28         }
29
30         failure {
31             NativeInterface.deny();
32         }
33     }
34 }
```

Listing 6.1: Authentication aspect.

The authentication aspect contains a native interface declaration with two methods. `authenticate()` prompts for the approval of a supervisor and returns a boolean value representing if the approval was given. `deny()` displays an error message to the teller that since the approval was not given, the transaction can not be completed.

The aspect field `isAuthenticated` is used to store if the teller was previously given an approval. The pointcut `targetMethods[M]` captures all calls to the `Withdraw()` method on the account class.

The narrows advice `largeWithdrawals` uses the boolean expression in the narrowing condition to check if the withdrawal should be allowed. The withdrawal is allowed if the requested amount is below $1,000, if the teller has previously received an approval, or if the supervisor gives his approval.

If the condition evaluates to true, the success block of the narrowing advice is executed. The `Withdraw()` method is called using the `proceed()` statement.

If the condition evaluates to false, the failure block is executed. Using the native interface method `deny()`, the teller is informed that the withdrawal is not allowed, and the transaction is aborted.

Since the `Withdraw()` method returns void it is not necessary to return a value or call the `abort()` statement in either the success or failure blocks.

6.1.2 Extracting a Native Interface

The authentication aspect contains a native interface, with two methods as seen in Listing 6.1. The native interface can be extracted from the aspect declaration be executing the weaver with the `-generatenativeinterface` option. Listing 6.2 show how the interface is extracted from the aspect into a Java interface.

```
$ java -jar weaver.jar -generatenativeinterface -language:java auth.xpa
```

Listing 6.2: Command line to extract the native interface into a Java interface.

The extracted interface for each of the supported platforms in shown in Listing 6.3 and Listing 6.4. The interfaces are implemented in the bank application in the class `NativeBridge` (shown in Appendix C), and provided to the weaver when weaving the aspect into the application.

Java interface

```
1 interface Authentication_NativeInterface {
2     boolean authenticate();
3     void deny();
4 }
```

Listing 6.3: Native interface extracted into a Java interface.

C# interface

```
1 interface Authentication_NativeInterface {
2     bool authenticate();
3     void deny();
4 }
```

Listing 6.4: Native interface extracted into a C# interface.

6.1.3 Weaving the Aspect

When the aspect has been implemented, and the native interface implemented in the `NativeBridge` class for the target platform, the next step is to weave the aspect into the bank application. Listing 6.5 show the command used to weave the aspect into the Java bank application.

```
$ java -jar weaver.jar -language:java -target:BankApplication.jar -nativeclass:
    BankApplication.NativeBridge auth.xpa
```

Listing 6.5: Command line to weave the Authentication aspect into the bank application.

The weaver first generates the singleton aspect class `Authentication` shown in Listing 6.6, and introduces the class into the bank application.

```
1  package aspect;
2
3  import BankApplication.NativeBridge;
4
5  public class Authentication {
6
7      private Authentication() {
8          isAuthenticated = false;
9          nativeInterface = new NativeBridge();
10     }
11
12     public static Authentication getInstance() {
13         if(instance == null)
14             instance = new Authentication();
15         return instance;
16     }
17
18     public Authentication_NativeInterface getNativeInterface() {
19         return nativeInterface;
20     }
21
22     public boolean askForAuthentication() {
23         if(getInstance().getNativeInterface().authenticate()) {
24             isAuthenticated = true;
25             return true;
26         } else {
27             return false;
28         }
29     }
30
31     private static Authentication instance;
32     Authentication_NativeInterface nativeInterface;
33     public boolean isAuthenticated;
34 }
```

Listing 6.6: Generated Java class for the *Authentication* aspect.

The `Withdraw` method in the `Account` class is renamed to `old_Withdraw`. The weaver generates a method for the advice `largeWithdrawals`, and it is into the

`Account` class as the `Withdraw` method with a reference to the original method. The resulting `Account` class is shown in Listing 6.7.

```
1  package BankApplication;
2
3  import aspect.*;
4  import java.io.PrintStream;
5
6  class Account {
7
8      public Account(int i, double d) {
9          accountNumber = i;
10         balance = d;
11     }
12
13     public int getAccountNumber() {
14         return accountNumber;
15     }
16
17     public double getBalance() {
18         return balance;
19     }
20
21     public void Deposit(double d) {
22         balance += d;
23         System.out.println((new StringBuilder()).append("Depositing: ").append(d)
               .append(". New balance: ").append(getBalance()).toString());
24     }
25
26     public void old_Withdraw(double arg0) {
27         balance -= arg0;
28         System.out.println((new StringBuilder()).append("Withdrawing: ").append(
               arg0).append(". New balance: ").append(getBalance()).toString());
29     }
30
31     public void Withdraw(double d) {
32         MethodArgument amethodargument[] = {
33             new MethodArgument("double", Double.valueOf(d))
34         };
35         if(((Double)amethodargument[0].getArgumentValue()).doubleValue() < 1000D
               || Authentication.getInstance().isAuthenticated || Authentication.
               getInstance().askForAuthentication())
36             old_Withdraw(d);
37         else
38             Authentication.getInstance().getNativeInterface().deny();
39     }
40
41     private int accountNumber;
42     private double balance;
43 }
```

Listing 6.7: Resulting *Account* Java class after weaving of the *largeWithdrawals* advice.

6.1.4 Output

The example application is tested with one account holding $5,000, and we will execute the following test:

- Open Account #1
- Withdraw $2,000 (should cause a login prompt to appear, because it is a large amount of money, and thus requires supervisor approval)
- Withdraw $1,500 (should use the previous approval and thus complete).

Listing 6.8 show approval process of the Java bank application. The complete output of the Java application demonstration is shown in Appendix B.

```
1 Enter amount you want to withdraw: 2000
2 ========================================
3 Your action requires supervisor approval.
4 ========================================
5 Username: boss
6 Password: 1234
7 Withdrawing: 2000.0. New balance: 3000.0
```

Listing 6.8: Part of the console output from the Java bank application. The supervisor is required to approve the large transaction of $2,000.

Listing 6.9 show a withdrawal attempt that fails, because of an incorrect approval.

```
1 Enter amount you want to withdraw: 2000
2 ========================================
3 Your action requires supervisor approval.
4 ========================================
5 Username: boss
6 Password: 4321
7 This transaction requires supervisor approval, transaction denied
```

Listing 6.9: Part of the console output from the Java bank application. The supervisor approval was incorrect and the withdrawal fails.

6.2 Example Two - Figure Editor

The traditional AspectJ figure editor[16] example is our second example. The example application consists of a display class `Display`, which should be redrawn every time the position of figure elements is updated. The `Point` and `Line` classes extend the abstract `FigureElement` class and provide methods to change the position of the element. The different classes are presented in the class diagram in Figure 6.2.

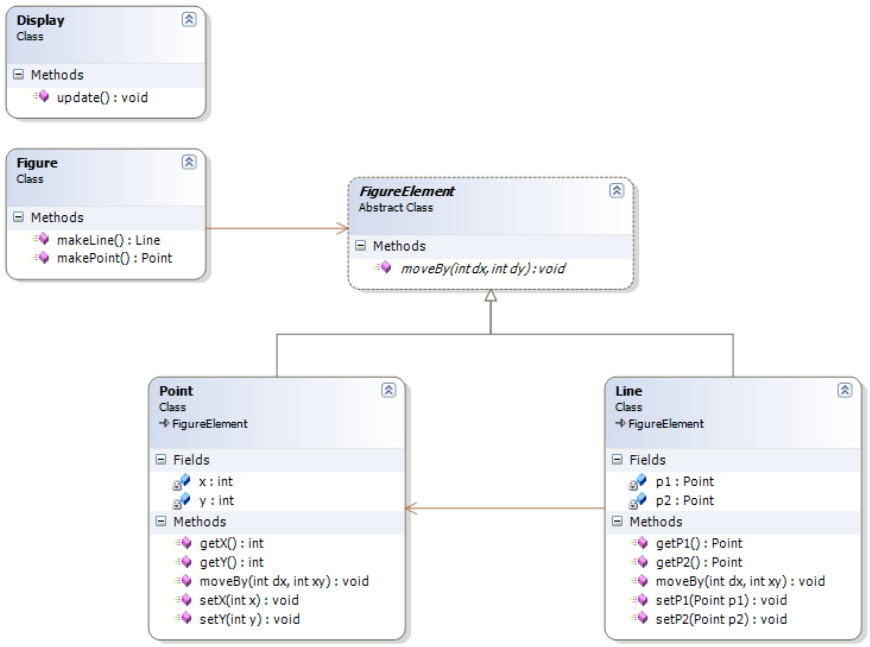


Figure 6.2: Figure editor - Class diagram

The traditional OOP solution to the issue of updating the display on changes of the positions would be to call `Display.update()` in all of the methods that modify the position fields – `X` and `Y` or `P1` and `P2`.

The AspectJ aspect *DisplayUpdating* shown in Listing 6.10 contains an enumeration of the methods that change the position of a figure. After one the methods have been called the aspect call the `Display.update()` method. This aspect relies on explicit enumeration of methods and would thus break if the application was extended with more figure elements.

```
1  aspect DisplayUpdating {
2
3    pointcut move():
4      call(void FigureElement.moveBy(int, int)) ||
5      call(void Line.setP1(Point))              ||
6      call(void Line.setP2(Point))              ||
7      call(void Point.setX(int))                ||
8      call(void Point.setY(int));
9
10   after() returning: move() {
11     Display.update();
12   }
13 }
```

Listing 6.10: AspectJ example of the DisplayUpdating aspect.

Our language allows the aspect to intercept methods based on their behavior instead of based on names. The aspect in Listing 6.11 intercepts methods that writes to a local field that are in turn read by the `Display.update()` method. These methods are matched by the `move` pointcut. The `repaint` around advice calls the `Display.update()` method through a native interface.

```
1  abstract aspect DisplayUpdating {
2      NativeInterface {
3          void displayUpdate();
4      }
5
6      abstract pointcut updateMethod[F];
7      abstract pointcut projectClass[M];
8      pointcut move[N] := "class members[M] M.updateMethod[F] F.member_of[B] B.
           projectClass[N] N.writes[F]";
9
10     advice repaint: around move {
11         proceed();
12         NativeInterface.displayUpdate();
13     }
14 }
```

Listing 6.11: Our example of the DisplayUpdating aspect using behavioral pointcuts.

The aspect contains an abstract pointcut `updateMethod` that is used to describe a specific update method in the target program. The abstract pointcut `updateMethod` is implemented in Listing 6.12. A policy is added to the concrete aspect to make it enforce that developers of the base-level program do not address the concern manually in the base code. The policy `manualUpdates` enforces that methods that change the position can not manually call update method.

```
1 aspect FigureDisplayUpdating extends DisplayUpdating {
2     pointcut updateMethod[F] := "method 'update member_of[/FigureEditor.Display]
          invokes[M] M.reads[F]";
3     pointcut projectClass[M] := "class 'FigureEditor.?* members[M]";
4
5     pointcut manualUpdate[M] := "class move[M] M.invokes[U] U.updateMethod[_]";
6     policy manualUpdates forbids manualUpdate: "Figure elements are not allowed
          to manually update the display.";
7 }
```

Listing 6.12: Concrete aspect that specifies the update method.

This aspect is made reusable by the fact that any updates to fields read by the update method are automatically intercepted. The OOP developer can thus add more figure element types without having to extend the concern.

The Pointcuts Explained

The pointcuts used in the Figure Editor example are relatively complex and are thus discussed in the following paragraphs. The most important pointcut is the *move* pointcut, as that is the pointcut that is used in the *repaint* advice. The move pointcut depends on two other pointcuts, *updateMethod* and *projectClass*.

```
pointcut updateMethod[F] := "method 'update member_of[/FigureEditor.Display]
    invokes[M] M.reads[F]";
```

The updateMethod pointcut captures the project specific update method and defines that the method defined as the subject invokes another method which reads the field that is added as a parameter to the pointcut. This represents the invoking of a getter method, which returns the field that is read by the update method.

```
pointcut projectClass[M] := "class 'FigureEditor.?* members[M]";
```

The projectClass pointcut specifies that the class writing to the given field is located in the `FigureEditor` namespace. This prevents the pointcut from capturing methods in system classes, which also write to fields that are read by the update method.

```
pointcut move[N] := "class members[M] M.updateMethod[F] F.member_of[C] C.
    projectClass[N] N.writes[F]";
```

The move pointcut finally captures all target methods `N` , which are defined as being methods that exists in a project class and write to a field that is read by the update method.

6.3 Summary

In this chapter we have demonstrated how the weaver can be used to weave a single aspect into an application written in multiple languages. We have demonstrated that a concern based pointcut can be used to make an aspect reusable and that it solves the fragile pointcut problem. In the example aspects we have demonstrated the use of several of our language constructs, such as native interfaces, narrows advices, and policies. The native interface declaration was extracted into a Java and C# interface, implemented and used in the final aspect weaving.

CHAPTER 7

Future Work

In this chapter we discuss possible future improvements to our language, and new ideas which we have not explored already. We present extensions to the pointcut language, the advice language, and to two of the advice types.

7.1 Pointcut Language

Our current version of JTL and thus our pointcut language is relatively simple. This section discusses possible improvements in form of a new JTL version, as well as syntax modifications.

Updated JTL version

The full JTL query language introduced in [8] included many more features than those implemented in the 0.1.13 version available for use in this project. An implementation of the full language specification is scheduled to be completed in the summer 2007. Among the features included in the full language specification are full support for querying behavioral properties of methods, that are only implemented in the form of the `reads` and `writes` predicates. The extended support for behavioral properties include the ability to reason about how input and output in methods are used using data flow analysis.

Although the pointcut language would greatly benefit from this improved support for querying behavioral properties, some behavioral properties are not clear until at run-time, and a set of dynamically evaluated predicates could be added to handle these scenarios. We propose that predicates to check the join-point context at run-time are added to the pointcut language, and that those predicates are excluded from the query sent to the JTL evaluation engine an instead checked dynamically at run-time in all the methods in the result set.

The dynamic predicates could be used to check control flow properties, or check some state similar to the `if` pointcut designator in AspectJ.

Syntax Improvements

Due to limitations in JTL and our pointcut declaration syntax, the pointcuts in the Figure Editor example (see Section 6.2) need to be more complex than we had originally hoped. Improvements on these parts could make pointcuts much more readable and understandable. To recap, the Figure Editor example required the following `move` pointcut:

```
pointcut move[N] := "class members[M] M.updateMethod[F] F.member_of[C] C.members[
    N] N.writes[F]";
```

Our original vision of this pointcut was the following:

```
pointcut move[M] := "class members[M] M.writes[F] N.updateMethod[F]";
```

This pointcut captures the concept of a set of methods `M` writing to a set of fields `F`, which is read by an update method `N`. JTL does not accept a query using this pointcut, as JTL believes that the LMV `N` is unbounded. In order for JTL to understand the bounds for a LMV, the LMV must be supplied as an argument to a predicate somewhere in the query, as in the following example:

```
pointcut move[M] := "class members[M] M.writes[F] F.read_by[N] N.updateMethod";
```

This pointcut expresses the same idea as previously, but makes sure that JTL knows the bounds of all LMVs. This above examples shows one of the two possible solutions for the bounds problem. The first solution is to provide all native predicates in both directions, e.g. `M.reads[F]` and `F.read_by[M]`. This makes it possible to make the bounds of all LMVs clear to JTL, without sacrificing power of expression. The alternative is to improve the bounds checking in JTL to analyse and understand complex relationships as in our ideal version of the pointcut. Since we have implemented neither solution, we were forced to use the pointcut shown in the Figure Editor example.

Our pointcut declaration syntax could be improved by introducing a predicate concept in addition to the pointcut concept. While pointcuts always must have exactly one argument, representing the join-point, predicates would allow any number of arguments but could not be used as actual pointcuts. Instead they would be used as helper predicates in actual pointcuts, allowing for greater freedom in expression concerns, which should improved understandability.

7.2 Advice Language

The current version of our advice language is very simplistic, as we only required functionality to write simple aspects. This section discusses how we believe that our advice language can be improved.

Expressive Power

Future iterations of our language should provide an advice language whose expressive power is close to mature OOP languages, such as Java and C#. For instance, our advice language currently does not provide the `for` and `switch` control structures. One important feature that needs to implemented is a form of object model. Our current advice language is limited to working with primitive types and a simple form of list and map type.

The object model should be used to access members on classes that are part of the join-point context, for instance arguments or return values.

Class Library

Our advice language should also feature an extensive class library with predefined classes in order to simplify development. The class library should contain classes like commonly used data types, utility classes and input/output classes. It should also be possible for other developers to create their own class libraries.

The class library could possibly be realized by creating a form of wrapper implementations for each target language, where each implementation wraps a class with the same functionality from the target language. This requires the implementation of a class or object model as described above, as well as means to redirect method calls to a language specific implementation. This approach could reduce development time for complex classes as long as the redirection mechanism is easy to use. Another benefit would be a clean integration into the target project, since it would not be required to create and add new implementations of classes that already are present in the framework for the target language.

Logic Meta-Variables in Advices

Kniesel and Rho [17] suggest that an AOP language should provide access to all LMVs that are specified in a pointcut, in the advice that makes use of the given pointcut. The authors provide an example of a pointcut capturing class posing between a base class and a derived class. In the example the LMVs for the derived class as well as constructor arguments are used to return instances of the derived class, whenever the constructor for the base class is called.

In our design phase we experimented with this idea, but were unable to determine how proper and useful values for these LMVs should be determined. Consider the following pointcut:

```
1 pointcut samplePointcut[M] := class members[M] M.method M.reads[F] X.Method X.
      writes[F];
```

This pointcut refers to all methods `M` that read a certain set of fields `F` , which in turn is written to by method `X`. The questions we raised were the following: What values should `F` and `X` have for each target method `M`? How should they be accessed? How can they be used, e.g. can a we invoke method `X`? Since we were unable to answer these questions in any satisfying way we decided to postpone this feature.

It appears that only elements that can be directly referenced from the target method can be used in an advice. Non-static elements, or elements outside of the scope of the target method are not always available during the execution of the advice body. Our conclusion is that this idea needs further analysis but may become useful in the future.

7.3 Advice Types

Some of our advice types can be further improved by adding additional syntactic sugar or by expanding their functionality. These ideas are described in the following sections.

Narrows Advice

The narrows advice type can be further improved by allowing developers to rely on default implementations of the `success` and `failure` blocks. If the developer does not specify a `success` block, the default implementation is used

instead, which consists of a simple `proceed` statement. Likewise the `failure` block could have a default implementation simply consisting of the `abort` statement.

Replacement Advice

We believe the replacement advice type can be further improved by allowing it to work on program elements other than methods. The replacement advice type could be used to replace fields or entire classes instead of methods,. For example, consider the following aspect:

```
1 aspect LegacyReplacement {
2     pointcut legacyClass = "class is[/java.lang.String]";
3     pointcut replacementClass = "class is[/aau.sw10.util.String]";
4
5     advice replacementAdvice : replacementClass replaces legacyClass;
6 }
```

This advice replaces the original String implementation with a new implementation. This sort of advice could be useful when maintaining or updating legacy applications, since it allows developers to modify the application without needing to modify the original source.

CHAPTER 8

Conclusion

Through this report we have documented that it is possible to create an AOP language that can weave generic and reusable aspects into programs written in multiple target languages, and also improve reusability for aspects and evolvability for the target program.

Our thesis was based on the idea that a concern based pointcut model should be able to work on several languages, as long as these languages were structurally similar. For this to work the pointcut mechanisms required a model that generalized the features of these languages.

In our analysis we have described a basic version of such an OOP model, based on several different OOP languages. Although the model does not fit all OOP languages, it fits those that we believe to be most important, namely Java, C#, Ruby, and Python. The OOP model includes the basic structural elements that these languages use. Although the model can be improved to better generalize several language specific features, it does demonstrate the concept.

Another requirement is that concerns must be universal across these languages, otherwise it is impossible to create a single aspect to address the same concern in different languages. Our assumption was that concerns could indeed be expressed in a universal manner, and we have not found any indications of the contrary throughout this project. However, without further analysis we cannot draw any final conclusion on this subject.

We based our pointcut language on JTL because it was publicly available, supported the features we required, and was a stand-alone query language, which made it easy to integrate. The full implementation of the language specification will increase the expressive power of the pointcut language significantly.

Our pointcut language can be used to make AspectJ-style pointcuts with a very similar syntax, and as such is no more difficult to use than AspectJ. The main purpose of our pointcut language is to make concern based pointcuts instead, where signatures play only a little part. Since AspectJ cannot be used

to express concern based pointcuts, we cannot compare the usability in both languages, but only state that our pointcut language offers a greater power of expression.

Using our pointcut language, it is possible to solve the *fragile pointcut problem* by capturing concerns instead of signatures. Combined with the proposition that concerns are universal, this means that the pointcut language should be able to capture concerns in any OOP language.

Apart from a pointcut language we required an advice language that could be translated into different target languages. We have shown a design and an implementation of a basic version of such an advice language. In order to overcome limited expressive power in advices, we have added the native interface concept. The native interfaces also serve the purpose of acting as a language independent interface to the target language. Our conclusion is that our advice language is simplistic, but achieves the overall goal of being generic and translatable into several target languages.

The genericity of our language enhances reusability and evolvability of aspects, as project or language specific bindings are kept to a minimum. In order to further encourage developers to reuse aspects, we decided to make our language more understandable by allowing developers to be more explicit of what is being done. In part this is achieved by introducing different advice types based on our analysis and creating explicit policy keywords.

To conclude on the thesis presented in our introduction, we have presented a solution to the *fragile pointcut problem* by using concern based pointcuts, and our generic advice language can be used to develop resuable aspects for multiple OOP languages. Compared to existing implementations, we believe that our pointcut language yields great power of expression, and that our advice language facilitates greater reusability due to its applicability on multiple languages.

Bibliography

[1] R. T. Alexander and J. M. Biemann, "Challenges of aspect-oriented technology," in *Workshop on Software Quality (WoSQ)*, 2002.

[2] R. Altman, A. Cyment, and N. Kicillof, "On the need for setpoints," in *European Interactive Workshop on Aspects in Software (EIWAS)*, 2005.

[3] S. Chiba and K. Nakagawa, "Josh: an open AspectJ-like language," in *3rd Int' Conf. on Aspect-Oriented Software Development (AOSD)*, 2004.

[4] C. Clifton and G. T. Leavens, "Spectators and assistants: Enabling modular aspect-oriented reasoning," Tech. Rep., 2002.

[5] C. Clifton, G. T. Leavens, and J. Noble, "Ownership and effects for more effectire reasoning about aspects," Tech. Rep., 2006.

[6] T. Cohen, J. Gil, I. Maman, and S. Cohen, "JTL wiki." [Online]. Available: http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/JTL

[7] T. Cohen, J. Gil, and I. Maman, "Extending JTL to other languages," 2006. [Online]. Available: http://www.cs.technion.ac.il/jtl/jtl-over-cs-b.pdf

[8] T. Cohen, J. Y. Gil, and I. Maman, "JTL—The Java Tools Language," in *21st ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[9] D. S. Dantas and D. Walker, "Harmless advice," in *33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 2006.

[10] E. Hajiyev, M. Verbaere, and O. de Moor, "Codequest: Source code querying with datalog." [Online]. Available: http://progtools.comlab.ox.ac.uk/projects/codequest

[11] E. Hajiyev, N. Ongkingco, P. Avgustinov, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere, "Datalog as a pointcut language in aspect-oriented

programming," in *21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, 2006.

[12] R. I. in Software Evolution, "Software evolution." [Online]. Available: http://www.program-transformation.org/Transform/SoftwareEvolution

[13] B. Jørgensen, E. Pedersen, and T. Norstved, "Tool Support for AspectDNG - Creating an Advice Wizard for simpler AOP development," Aalborg University, Tech. Rep., 2006.

[14] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the evolution of aspect-oriented software with model-based pointcuts," in *20th European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[15] ——, "A model-driven pointcut language for more robust pointcuts," in *4th Int. AOSD Workshop on Software Engineering Properties of Languages for Aspect Technology (SPLAT)*, 2006.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting started with AspectJ," *Comm. ACM*, vol. 44, no. 10, pp. 59–65, Oct. 2001.

[17] G. Kniesel and T. Rho, "Generic aspect languages - needs, options and challenges," in *2nd French Workshop on Aspect-Oriented Software Development (JFDLPA)*. [Online]. Available: http://www.lifl.fr/jfdlpa05/kniesel.pdf

[18] C. Koppen and M. Stoerzer, "PCDiff: Attacking the fragile pointcut problem," in *1st European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[19] T. O'Reilly, "Programming language trends," Website. [Online]. Available: http://radar.oreilly.com/archives/2006/08/programming_language_trends_1.html

[20] K. Ostermann, M. Mezini, and C. Bockisch, "Alpha project." [Online]. Available: http://www.st.informatik.tu-darmstadt.de/static/pages/projects/alpha/index.html

[21] ——, "Expressive pointcuts for increased modularity," in *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[22] Palo Alto Research Center, "Aspectj." [Online]. Available: http://www.parc.com/research/projects/aspectj/default.html

[23] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005.

[24] T. Rho, G. Kniesel, M. Appeltauer, and A. Linder, "Logicaj (logic aspects for java)." [Online]. Available: http://roots.iai.uni-bonn.de/research/logicaj/logicaj2/

[25] T. Rho, G. Kniesel, and M. Appeltauer, "Fine-grained generic aspects," in *Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2006.

[26] M. Rinard, A. Salcianu, and S. Bugrara, "A classification system and analysis for aspect-oriented programs," in *12th ACM SIGSOFT symposium on Foundations of software engineering*, 2004.

[27] F. Steimann, "The paradoxical success of aspect-oriented programming," in *21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, 2006.

[28] T. Tourwé, J. Brichau, and K. Gybels, "On the existence of the aosd-evolution paradox," in *Workshop on Software-engineering Properties of Languages (AOSD 2003)*, 2003.

APPENDIX A

Acronyms

CLARA Cross-language Reusable Aspect-language

AOP Aspect-Oriented Programming

AOSD Aspect-Oriented Software Development

AST Abstract Syntax Tree

IDE Integrated Development Environment

JTL Java Tools Language

OOP Object-Oriented Programming

LMP Logic Meta Programming

LMV Logic Meta Variable

BCEL Bytecode Engineering Library

APPENDIX B

Demonstration: Bank Application Output

```
BANK APPLICATION 1.0
====================
Select your action:
1: Create new account
2: Open account
0: Quit

Enter your choice: 2

Enter the account number you wish to open: 1
ACCOUNT
=======
Account number: 1
Balance: 5000.0

What do you want to do?
1: Withdraw
2: Deposit
3: Transfer to another account.
0: Go back to main screen

Choice: 1

Enter amount you want to withdraw: 2000
========================================
Your action requires supervisor approval.
========================================
Username: boss
Password: 1234
Withdrawing: 2000.0. New balance: 3000.0
ACCOUNT
=======
Account number: 1
Balance: 3000.0

What do you want to do?
1: Withdraw
2: Deposit
3: Transfer to another account.
0: Go back to main screen

Choice: 1

Enter amount you want to withdraw: 1500
Withdrawing: 1500.0. New balance: 1500.0
```

Listing B.1: Output from the Java Bank application test.

APPENDIX C

Demonstration: Bank Native Interface

```java
package BankApplication;

import aspect.Authentication_NativeInterface;
import java.util.Scanner;

public class NativeBridge implements Authentication_NativeInterface {

    public boolean authenticate() {
        System.out.println("========================================");
        System.out.println("Your action requires supervisor approval.");
        System.out.println("========================================");

        Scanner in = new Scanner(System.in);
        System.out.print("Username: ");
        String username = in.nextLine();

        System.out.print("Password: ");
        String password = in.nextLine();

        for (User u : Bank.getInstance().getUsers()) {
            if (u instanceof Supervisor) {
                if (u.getUsername().equals(username) && u.getPassword().equals(
                    password)) {
                    return true;
                }
            }
        }

        return false;
    }

    public void deny() {
        System.err.println("This transaction requires supervisor approval,
            transaction denied");
    }
}
```

Listing C.1: Java implementation of the native interface Authentication_NativeInterface

APPENDIX D

Grammar

⟨program⟩ =
    ⟨aspect-declaration⟩

⟨aspect-declaration⟩ =
    'abstract' 'aspect' ⟨identifier⟩ '{' ⟨abstract-aspect-body⟩ '}' |
    'aspect' ⟨identifier⟩ '{' ⟨aspect-body⟩ '}' |
    'aspect' ⟨identifier⟩ 'extends' ⟨identifier⟩ '{' ⟨aspect-body⟩ '}'

⟨abstract-aspect-body⟩ =
    ⟨native-interface⟩ ⟨abstract-aspect-member-list⟩ |
    ⟨abstract-aspect-member-list⟩

⟨aspect-body⟩ =
    ⟨native-interface⟩ ⟨aspect-member-list⟩ |
    ⟨aspect-member-list⟩

⟨native-interface⟩ =
    'NativeInterface' '{' ⟨native-interface-member-list⟩ '}'

⟨abstract-aspect-member-list⟩ =
    ⟨abstract-aspect-member⟩ |
    ⟨abstract-aspect-member-list⟩ ⟨abstract-aspect-member⟩

⟨aspect-member-list⟩ =
    ⟨aspect-member⟩ |
    ⟨aspect-member-list⟩ ⟨aspect-member⟩

⟨native-interface-member-list⟩ =
    ⟨native-interface-member⟩ |
    ⟨native-interface-member-list⟩ ⟨native-interface-member⟩

⟨abstract-aspect-member⟩ =
    ⟨pointcut-declaration⟩ |
    ⟨abstract-pointcut-declaration⟩ |
    ⟨advice-declaration⟩ |
    ⟨policy-declaration⟩ |
    ⟨abstract-method-declaration⟩ |
    ⟨method-declaration⟩ |
    ⟨field-declaration⟩

⟨aspect-member⟩ =
    ⟨pointcut-declaration⟩ |
    ⟨advice-declaration⟩ |
    ⟨policy-declaration⟩ |
    ⟨field-declaration⟩ |
    ⟨method-declaration⟩

⟨native-interface-member⟩ =
    ⟨identifier⟩ ⟨identifier⟩ ';' |
    ⟨identifier⟩ ⟨identifier⟩ '(' ⟨method-argument-list⟩ ')' ';'

⟨method-declaration⟩ =
    ⟨identifier⟩ ⟨identifier⟩ '(' ⟨method-argument-list⟩ ')' '{' ⟨method-body⟩ '}'

⟨field-declaration⟩ =
    ⟨identifier⟩ ⟨identifier⟩ '=' expression ';' |
    ⟨identifier⟩ ⟨identifier⟩ ';'

⟨policy-declaration⟩ =
    'policy' ⟨identifier⟩ 'forbids' ⟨identifier⟩ ':' '"' ⟨policy-message⟩ '"' ';' |
    'policy' ⟨identifier⟩ ⟨identifier⟩ 'requires' ⟨identifier⟩ ':' '"' ⟨policy-message⟩ '"' ';'

⟨advice-declaration⟩ =
    'advice' ⟨identifier⟩ ':' 'around' ⟨identifier⟩ '{' ⟨advice-body⟩ '}' |
    'advice' ⟨identifier⟩ ':' 'narrows' ⟨identifier⟩ '{' ⟨advice-body-narrows⟩ '}' |
    'advice' ⟨identifier⟩ ':' 'replaces' ⟨identifier⟩ '{' ⟨advice-body-replaces⟩ '}'

⟨pointcut-declaration⟩ =
    'pointcut' ⟨identifier⟩ '[' ⟨pointcut-parameter⟩ ']' ':=' '"' ⟨pointcut-expression⟩ '"' ';'

⟨abstract-pointcut-declaration⟩ =
    'abstract' 'pointcut' ⟨identifier⟩ '[' ⟨pointcut-parameter⟩ ']' ';'

⟨abstract-method-declaration⟩ =
    'abstract' ⟨identifier⟩ ⟨identifier⟩ '(' ⟨method-argument-list⟩ ')' ';'

⟨native-interface-method-declaration⟩ =
    ⟨identifier⟩ ⟨identifier⟩ '(' ⟨method-argument-list⟩ ')' ';'

⟨native-interface-field-declaration⟩ =
    ⟨identifier⟩ ⟨identifier⟩ ';'

⟨method-argument-list⟩ =
    ⟨method-argument⟩ |
    ⟨method-argument-list⟩ ',' ⟨method-argument⟩

⟨method-body⟩ =
    ⟨method-body-statement-list⟩

⟨method-body-statement-list⟩ =
⟨method-body-statement⟩ |
⟨method-body-statement-list⟩ ⟨method-body-statement⟩

⟨policy-message⟩ =
⟨string-literal⟩

⟨advice-body⟩ =
⟨advice-boyd-statement-list⟩

⟨advice-body-replaces⟩ =
⟨advice-body-replaces-statement-list⟩

⟨advice-body-narrows⟩ =
'condition' '{' ⟨advice-expression⟩ '}' 'success' '{' ⟨advice-body⟩ '}' 'failure' '{' ⟨advice-body-replaces⟩ '}'

⟨pointcut-expression⟩ =
⟨string-literal⟩

⟨pointcut-parameter⟩ =
⟨identifier⟩

⟨method-body-statement⟩ =
⟨variable-declaration-statement⟩ |
⟨variable-declaration-assignment-statement⟩ |
⟨assignment-statement⟩ |
⟨invoke-statement⟩ |
⟨native-interface-invoke-statement⟩ |
⟨compound-if-statement⟩ |
⟨while-statement⟩ |
⟨method-foreach-statement⟩ |
⟨method-return-statement⟩

⟨invoke-statement⟩ =
⟨identifier⟩ '(' ⟨invoke-argument-list⟩ ')' ';'

⟨advice-body-replaces-statement⟩ =
⟨advice-variable-declaration-statement⟩ |
⟨advice-variable-declaration-assignment-statement⟩ |
⟨advice-assignment-statement⟩ |
⟨advice-invoke-statement⟩ |
⟨advice-native-interface-invoke-statement⟩ |
⟨advice-replaces-compound-if-statement⟩ |
⟨advice-replaces-while-statement⟩ |
⟨advice-replaces-foreach-statement⟩ |
⟨advice-return-statement⟩

⟨advice-body-statement⟩ =

⟨advice-variable-declaration-statement⟩ |
⟨advice-variable-declaration-assignment-statement⟩ |
⟨advice-assignment-statement⟩ |
⟨advice-invoke-statement⟩ |
⟨advice-native-interface-invoke-statement⟩ |
⟨advice-compound-if-statement⟩ |
⟨advice-while-statement⟩ |
⟨advice-foreach-statement⟩ |
⟨advice-return-statement⟩ |
⟨advice-proceed-statement⟩

⟨expression⟩ =
⟨primary-expression⟩ |
⟨expression⟩ ⟨operator⟩ ⟨primary-expression⟩

⟨primary-expression⟩ =
⟨integer-literal⟩ |
⟨float-literal⟩ |
⟨boolean-literal⟩ |
⟨string-literal⟩ |
⟨identifier⟩ |
⟨invoke-expression⟩ |
⟨native-interface-invoke-expression⟩ |
'+' ⟨primary-expression⟩ |
'-' ⟨primary-expression⟩ |
'(' ⟨expression⟩ ')'

⟨operator⟩ =
'+' |
'-' |
'*' |
'/' |
'<' |
'>' |
'==' |
'!=' |
'<=' |
'>=' |
'&&' |
'||'

⟨invoke-argument-list⟩ =
⟨expression⟩ |
⟨invoke-argument-list⟩ ',' ⟨expression⟩

⟨variable-declaration-assignment-statement⟩ =
⟨identifier⟩ ⟨identifier⟩ '=' ⟨expression⟩ ';'

⟨variable-declaration-statement⟩ =

⟨identifier⟩ ⟨identifier⟩ ';'

⟨assignment-statement⟩ =
⟨identifier⟩ '=' ⟨expression⟩ ';'

⟨compound-if-statement⟩ =
⟨if-statement⟩ |
⟨if-statement⟩ ⟨else-if-statement-list⟩ |
⟨if-statement⟩ ⟨else-if-statement-list⟩ ⟨else-statement⟩ |
⟨if-statement⟩ ⟨else-statement⟩

⟨if-statement⟩ =
'if' '(' ⟨conditional-expression⟩ ')' '{' ⟨method-body⟩ '}'

⟨else-if-statement⟩ =
'else' 'if' '(' ⟨conditional-expression⟩ ')' '{' ⟨method-body⟩ '}'

⟨else-statement⟩ =
'else' '{' ⟨method-body⟩ '}'

⟨while-statement⟩ =
'while' '(' ⟨conditional-expression⟩ ')' '{' ⟨method-body⟩ '}'

⟨conditional-expression⟩ =
⟨expression⟩

⟨advice-variable-declaration-assignment-statement⟩ =
⟨identifier⟩ ⟨identifier⟩ '=' ⟨advice-expression⟩ ';'

⟨advice-assignment-statement⟩ =
⟨identifier⟩ '=' ⟨advice-expression⟩ ';'

⟨advice-invoke-statement⟩ =
⟨identifier⟩ '(' ⟨advice-invoke-argument-list⟩ ')' ';'

⟨advice-replaces-compound-if-statement⟩ =
⟨advice-replaces-if-statement⟩ |
⟨advice-replaces-if-statement⟩ ⟨advice-replaces-else-if-statement-list⟩ |
⟨advice-replaces-if-statement⟩ ⟨advice-replaces-else-if-statement-list⟩ ⟨advice-replaces-else-statement⟩ |
⟨advice-replaces-if-statement⟩ ⟨advice-replaces-else-statement⟩

⟨advice-replaces-if-statement⟩ =
'if' '(' ⟨advice-conditional-expression⟩ ')' '{' ⟨advice-body-replaces⟩ '}'

⟨advice-replaces-else-if-statement⟩ =
'else' 'if' '(' ⟨advice-conditional-expression⟩ ')' '{' ⟨advice-body-replaces⟩ '}'

⟨advice-replaces-else-statement⟩ =

'else' '{' ⟨advice-body-replaces⟩ '}'

⟨advice-replaces-while-statement⟩ =
'while' '(' ⟨advice-conditional-expression⟩ ')' '{' ⟨advice-body-replaces⟩ '}'

⟨advice-conditional-expression⟩ =
⟨advice-expression⟩

⟨advice-expression⟩ =
⟨advice-primary-expression⟩ |
⟨advice-expression⟩ ⟨operator⟩ ⟨advice-primary-expression⟩

⟨advice-primary-expression⟩ =
⟨integer-literal⟩ |
⟨float-literal⟩ |
⟨boolean-literal⟩ |
⟨string-literal⟩ |
⟨identifier⟩ |
⟨advice-invoke-expression⟩ |
⟨advice-native-interface-invoke-expression⟩ |
'+' ⟨primary-expression⟩ |
'-' ⟨primary-expression⟩ |
'(' ⟨expression⟩ ')' |
⟨meta-variable-expression⟩ |
⟨meta-variable-indexer-expression⟩ |
⟨proceed-result⟩

⟨advice-while-statement⟩ =
'while' '(' ⟨advice-conditional-expression⟩ ')' '{' ⟨advice-body⟩ '}'

⟨advice-foreach-statement⟩ =
'foreach' '(' ⟨advice-foreach-expression⟩ ')' '{' ⟨advice-body⟩ '}' |
'foreach' '(' ⟨method-foreach-expression⟩ ')' '{' ⟨advice-body⟩ '}'

⟨advice-replaces-foreach-statement⟩ =
'foreach' '(' ⟨advice-foreach-expression⟩ ')' '{' ⟨advice-body-replaces⟩ '}' |
'foreach' '(' ⟨method-foreach-expression⟩ ')' '{' ⟨advice-body-replaces⟩ '}'

⟨method-foreach-statement⟩ =
'foreach' '(' ⟨method-foreach-expression⟩ ')' '{' ⟨advice-body-replaces⟩ '}'

⟨advice-foreach-expression⟩ =
⟨meta-variable-expression⟩ 'as' ⟨identifier⟩

⟨method-foreach-expression⟩ =
⟨identifier⟩ 'as' ⟨identifier⟩

⟨advice-compound-if-statement⟩ =
⟨advice-if-statement⟩ |

⟨advice-if-statement⟩ ⟨advice-else-if-statement-list⟩ |
⟨advice-if-statement⟩ ⟨advice-else-if-statement-list⟩ ⟨advice-else-statement⟩ |
⟨advice-if-statement⟩ ⟨advice-else-statement⟩

⟨advice-if-statement⟩ =
'if' '(' ⟨advice-conditional-expression⟩ ')' '{' ⟨advice-body⟩ '}'

⟨advice-else-if-statement⟩ =
'else' 'if' '(' ⟨advice-conditional-expression⟩ ')' '{' ⟨advice-body⟩ '}'

⟨advice-else-statement⟩ =
'else' '{' ⟨advice-body⟩ '}'

⟨advice-variable-declaration-statement⟩ =
⟨identifier⟩ ⟨identifier⟩ ';'

⟨advice-native-interface-invoke-expression⟩ =
'NativeInterface' '.' ⟨identifier⟩ '(' ⟨advice-invoke-argument-list⟩ ')'

⟨advice-native-interface-invoke-statement⟩ =
'NativeInterface' '.' ⟨identifier⟩ '(' ⟨advice-invoke-argument-list⟩ ')' ';'

⟨unary-operator⟩ =
'+' |
'-'

⟨meta-variable-expression⟩ =
⟨meta-variable⟩ '.' ⟨identifier⟩

⟨advice-invoke-argument-list⟩ =
⟨advice-expression⟩ |
⟨advice-expression⟩ ',' ⟨advice-invoke-argument-list⟩

⟨native-interface-invoke-expression⟩ =
'NativeInterface' '.' ⟨identifier⟩ '(' ⟨invoke-argument-list⟩ ')'

⟨native-interface-invoke-statement⟩ =
'NativeInterface' '.' ⟨identifier⟩ '(' ⟨invoke-argument-list⟩ ')' ';'

⟨invoke-expression⟩ =
⟨identifier⟩ '(' ⟨invoke-argument-list⟩ ')'

⟨advice-invoke-expression⟩ =
⟨identifier⟩ '(' ⟨advice-invoke-argument-list⟩ ')'