Universiteit Antwerpen

Towards A Framework For Creating Aspect-Oriented Languages

Chris Vesters

Principal Adviser: Dirk Janssens
Assistant Adviser: Tim Molderez

Dissertation Submitted in June 2014 to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Tim Molderez for guiding me during the entire process, providing me with ideas and asking the right questions. I would like to thank professor Janssens for making this thesis possible. A lot of thanks go out to my family and friends for supporting me during the entire period, and pulling me through all the work. Lastly I would like to thank the creators of ANTLR and ObjectAid for creating this software which made my work a lot easier.

Nederlandstalige Samenvatting

Aspectgeoriënteerde programmeertalen worden momenteel vaak opgebouwd vertrekkende vanaf een bestaande taal waaraan dergelijke elementen worden toegevoegd. Deze extensie is vrij verregaand en vergt behoorlijk wat werk doordat ze altijd vanaf nul wordt begonnen.

In deze thesis stellen we een framework voor dat de kernelementen van aspectgeoriënteerd programmeren bevat volledig los van elke basistaal. De bedoeling van dit framework is om het mogelijk te maken om nieuwe en reeds bestaande talen te voorzien met aspectgeoriënteerd programmeren, sneller en makkelijker dan momenteel het geval is. De grootste uitdaging hierbij is het vinden van een balans tussen de abstractie en het vereiste werk. Een framework dat niet abstract genoeg is, zal slechts in enkele gevallen bruikbaar zijn, terwijl een te abstract framework geen bijdrage levert tot het verminderen van het benodigde werk.

De capaciteiten en gebreken worden aangetoond door middel van Small C en Dot, twee uiteenlopende talen, te voorzien van aspect-georienteerd programmeren. We gaan ook voor verschillende domein-specifieke aspect talen na of we deze kunnen implementeren met behulp van het framework.

De resultaten zijn afhankelijk van de beschikbare tools en de gewenste features, maar algemeen kan gesteld worden dat het framework in staat is om verschillende soorten talen te ondersteunen. Bovendien zijn ook de features aanwezig in het framework makkelijk te gebruiken wat de benodigde tijd zou moeten reduceren.

Abstract

Aspect-oriented programming languages are often based on an existing base language to which aspect-oriented elements are added. This extension is pretty invasive and takes a lot of work since it is done from scratch.

A framework is presented that contains all core elements of aspect-oriented programming, completely independent of any base language. The goal is to make it possible to extend the framework and provide new and existing languages with aspect-oriented programming faster and easier than is currently possible. The biggest challenge is the trade-off between the level of abstraction and the work required to extend it. A framework that does not sufficiently abstract from the base language can only be used for certain languages, while too much abstraction wouldn't add any value as the amount of work to extend it would be the same as starting from scratch.

We show the functionality of the framework by using it on Small C and Dot, two different type of languages. We also examine several domain-specific aspect languages and discuss whether they can be implemented using the framework.

CHAPTER 1

Introduction

1.1 Aspect-Oriented Programming

Aspect-oriented programming [4] (from now on referred to as AOP) is a way of programming that allows separating code beyond the capabilities of object-oriented programming. It does this by enabling the separation of crosscutting concerns from core concerns. By doing this which we achieve more modular code, prevent code tangling and code scattering which results in code that is easier to maintain and modify. In figure 1.1 we see that the calls to the security module are scattered across multiple modules, where they are tangled with the rest of the code.

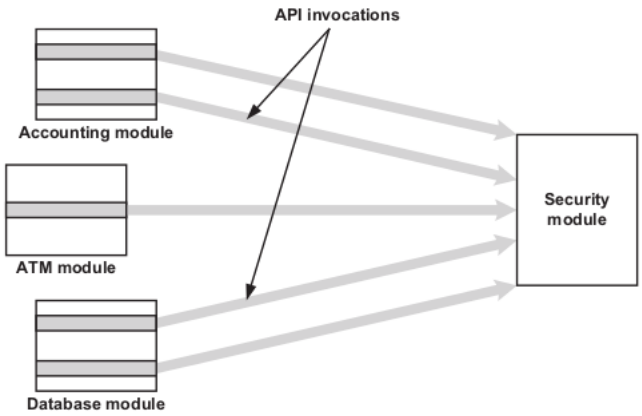

Figure 1.1: A representation of code scattering and code tangling.

AOP works by identifying join points, which are certain points during the execution of the program, e.g. a method call or execution. The set of all the join points is called the join point model. It is at these join points that we can modify the original program execution, which can go from something as simple as logging to something more invasive where the original code isn't even executed at all.

To specify which join points we would like to interact with, a pointcut is used. Besides referring to a certain join point we also need a way to modify the execution of it. This is done by an advice, note that there can be multiple kinds of advice, e.g. before, after and around in AspectJ. In the end all these elements are put together by the weaver which results in woven code. The weaver will make sure that the advice is called at the join points identified by the pointcut it is connected to.

The most well-known AOP language is AspectJ, which is an extension of Java. To demonstrate the concept of AOP a couple of AspectJ examples are shown here. In listing 1.1 a pointcut is specified that will match any deliver method of the MessageCommunicator, no matter the arguments or return type. Before this method is executed the user is authenticated.

```
1  package ajia.security;
2  import ajia.messaging.MessageCommunicator;
3
4  public aspect SecurityAspect {
5    private Authenticator authenticator = new Authenticator();
6
7    pointcut secureAccess(): execution(* MessageCommunicator.deliver(..));
8
9    before() : secureAccess() {
10     System.out.println("Checking and authenticating user");
11     authenticator.authenticate();
12   }
13 }
```

Listing 1.1: An aspect to authenticate the user.

Another example, shown in listing 1.2, determines the time required to execute any public method.

```
1  package ajia.profile;
2
3  public aspect ProfilingAspect {
4    pointcut publicOperation() : execution(public * *.*(..));
5
6    Object around() : publicOperation() {
7      long start = System.nanoTime();
8      Object ret = proceed();
9      long end = System.nanoTime();
10     System.out.println(thisJoinPointStaticPart.getSignature() + " took " + (
       end-start) + " nanoseconds");
11     return ret;
12   }
13 }
```

Listing 1.2: An aspect to profile methods.

1.2 Problem

AOP is often implemented by extending an existing language, which often leads to a completely new language requiring new compilers and breaking existing tools for the base language. Especially the latter is a reason why some people haven't made the step to start working with aspect-oriented languages.

Though the concepts of AOP are quite general, which is reflected by the similarities in different aspect-oriented languages, all the AOP parts of these languages were written from scratch. The main reason for this is the absence of a general, language-independent library or framework to handle aspect-orientation. Another reason is that by building a language from scratch the developer gets more freedom to implement the desired features.

1.3 Goal

The goal of this thesis is to build such a framework that encloses all core ideas of AOP in an abstract manner without being specific about a particular base language. This should enable us to use the framework to develop an aspect-oriented language quicker and easier than currently is the case, by extending a couple of parts. The framework should be able to work for completely new base languages, and existing ones we want to extend. The latter one will pose the most problems as we wish to modify the existing compiler as little as possible. By making the framework abstract enough we want to ensure that it can be used for different types of base languages.

1.4 Overview

The remainder of this thesis is structured as follows: in chapter 2 the components of an aspect-oriented language are explained. In chapter 3 the developed framework is presented. In chapters 4 and 5 the framework is instantiated for Small C and Dot. In chapters 6, 7 and 8 we examine how some domain-specific aspect languages can be implemented using the framework. In chapter 9 the current version of the framework is discussed, and the future work is presented in chapter 10. Chapter 11 discusses some related work and chapter 12 concludes the thesis.

CHAPTER 2

Aspect-Oriented Language Components

An aspect-oriented language consists of several components. To explain these I will start from the base language and work my way up to get an aspect-oriented language. At every point I will make it more concrete with a small AspectJ example.

The starting point of creating an aspect-oriented language is the base language, this is the existing language that we want to extend or is a completely new language. In the latter case the base language will be the new language excluding all the aspect-oriented features. The base language can be any kind of language: procedural, object-oriented, functional, logical, data-oriented, etc.

After defining the base language we can define the join point model. This consists of all the points that occur during execution on which intervention of the aspects is allowed. The join point model itself is completely implicit, but it still is an important part of any aspect-oriented language as it is the link between the base language and the aspect-oriented part.

Knowing at which points we can intervene doesn't allow us to do anything, we need a way to identify certain join points. This identification is done by using pointcuts, specifying one or more join points. Pointcuts make the join point model explicitly present in the aspect-oriented language. The way in which pointcuts allow us to refer to join points should be as close as possible to the way in which they appear in the base language. Listing 2.1 shows two pointcuts. Both match the method 'bar' of an object 'Foo', though methodCall will match the call of it, and methodExec will match the execution.

```
1 pointcut methodCall(): call(public void Foo.bar(int));
2 pointcut methodExec(): execution(public void Foo.bar(int));
```

Listing 2.1: Two method pointcuts.

We use the example shown in listing 2.2 to show the difference between the two pointcuts shown earlier. The pointcut 'methodCall' matches the actual call from 'callbar' while 'methodExec' matches the execution of the method itself.

```
1  public class Foo {
2    void bar(int i) {
3      ...
4    }
5  }
6
7  public class CallFoo {
8    void callbar() {
9      bar(5);
10   }
11 }
```

Listing 2.2: Example Java code.

Now that we can refer to certain join points, we can actually start modifying the original code. This modification is encapsulated in an advice, which contains code that has to be executed. When this code should be executed is defined by the pointcut with which the advice is associated. An example of how this connection is established in AspectJ is shown in listing 2.3.

```
1 pointcut methodCall(): call(public void Foo.bar(int));
2
3 before(): methodCall() {
4   System.out.println("Calling bar");
5 }
```

Listing 2.3: An example of an advice.

Whenever the program passes a join point that matches with the associated pointcut, this advice must be executed. The code in the advice should be written in the base language to prevent the user from learning an entire new language. The examples in listing 1.1 and 1.2 clearly show that most part of the advice body consists of plain Java code. The advice should also be able to have access to information about the join point. In AspectJ there are two ways to pass information. A first one is by using arguments as shown in listing 2.4, another is by using the special **thisJoinPoint** keyword as shown in listing 2.5.

```
1 pointcut methodCall(int i) : call(public  void Foo.bar(int) && args(i);
2
3 before(int o) : methodCall(o) {
4   System.out.println("Calling bar with " + o);
5 }
```

Listing 2.4: Example of argument passing between pointcut and join point.

```
1 before() : methodCall() {
2   System.out.println("Calling " + thisJoinPoint);
3 }
```

Listing 2.5: Example of using the thisJoinPoint keyword.

One problem that arises is: what is suppose to happen if we encounter a join point at which multiple advice are to be executed? This can be either caused by multiple pointcuts matching to the join point, or multiple advice being associated with the same pointcut. The latter case can be easily solved as we can combine the two advice into one bigger advice, as shown in listing 2.6 and 2.7.

```
1 before(): fooCall() {
2   System.out.println("Advice 1.");
3 }
4
5 before(): fooCall() {
6   System.out.println("Advice 2.");
7 }
```

Listing 2.6: Two colliding advice.

```
1 before(): fooCall() {
2   System.out.println("Advice 1.");
3   System.out.println("Advice 2.");
4 }
5
6 before(): fooCall() {
7   System.out.println("Advice 2.");
8   System.out.println("Advice 1.");
9 }
```

Listing 2.7: Two possible merged versions of the advice.

The first is a bit more tricky since one pointcut can match join points different from the other, which means we can't just combine them. Imagine the two pointcuts shown in listing 2.8, it is clear that they would both match a method call to 'foobar', but while the first also matches a call to the method 'foo' the second will not. The other way around the second will match a call to the method 'bar', but the first will not.

```
1 pointcut fooCall(): call(public void *.foo*());
2 pointcut barCall(): call(public void *.*bar());
```

Listing 2.8: Overlapping pointcuts.

The solution to this is of course to introduce an ordering mechanism. This ordering can be either placed on the pointcuts or on the advice, in the first case we still need to make sure that there is only one advice for each pointcut though. In AspectJ this is done by specifying the advice in the file in the desired order. An example of this is shown in listing 2.9 and its output in listing 2.10.

```
1 before(): fooCall() {
2   System.out.println("Advice 1.");
3 }
4
5 around(): fooCall() {
6   System.out.println("Begin.");
7   proceed();
8   System.out.println("End.");
9 }
10
11 before(): fooCall() {
12   System.out.println("Advice 2.");
13 }
```

Listing 2.9: Example of declaring precedence.

```
1 Advice 1
2 Begin
3 Advice 2
4 End
```

Listing 2.10: Output of the ordered advice.

One final component we need, is something that will actually make sure that the advice are executed when the join points are encountered. The weaver will make this happen, and can do this in multiple ways. The easiest is compile-time weaving, which comes down to a pre-processing step. A more advanced form is run-time-weaving, yet other forms are available depending on the base language.

With the components specified above it is perfectly possible to create an aspect-oriented language. However, some features are still missing, e.g. communication between different advice is impossible. Since aspect-orientation is often presented as an extension of object-oriented programming it makes sense to introduce some form of module or container to encapsulate pointcuts and advice. By doing this we immediately have the advantages offered by object-oriented languages among which are reusability, inheritance and encapsulation. Some examples of how reuse of an aspect is done in AspectJ is shown in listing 2.11.

```
1  public aspect ProgramPoints {
2    public pointcut main(): execution(public * *.*(..));
3    public pointcut expensive(): execution(* *.Calculator || *.Renderer).*(..));
4  }
5
6  public abstract aspect Profiling {
7    int count = 0;
8    abstract pointcut toProfile();
9    after(): toProfile() {
10     count++;
11   }
12 }
13
14 public aspect MyProfiling extends Profiling {
15   pointcut toProfile():
16   ProgramPoints.expensive();
17 }
```

Listing 2.11: Possibilities to reuse aspects.

An overview of all the components and how they are linked together is shown in figure 2.1. The weaver is not considered to be part of the aspect language, just like a compiler is not considered a part of the language.

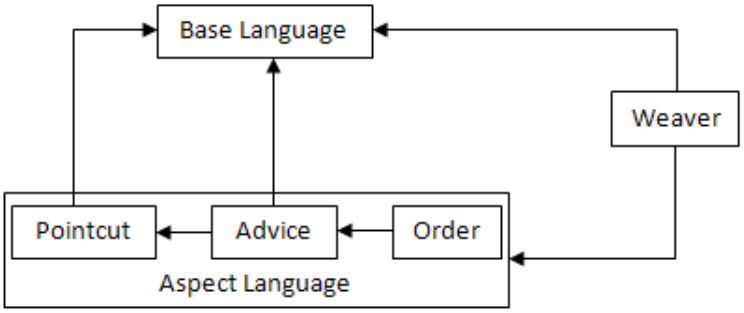

Figure 2.1: An overview of the components.

CHAPTER 3

Aspect-Oriented Framework

Since the framework has to be easily extensible to work with any kind of language it has to be very abstract and may not contain any information about a base language. On the other hand, creating a framework that is too abstract may require more effort from the developer in order to instantiate the framework with a particular base language, making the framework completely miss its goal. This means that the level of abstraction has to be well considered.

First I will go into detail how the components that were identified in the previous chapter are represented and work in the framework. Then I will provide information about how the weaver works. Once all the components have been fully explained some remarks about the current version of the framework are made. To conclude the chapter I explain how we can now expand this framework to work with any base language.

The first component of the aspect language that we identified are pointcuts. To completely specify a join point we need some way to identify its signature, this however is language specific and can therefore not be provided by the framework. Other common structures are arguments and context. The arguments are meant for information that can be used in the advice, while the context is a way to restrict matching join points based on information that goes beyond the scope of the join point. The pointcut is implemented as an abstract Pointcut class.

Letting an advice link to only one pointcut is too restrictive in some cases. It is easy to imagine we want an advice to be executed upon multiple join points. If these join points differ too much in signature it becomes impossible to describe them by one pointcut without also matching unwanted join points. This will result in duplicating the advice, to link it with more than one pointcut. It is for this reason that the framework provides a way to group those pointcuts in a set. Instead of linking advice to a single pointcut, they are now linked with an entire set. A set will match a join point if at least one of the pointcuts it contains matches the join point. Arguments can be made available for the advice by defining it in the set. All the pointcuts in that set will have to define those arguments on their own as well. A UML diagram of the PointcutSet and the Pointcut class can be seen in figure 3.1.

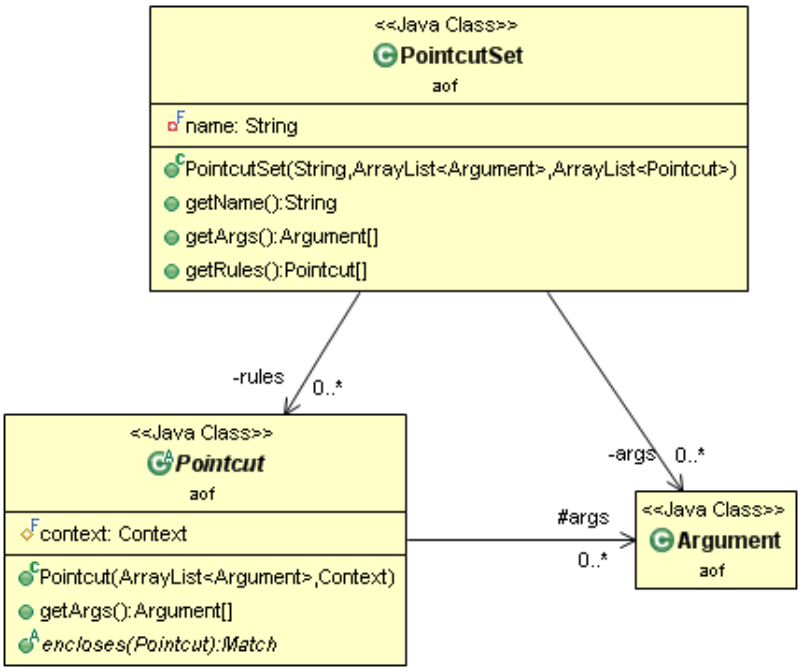

Figure 3.1: UML diagram of the PointcutSet and Pointcut class.

The advice are implemented by the abstract Advice class. This class has a name, which will make ordering them a lot easier, and a set of arguments. These arguments are the ones that can be used inside the advice body. Note that the arguments of the pointcutset to which this advice is linked to must be assignable to these arguments. The body or actual code of the advice is not provided by this class, simply because it depends on the base language as we want this to be in the same formalism. The UML diagram of the Advice class is shown in figure 3.2.

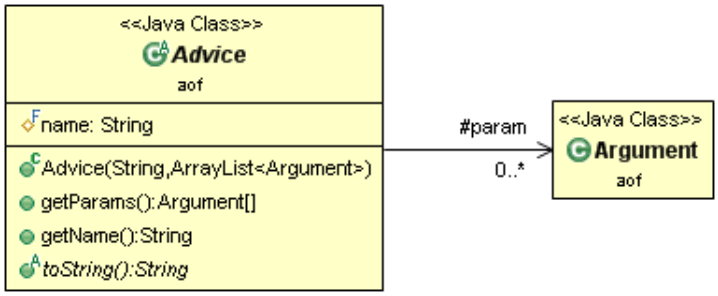

Figure 3.2: UML diagram of the Advice.

I already mentioned arguments and context but didn't explain how these are implemented. An argument is hard to abstract as every base language has its own vision of what an argument is. Though two concepts are very common, being a name and a type. For this reason the Argument class has a name, which is also required for access to the argument, and a

type. While the name can be a string as this is general enough, the type is harder since it can not be a base language type, but should be more general. For this reason an interface called Type was created which allows the user to use his own type system. A UML diagram of this Argument class and the Type interface are shown in figure 3.3.

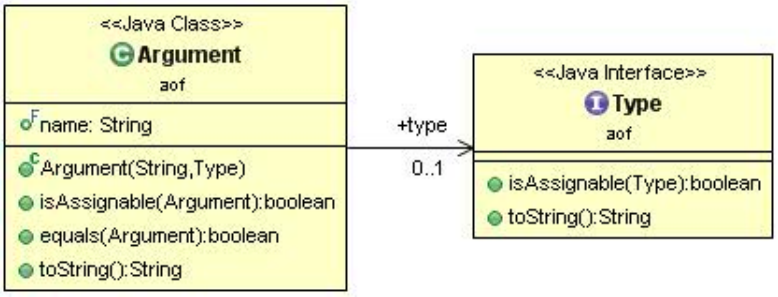

Figure 3.3: UML diagram of the Argument class.

A context is even more language dependent, which makes it impossible for the framework to provide more than the currently provided interface Context. Though there is a clear difference between arguments and context conceptually, this is however not often the case for real elements. Both context and arguments will often just contain a name and a value. In the current version of the framework the Context interface looks more like the Type interface, as can be seen by comparing the UML diagrams shown in figure 3.4 than the Argument class. I discuss this in more detail in chapter 10.

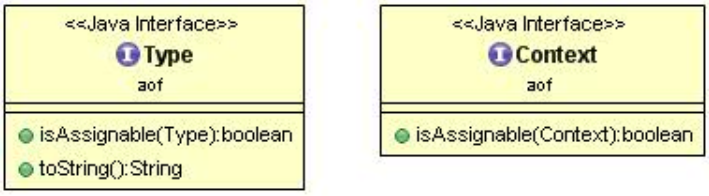

Figure 3.4: UML diagram of the Type and Context interface.

Different types of ordering exist: by assigning values, explicit linking, etc. The framework does not dictate the user which he has to use, instead he is free to choose whichever fits his needs. The user is also free to specify an order on either pointcuts or advice. Despite this freedom on the ordering 'language' the ordering itself is still limited. This is because the order has to be translated to a partial ordering of the advice.

The weaver implemented as an abstract class, provides a way to register all components. The weaver is the part that connects everything together, since the advice does not keep a link to the pointcut itself, nor the other way around. This allows a better separation between pointcuts and advice and gives the weaver more freedom to operate.

The actual weaving can not be done by the weaver as this depends on a lot of things such as preferences of the user and the base language. The framework does however provide some methods that simplify the weaving step. The most important one is 'executingAdvices', given a certain join point this method will return an ordered list of the advice that have to

be executed. This ordering is done by the AdviceComparator, based on the partial ordering achieved from the ordering mechanism. This means that an advice in the list will be placed after all its predecessors. Note that the concept of predecessor has no meaning to the weaver. It is only used to return the ordered list. As an example we consider the partial ordering shown in figure 3.5. For this graph the weaver can return several ordered lists among which:

- A; B; C; D; E; F; G
- A; C; B; D; E; F; G
- A; D; C; B; E; F; G
- A; D; C; E; G; B; F
- D; A; C; E; B; G; F

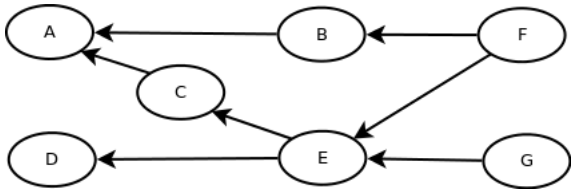

Figure 3.5: A partial ordering of seven advice.

Which order is chosen is unpredictable, and it shouldn't matter as no order was specified among those elements.

In case the user needs to know the pointcuts that match a certain joinpoint a method 'enclosingPointcuts' is provided. Both the 'executingAdvices' and 'enclosingPointcuts' return an instance of the Match class for each match they find. This 'match' is just an argument mapping from those specified in the join point to those that are specified in the advice or pointcut. This Match class, shown in figure 3.6, is used to get the join point specific value for an argument of the advice or pointcut.

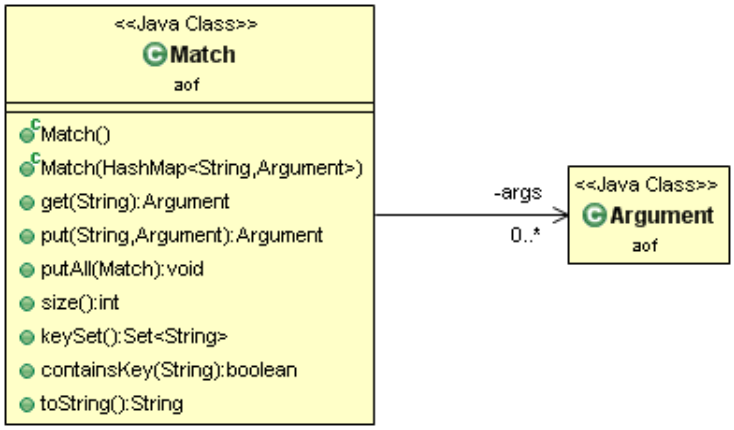

Figure 3.6: UML diagram of the Match class.

The weaver also provides a way to specify input and output files. This is done by using the flags shown in 3.1.

-a *files*	Specify the aspect files.
-s *files*	Specify the source files.
-oDir *dir*	Specify an output directory (optionally).

Table 3.1: An overview of the flags available in the weaver.

An UML diagram of the weaver, showing its complete API, is shown in figure 3.7.

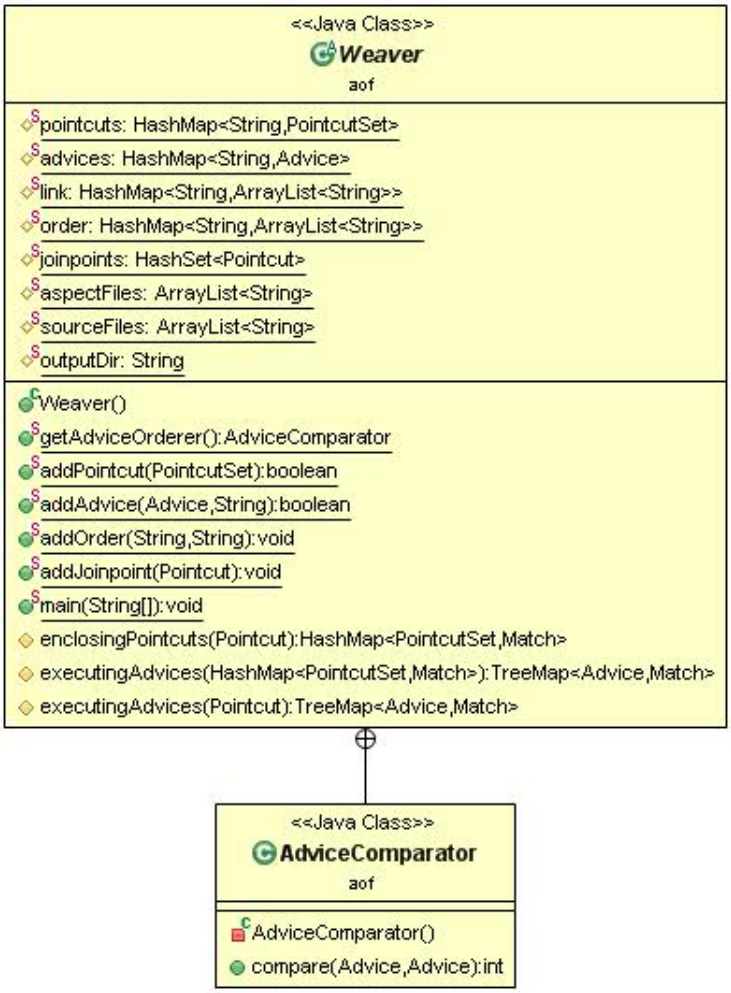

Figure 3.7: UML diagram of the Weaver class.

This concludes the explanation about all the components of the framework. A complete overview of the framework and how everything is connected can be seen in figure 3.8.

Some trade offs had to be made when designing the framework, which I will now discuss in more detail. First of all it needs to be noted that both Context and Type are interfaces which both specify only one method called 'isAssignable'. This means that Context and Type, though called different, do the same thing. We could use one general interface but this is not done to be able to handle future evolution.

Adding a notion of aspect would allow a lot of nice features such as adding members, that can be used for communication between advice, and inheritance. Aspects could also be used as a way to encapsulate pointcuts and advice, though this can also be done without aspects simply by separating them over multiple files. Adding aspects to the framework would have introduced a lot of extra complexity. Since the goal of this thesis is to demonstrate the

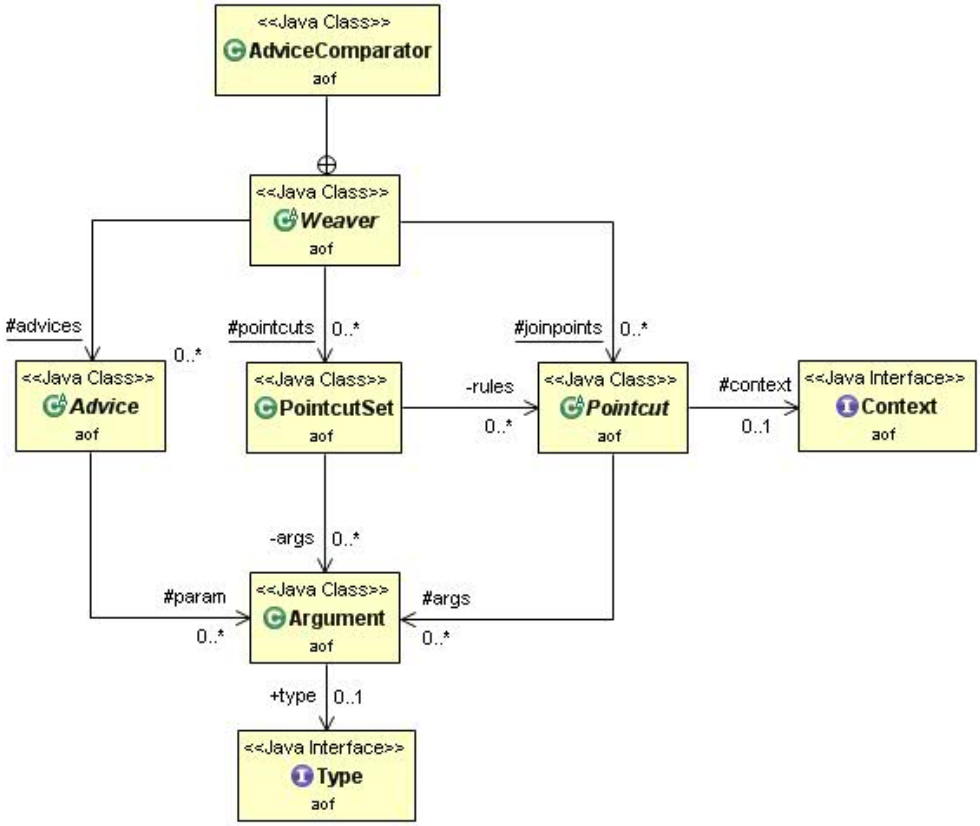

Figure 3.8: UML diagram of the entire framework.

usability of such a framework, I have chosen to not include this but keep it as a future work.

The framework uses an abstract notion of type, this is to achieve the level of abstraction required to separate the framework from the base language. This however means that for each base language we need a new type system that implements the Type interface, which might result in a lot of work. Sometimes it is however possible to use the same type system the compiler of the base language uses, if we can add the interface to it. Doing this would greatly reduce the cost of developing an AOP language.

The advice and pointcuts are not directly connected with each other, the connection is achieved by the weaver. Because adding an explicit link does not provide any advantages, I have chosen to do it this way to ensure that the components are as loosely coupled as possible. Nevertheless the framework demands that at the moment a pointcut is added the pointcutset it is linked with must be be known to the compiler. This is a restriction in the current version of the compiler but can be easily solved by delaying this check until the actual weaving.

Something that was not discussed so far is what happens if a join point is encountered that is matched by multiple pointcuts in the same pointcutset. In this case there basically are two options: either we execute the advice for each matched pointcut, or we only select the first matched pointcut of the set. Currently the latter is implemented in the framework, mostly because if we still want the first to happen, we can simply split the pointcutset.

The base language compiler, or a modification of it has to extract join points from the source code. The join points that are generated will actually be instances of the Pointcut class. By doing this comparing whether a pointcutset contains a pointcut that matches the join point becomes trivial. Implementing both join points and poincuts as an instance of one class can be dangerous since they just aren't the same. One example is that poincuts can contain wildcards while join points can not. It may also be feasible to allow specifying types and sub-types in pointcuts to match multiple join points, where in the join point the types are fixed and known. This means that we have to use the Poincut class carefully if we use it for a join point.

To conclude this chapter I will briefly explain how the framework can be instantiated to work with a base language. For more detailed examples I refer to chapters 4 and 5 where the framework is instantiated for two different languages. Before being able to use the framework, the developer has to specify a language in which it is possible to specify pointcuts, advice and optionally an order. A parser has to be provided for this language, which will create the pointcuts and advice which are handed over to the weaver.

The pointcuts and advice that have to be created have to be specific for the base language, this means that a subclass of Advice and one or more subclasses for Pointcut have to be created. The advice subclass is simple, it must contain the body of the advice. The pointcuts need to contain all information to identify a join point. Since there will be multiple types of pointcuts, one subclass has to be implemented for each type of pointcut. If desired a context can be added to the pointcut, by creating a class with the Context interface that contains all the context information to be used.

The pointcuts and advice of course requires the developer to implement a type system. Depending on the availability of the type system used by the base language, and more precisely the compiler, it is possible to add small modifications to get the type system without

much work. If not, this might be a bigger problem.

Now we can create pointcuts and advice, we still need to expand the Weaver class to add the actual weaving. Since the framework gives us the advice to execute for each join point all the extended weaver has to do is generate code and weave the advice with the source code. This sounds easier than it sometimes is: It is especially important how we can interact with the base language. Eventually it all comes down to putting all the information in the weaver and generating the woven code. A schematic overview of this is shown in figure 3.9; note that the aspect language is split up into three components. It is possible to have this explicit separation, but most of the time this will cause more work to implement and thus not be this explicit.

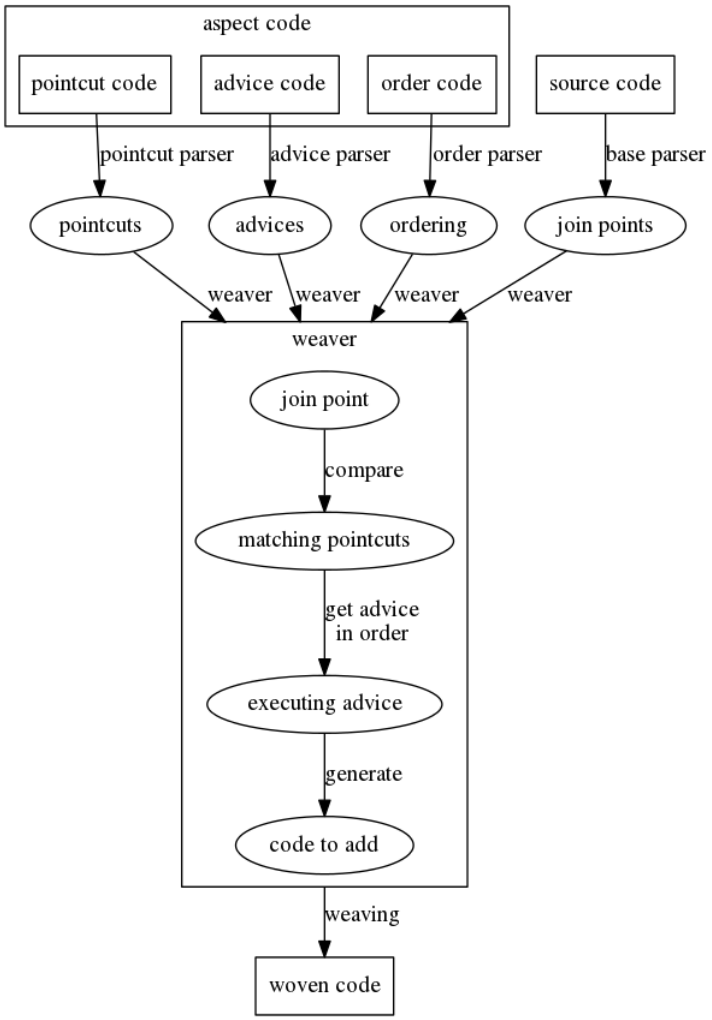

Figure 3.9: Schematic overview of the weaving process.

CHAPTER 4

Small C

To demonstrate the use of our framework in practice, it has been instantiated for a number of base languages. In this section the framework is applied to Small C.

Small C is a subset of the procedural C language. Though it does not support structs or including other files, it does give us a good overview of what aspect-orientation in a procedural programming language is like. As the source code of a compiler for this language was already available, this made Small C an interesting first step to test the framework.

Despite the resemblance with C, I specify an example in listing 4.1 to show the normal structure of a Small C program. This example will also be used later to demonstrate the functionality of the framework.

```
1  #include <stdio.h>
2
3  int count = 0;
4
5  int mySquare(int a) {
6    return a * a;
7  }
8
9  int mySqrt(int a) {
10   int i = 1;
11   while (mySquare(i) <= a) {
12     i = i + 1;
13   }
14   return i - 1;
15 }
16
17 void main () {
18   int a = 2;
19   int b = 10;
20
21   x = mySquare(a);
22   count = count + 1;
23   x = mySqrt(a);
24   count = count + 1;
25   x = mySqrt(25);
26   count = count + 1;
27   x = mySquare(count);
28   count = count + 1;
29 }
```

Listing 4.1: A simple Small C program.

4.1 Join Point Model

Due to the minimalistic character of the language the join point model is limited too. The join points identified can be divided into two groups, each with two types. An overview of

the join points is given in table 4.1.

Join Point	Type	Description
Function	Call	The moment a function is called.
	Execution	The moment a function is executed
Global Variable	Set	The moment a global variable is set.
	Get	The moment a global variable is accessed.

Table 4.1: Small C join points.

4.2 Base Language Compiler

Because the join points match some of the basic elements of the language, it was easy to find the interesting points in the compiler. At these points the compiler would already gather all the required information as it needed them itself to build the AST. This minimizes the code that was required to add, leaving only some extra code to create the specific join point and to convert arguments to a type that could be used by the framework.

4.3 Aspect Language

Before discussing the parts of the framework it is required to explain some basic elements of the language. Though the language is mostly based on AspectJ, there are some differences mostly caused by the attempt to keep the language as simple as possible.

In order to address multiple variables or functions simultaneously, the concept of wildcards is often introduced. This is also the case for this language, the '..' wildcard allows the user to specify a space that can be covered by any 0 or more characters.

The compiler I used already had an entire type system, which could easily be modified to be used for the framework too. All that was required was making the CType class implement the Type interface. To support wildcards an extra type, the AnyType, was added. Note that it is impossible to partly specify a type such as '..int' which could match types such as longint, shortint and int. This is not a problem since the only types allowed in Small C are: 'int', 'float', 'char', 'void' and 'boolean' in combination with the array and pointer additions. An overview of the type system is shown in figure 4.1.

A complete overview of the language in EBNF notation can be seen in listing 4.2. Examples of the language will be given later, when each part is explained.

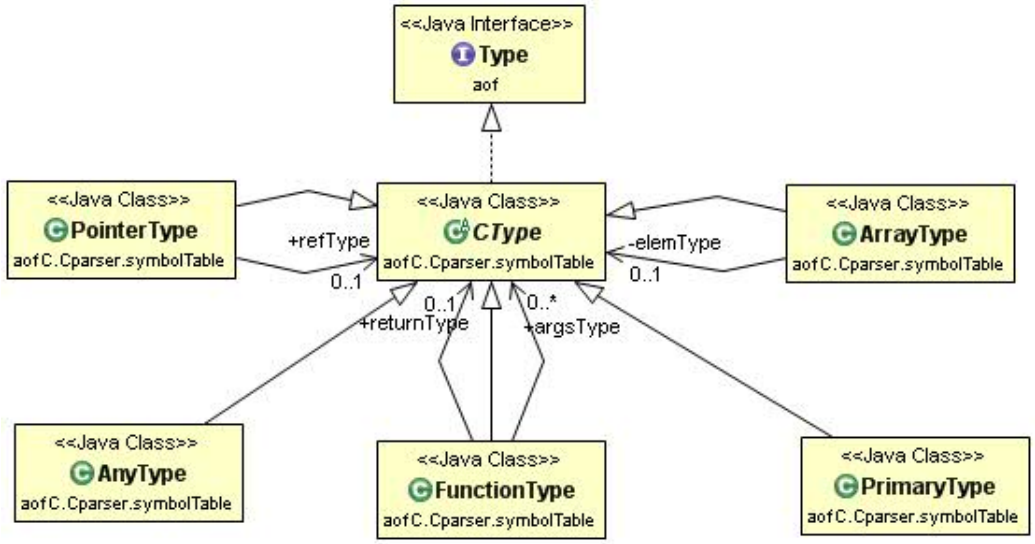

Figure 4.1: UML diagram of the CType class and its subclasses.

```
1  file ::= (WS | order | pointcut | advice)* EOF
2  pointcut ::= 'pointcut' WS NAME WS? '(' args ')' WS? '{' rules '}'
3  rules ::= (WS | method | member)*
4  member ::= ('set' | 'get') WS type WS NAME WS? ';'
5  method ::= ('call' | 'execute') WS type WS NAME WS? '(' args ')' WS? ';'
6  advice ::= ('advice' WS NAME WS? ':' WS? )? ('before' | 'after' | 'around') WS
       NAME WS? '(' args ')' WS? '{' .* '}'
7  order ::= 'order' WS? '{' WS? NAME (WS? ';' WS? NAME)* WS? '}'
8  args ::= (type (WS NAME)? (WS? ',' WS? type (WS NAME)? )* )?
9  type ::= NAME ('[' ']' | '*')?
10
11 NAME ::= (LETTER | SPECIALCHARS | '..') (LETTER | DIGIT | SPECIALCHARS | '..')
       *
12 WS ::= (' ' | '\n' | '\t' | '\r')+
13 OTHERCHARS ::= ~(LETTER | DIGIT | SPECIALCHARS)
14 DIGIT ::= '0..9'
15 LETTER ::= ('a..z' | 'A..Z')
16 SPECIALCHARS ::= ('*' | '_')
```

Listing 4.2: EBNF notation of the aspect langauge.

4.4 Pointcuts

All the kind of join points are mapped to pointcuts in a one-on-one manner. Which means that there two pointcut classes, being MethodPointcut and MemberPointcut. The different types are handled by adding a flag to the pointcut. Because identifying a method or a global variable is completely different we need to separate these two. However, because a call and an execution both execute on a method we can combine these two together. This also holds for the get and set of a global variable.

To identify a method join point we need the signature of the method, consisting of the

name of the method, the return type and a set of arguments. The arguments are already implemented by the abstract super class, and thus are not part of the MethodPointcut class. A UML diagram for the MethodPointcut class is shown in figure 4.2. Wildcards can be used in every element of the pointcut: in the name of the method, as a type of an argument, as return type or as part of the argument list. Examples of the pointcuts are shown later when discussing each type in more detail. For now I show a couple of examples of how wildcards can be used in listing 4.3.

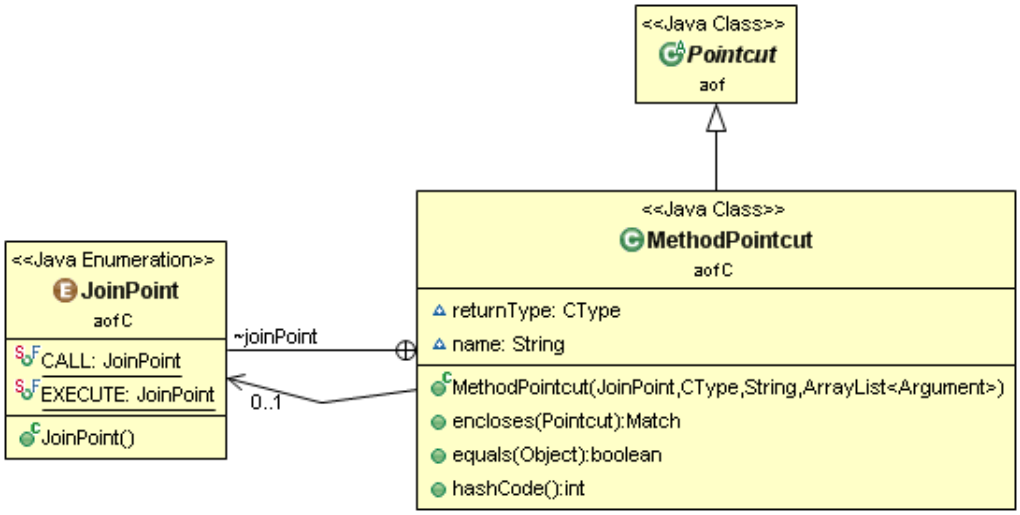

Figure 4.2: UML diagram of the MethodPointcut class.

```
1  pointcut p() {
2    call int my..();
3  }
4
5  pointcut q() {
6    call int mySquare(..);
7  }
8
9
```

```
10 pointcut r() {
11   get int ..;
12 }
13
14 pointcut s() {
15   get .. count;
16 }
```

Listing 4.3: Some examples of wildcards.

It is not allowed to use a wildcard in the name of the arguments. The first reason is that the names are used to map them onto the arguments specified in the pointcutset. The second is that these will be passed on to the advice to be used there. Doing this wouldn't make any difference because the arguments are only matched on type and not on name. It is also required that a pointcut contains all the arguments the enclosing pointcutset defines. An example of two method pointcuts are shown in listings 4.4 and 4.5, these pointcuts match respectively the calling and the execution of a method mySquare.

```
1  pointcut p() {
2    call int mySquare(int);
3  }
```

Listing 4.4: Example of a method call pointcut.

```
1  pointcut p() {
2    execute int mySquare(int);
3  }
```

Listing 4.5: Example of a method execution pointcut.

The MemberPointcut is pretty empty, simply because the best way to identify a global variable is to see it as an argument of the pointcut. By doing this the only thing the MemberPointcut class has to add is the flag to identify the different types, as can be seen in the UML diagram shown in figure 4.3. Just as with the MethodPointcut it is possible to use a wildcard as type, but it is also allowed to specify a wildcard as (part of) the name of the member to match multiple global variables. Because a MemberPointcut only contains one argument, being the member, it is always this argument that is mapped to the pointcutset argument. Since every pointcut needs to declare each pointcut defined by the pointcutset, a pointcutset that contains a member pointcut can only have one argument. An example of two member pointcuts are shown in listings 4.6 and 4.7, these pointcuts match respectively the getting and setting of a global variable count.

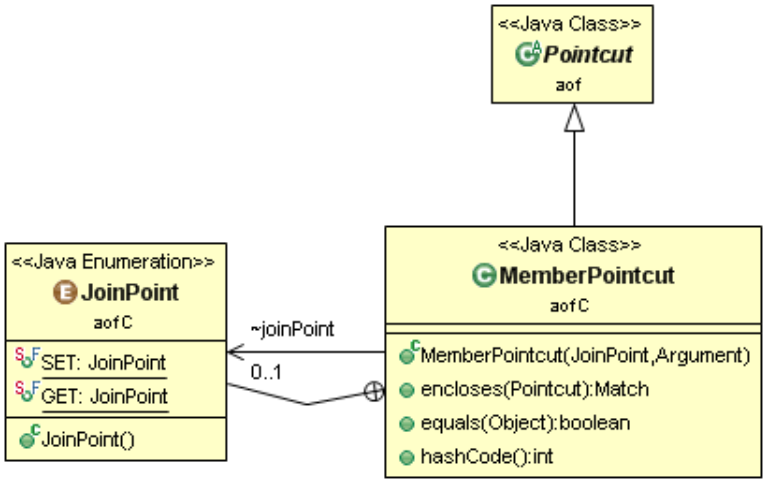

Figure 4.3: UML diagram of the MemberPointcut class.

```
1 pointcut p() {
2   get int count;
3 }
```

Listing 4.6: Example of a member get pointcut.

```
1 pointcut p() {
2   set int count;
3 }
```

Listing 4.7: Example of a member set pointcut.

The current implementation does not use the context of the framework, which limits usability. However, I am convinced that adding a context would not add much functionality since the only context in Small C is the call hierarchy. Note that due to the lack of a context in the pointcuts the method call and execution pointcut will yield the same result, with a context we could for instance get the current method and use this in the advice.

4.5 Advice

The advice is implemented as the CAdvice class shown in figure 4.4. This class contains a string which represents Small C code. The code is allowed to contain arguments that are defined by the pointcutset it is linked to, but these are not treated specially in the CAdvice class itself. Besides the code, the CAdvice class also contains a flag to indicate the type.

We distinguish three types of advice, specifying the moment of execution: before, after and around. These are the same as are present in AspectJ, and are straightforward. These types break the clear separation between pointcuts and advice and can be considered to be bad programming, yet this way we can create pointcutsets that are more reusable.

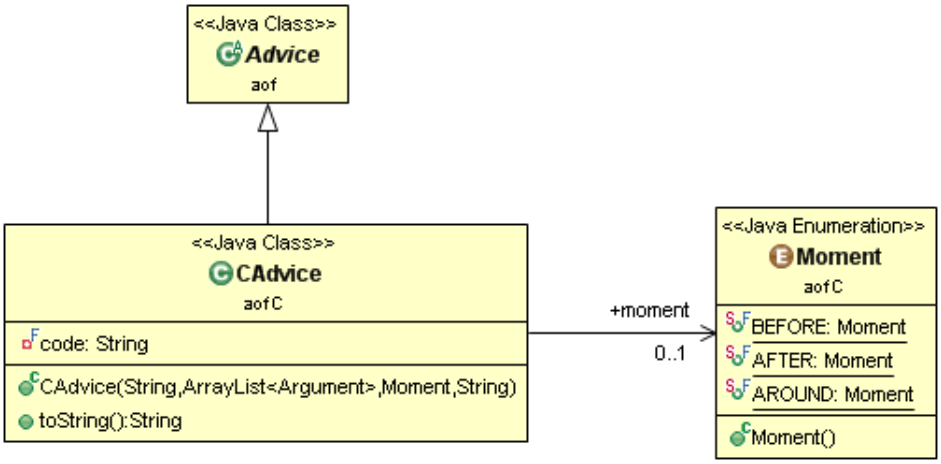

Figure 4.4: UML diagram of the CAdvice class.

While the before and after can only be used to add extra code, the around advice can be used to skip the original code, or modify the values of the parameters. The **proceed()** call can be used to jump to the original join point (or the next advice, if any). Join points getting or setting a global variable are seen as a typical getter or setter and are treated the same as methods. Note that if the join point has a return type, either a method with a return type or a getter for a variable, the value will be returned by the **proceed()** call and it is the responsibility of the user to capture this.

```
1 advice b: before p() {
2   printf("Advice B\n");
3 }
```

Listing 4.8: Example of a before advice.

```
1 advice a: after p() {
2   printf("Advice A\n");
3 }
```

Listing 4.9: Example of an after advice.

```
1 advice c: around p() {
2   printf("BEGIN C!\n");
3   proceed();
4   printf("END C!\n");
5 }
```

Listing 4.10: Example of an around advice.

An advice needs to be linked with a pointcutset, this is done by referring to it's name and specifying the arguments it requires. Note that the arguments need to match on type, but may differ on name. The variables declared by the linked pointcutset can be used in the body of the advice.

```
1 pointcut p(int i) {
2   call int mySquare(int i);
3 }
4
5 advice a: before p(int v) {
6   printf("BEGIN: %i\n", v);
7 }
```

Listing 4.11: Example of arguments.

The body of the advice is not parsed in any way at the moment it is created. It is not even parsed anywhere in the entire process until the very end which is simply compiling the woven code. This makes it possible to achieve communication between advice by using global variables of the original source code.

4.6 Order

There are two situations in which multiple advice can be mapped to the same moment in execution. The most obvious situation is when multiple advice are linked to the same pointcutset, but also if we have multiple pointcutsets that happen to match the same join point. In both cases it is possible that advice give rise to conflicts, but only if their types overlap.

First, we will discuss the natural order of the advice. If there is no conflict the order is determined by the type of the advice, if there is a conflict but no order, it's up to the framework to resolve it. The location of the advice compared to the join point is shown in figure 4.5. An overview of all the possible conflicts is more convienently shown in table 4.2.

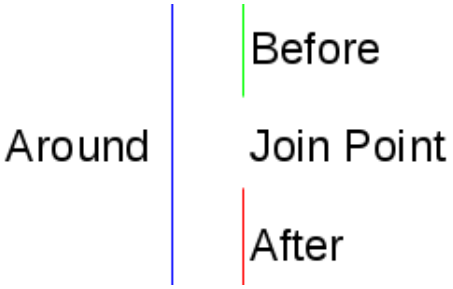

Figure 4.5: The location of the advice.

	before	after	around
before	+	-	+
after	-	+	+
around	+	+	+

Table 4.2: An overview of the possible conflicts.

To determine the order when conflicts do arise it is possible to specify an order between any two advice, even those that can never cause a conflict. This may seem useless, but it becomes useful in cases where we want more than two advice to be in a specific order and

they are of different types. An order is specified as a list of advice, in which a later advice is closer to the join point than an earlier one. An example of specifying an order is shown in listing 4.12 and listing 4.13: in this example *jp* is used to denote the join point. The order specified here compares a before and after advice but will have no effect as they can never cause a conflict. Note that it is possible to switch the before and after advice in the order and still get the same result.

```
order{c; b; a}

advice b: before p() {
  printf("Advice B\n");
}

advice a: after p() {
  printf("Advice A\n");
}

advice c: around p() {
  printf("BEGIN C!\n");
  proceed();
  printf("END C!\n");
}
```

Listing 4.12: An example of a non-trivial order.

```
printf("BEGIN C!\n");
printf("Advice B\n");
jp;
printf("Advice A\n");
printf("END C!\n");
```

Listing 4.13: The result of the order.

The example shown in listing 4.14 and 4.15 shows an implicit order between a before and after advice where an after advice comes before a before advice.

```
order{a; c; b}

advice b: before p() {
  printf("Advice B\n");
}

advice a: after p() {
  printf("Advice A\n");
}

advice c: around p() {
  printf("BEGIN C!\n");
  proceed();
  printf("END C!\n");
}
```

Listing 4.14: An example of a non-trivial order.

```
printf("BEGIN C!\n");
printf("Advice B\n");
jp;
printf("END C!\n");
printf("Advice A\n");
```

Listing 4.15: The result of the order.

4.7 Weaver

The weaver is the largest piece of code. After all it is the core part of every aspect-oriented language. Once it has all the information it can start putting it together; it does this by writing wrapper methods for the join points, even if there is no advice to be execute for this join point. This will cause writing a lot of dummy wrappers, which do nothing more than just forwarding the call. These wrappers could have been omitted, but for simplicity they are not. Which wrappers are generated depends on the join point, and partly on the advice. For every global variable declared there will be two wrappers, one to handle the access to

the variable, and another for the assignment of a new value. The same holds for a method but with call and execution instead. An example of these wrappers is shown in listing 4.16 and 4.17. The original operations will be modified so that they use the wrappers instead. An example of these changes is shown in listing 4.18 and 4.19.

```
int count_get_() {
  int _result_;
  _result_ = count;
  return _result_;
}

void count_set_(int _new_) {
  count = _new_;
}
```

Listing 4.16: Example of variable wrappers.

```
int mySqrt_exec_(int a) {
  int _result_;
  _result_ = mySqrt(a);
  return _result_;
}

int mySqrt_call_(int a) {
  int _result_;
  _result_ = mySqrt_exec_(a);
  return _result_;
}
```

Listing 4.17: Example of method wrappers.

```
x = mySqrt(25);
count = count + 1;
x = mySquare(count);
```

Listing 4.18: Example of code before weaving.

```
x = mySqrt_call_(25);
count_set_(count_get_() + 1);
x = mySquare_call_(count_get_());
```

Listing 4.19: Example of code after weaving.

Beside these wrappers some other methods are generated as well. One for each advice of the type before or after, and one for each matching join point of an around advice. The reason the around advice needs multiple instances is that it might have to proceed to the original method. While this is not the case for the before and after advice, it is still much easier to put them in a separate method. Especially considering that such an advice can alter the values of arguments, but that these changes may not be passed on to the actual method. By placing them in a method this behavior is ensure by the Small C language. Listen 4.20 shows the generated methods for the advice specified in listings 4.8, 4.9 and 4.10.

```
void _a() {
  printf("Advice A\n");
}

void _b() {
  printf("Advice B\n");
}

void _c_myMethod_() {
  printf("BEGIN C!\n");
  {
     myMethod();
  }
  printf("END C!\n");
}
```

Listing 4.20: Generated methods.

Beside generating wrappers, the weaver is also responsible for delivering the right arguments to the advice. This is another situation that shows the advantages of putting an advice in a separate method, as we can keep the names that are used in the advice code. If the around advice contains a proceed() call, the weaver has to replace this with a call to the original method or the next around advice. The arguments specified in the proceed must be preserved and passed on. In order not to complicate the weaver any further it is required

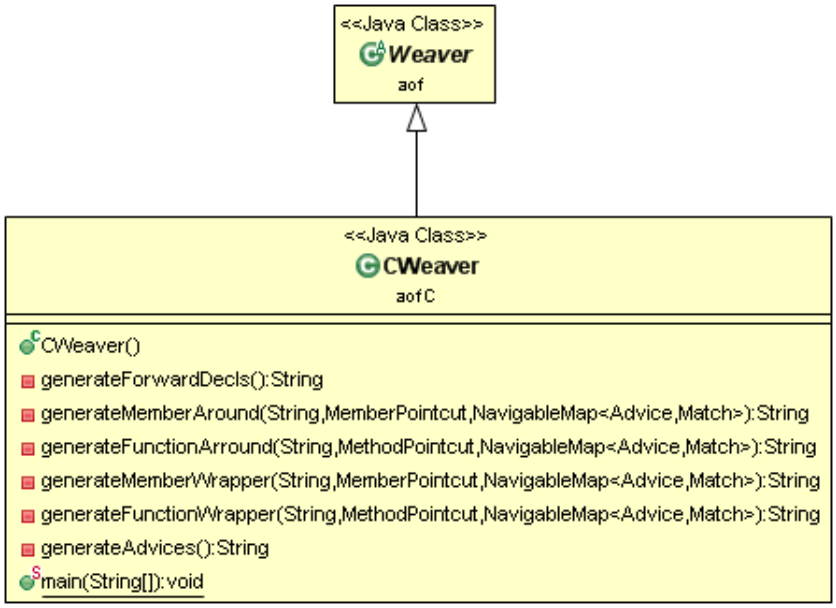

Figure 4.6: UML diagram of the CWeaver class.

that the user manages the declaration of a value to store the return result and explicitly return this value if a return value is required. The declaration, assigning the return value of the proceed() and returning the value all must occur separately. This is because it is possible that the weaver will insert calls to other advice before and after the proceed() call. An example of this is shown in listing 4.21, where the around advice is generated using the order as specified in listing 4.12.

```
1  int _c_mySquare_(int a) {
2    int retVal = 0;
3    printf("BEGIN C!\n");
4    {
5      _b();
6      retVal = mySquare(a);
7      _a();
8    }
9    printf("END C!\n");
10   return retVal;
11 }
```

Listing 4.21: A generated method for an around advice with advice nested within.

Now that all the components have been introduced, I give a final overview of the entire structure in figure 4.7 to make it extra clear how all the components are connected.

4.8 Example

In this section a complete example is shown to get a better feeling of how all the pieces fit together.

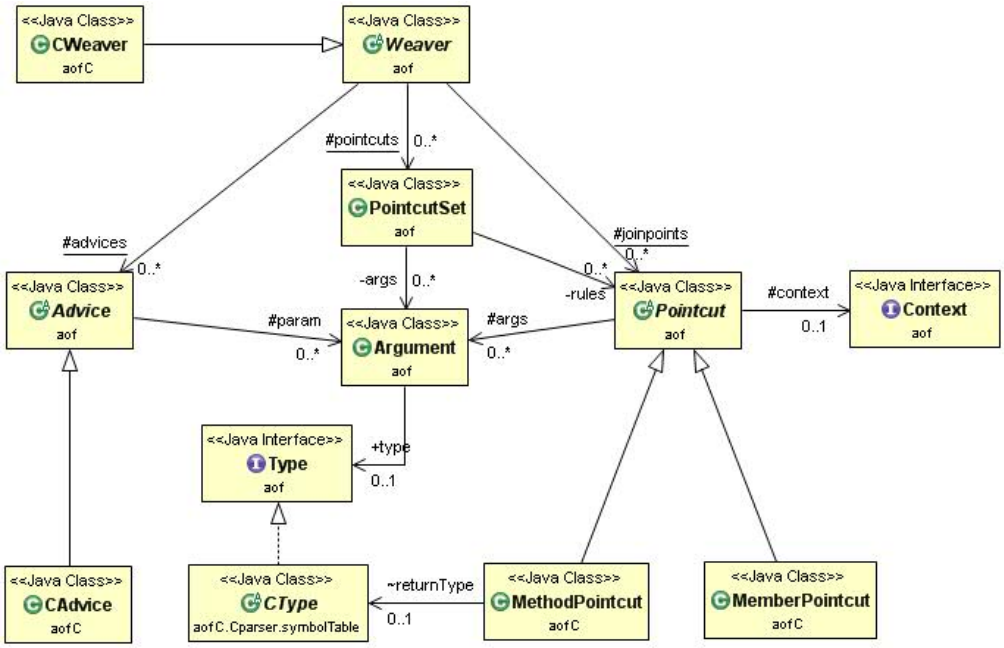

Figure 4.7: A full overview of the use of the framework for Small C.

```
#include <stdio.h>

int count = 0;

/* a^b */
int myPow(int a, int b) {
  int x;
  int result = 1;

  if (b <= 0) {
    return 1;
  }

  for (x = 1; x <= b; x = x + 1) {
    result = result * a;
  }

  return result;
}

/* a^2 */
int mySquare(int a) {
  return a * a;
}

/* square root of a. */
int mySqrt(int a) {
  int i = 1;
  while (mySquare(i) <= a) {
    i = i + 1;
  }

  return i - 1;
}

/* b-th root of a. */
int myRoot(int a, int b) {
  int i = 1;
  while (myPow(i, b) <= a) {
    i = i + 1;
  }

  return i - 1;
}

/* log_b(a) */
int myLog(int a, int b) {
  int i = 1;
  while (myPow(b, i) <= a) {
    i = i + 1;
  }

  return i - 1;
}

void main () {
  int a = 2;
  int b = 10;

  int x = myPow(a, b);
  count = count + 1;
  x = mySquare(a);
  count = count + 1;
  x = mySqrt(a);
  count = count + 1;
  x = mySqrt(25);
  count = count + 1;
  x = myRoot(1024, 2);
```

```
  count = count + 1;
  x = myLog(1024, 2);
  count = count + 1;
  x = mySquare(count);
}
```

```
order{b; c; a}

pointcut p(int x) {
  call int my..(int x, ..);
}

pointcut q(int i) {
  call int random(int i);
}

pointcut v(int count) {
  get int count;
}

advice getv: before v(int i) {
  printf("Count = %i\n", i);
}

advice a: after p(int i) {
  printf("BEGIN A\n");
}

advice b: after p(int i) {
  printf("BEGIN B\n");
}

advice c: around p(int i) {
  int retVal = 0;
  printf("BEGIN C!\n");
  retVal = proceed(i);
  printf("END C!\n");
  return retVal;
}
```

Listing 4.22: Source code

Listing 4.23: The aspect code.

```
#include <stdio.h>

int count = 0;
int myPow(int a, int b) {
  int x = 0;
  int result = 1;
  if (b <= 0) {
    return 1;
  }

  for (x = 1; x <= b; x = x + 1) {
    result = result * a;
  }

  return result;
}

int mySquare(int a) {
  return a * a;
}

int mySqrt(int a) {
  int i = 1;
  while(mySquare_call_(i) <= a) {
    i = i + 1;
  }

  return i - 1;
}

int myRoot(int a, int b) {
  int i = 1;
  while(myPow_call_(i, b) <= a) {
    i = i + 1;
  }

  return i - 1;
}

int myLog(int a, int b) {
  int i = 1;
  while(myPow_call_(b, i) <= a) {
    i = i + 1;
  }

  return i - 1;
}

void main() {
  int a = 2;
  int b = 10;
  int x = myPow_call_(a, b);
  count_set_(count_get_() + 1);
  x = mySquare_call_(a);
  count_set_(count_get_() + 1);
  x = mySqrt_call_(a);
  count_set_(count_get_() + 1);
  x = mySqrt_call_(25);
  count_set_(count_get_() + 1);
  x = myRoot_call_(1024, 2);
  count_set_(count_get_() + 1);
  x = myLog_call_(1024, 2);
  count_set_(count_get_() + 1);
  x = mySquare_call_(count_get_());
}

void _getv(int i) {
  printf("Count = %i\n", i);
}

void _b(int i) {
  printf("BEGIN B\n");
}

void _a(int i) {
  printf("BEGIN A\n");
}
```

```
79  int count_get_() {
80    int _result_;
81    _getv(count);
82    _result_ = count;
83    return _result_;
84  }
85
86  int myPow_exec_(int a, int b) {
87    int _result_;
88    _result_ = myPow(a, b);
89    return _result_;
90  }
91
92  void main_call_() {
93    main_exec_();
94  }
95
96  int myLog_exec_(int a, int b) {
97    int _result_;
98    _result_ = myLog(a, b);
99    return _result_;
100 }
101
102 int _c_myLog_(int i, int b) {
103   int retVal = 0;
104   printf("BEGIN C!\n");
105   {
106     retVal = myLog_exec_(i, b);
107     _a(i);
108   }
109
110   printf("END C!\n");
111   return retVal;
112 }
113
114 int myLog_call_(int a, int b) {
115   int _result_;
116   _result_ = _c_myLog_(a, b);
117   _b(a);
118   return _result_;
119 }
120
121 int mySqrt_exec_(int a) {
122   int _result_;
123   _result_ = mySqrt(a);
124   return _result_;
125 }
126
127 int _c_myRoot_(int i, int b) {
128   int retVal = 0;
129   printf("BEGIN C!\n");
130   {
131     retVal = myRoot_exec_(i, b);
132     _a(i);
133   }
134
135   printf("END C!\n");
136   return retVal;
137 }
138
139 int myRoot_call_(int a, int b) {
140   int _result_;
141   _result_ = _c_myRoot_(a, b);
142   _b(a);
143   return _result_;
144 }
145
146 int mySquare_exec_(int a) {
147   int _result_;
148   _result_ = mySquare(a);
149   return _result_;
150 }
151
152 int _c_mySqrt_(int i) {
153   int retVal = 0;
154   printf("BEGIN C!\n");
155   {
156     retVal = mySqrt_exec_(i);
157     _a(i);
158   }
159
160   printf("END C!\n");
161   return retVal;
162 }
163
164 int mySqrt_call_(int a) {
165   int _result_;
166   _result_ = _c_mySqrt_(a);
167   _b(a);
168   return _result_;
169 }
170
171 int _c_myPow_(int i, int b) {
172   int retVal = 0;
173   printf("BEGIN C!\n");
174   {
175     retVal = myPow_exec_(i, b);
176     _a(i);
177   }
178
179   printf("END C!\n");
180   return retVal;
181 }
182
183 int myPow_call_(int a, int b) {
184   int _result_;
185   _result_ = _c_myPow_(a, b);
186   _b(a);
187   return _result_;
188 }
189
190 void main_exec_() {
191   main();
192 }
193
194 int myRoot_exec_(int a, int b) {
195   int _result_;
196   _result_ = myRoot(a, b);
197   return _result_;
198 }
199
200 void count_set_(int _new_) {
201   count = _new_;
202 }
203
204 int _c_mySquare_(int i) {
205   int retVal = 0;
206   printf("BEGIN C!\n");
```

```
207   {
208     retVal = mySquare_exec_(i);
209     _a(i);
210   }
211
212   printf("END C!\n");
213   return retVal;
214 }
215
```

```
216 int mySquare_call_(int a) {
217   int _result_;
218   _result_ = _c_mySquare_(a);
219   _b(a);
220   return _result_;
221 }
```

Listing 4.24: Woven code

4.9 Difficulties

Despite having a modifiable compiler, there were a lot of problems of weaving the advice into the code. This was caused by the fact that the compiler was not designed to allow changes after parsing the code. Changing this would require big changes in the compiler and was therefore not really an option. Instead of changing the compiler another alternative was used, by creating two compilers, both with small changes, we could weave the file in multiple passes.

The phases in which the woven code is generated are:

1. The source code will be parsed and all join points will be specified to the weaver. The weaver will produce wrappers and add these to a temporary output file.
2. The modified file will be parsed and the original calls will be modified to point to the wrappers.
3. The actual advice bodies will be written to the wrappers.
4. Compile the woven file with a regular compiler, this is not done by the weaver and has to be done by the user himself.

It was impossible to change the calls in the first phase due to a simple circular dependency. The compiler checks that the called function actually exists. Since the compiler does not allow to add declarations before the current point, these declarations have to exist in the source code itself. To do this however we need to know which join points occur, and thus we need to have parsed the file before.

Adding the content can not be done before phase 2, simply because the content contains function calls to the actual functions which need to be kept. Due to the way the compiler handles the code, it is impossible to distinguish a call from within a wrapper from one within a 'normal' function, yet we need to treat them separately. Therefore we need to add the content in a separate phase.

CHAPTER 5

Dot

In this section the framework is applied to Dot. Dot is a language that allows the user to easily specify the structure and appearance of a graph. Some of the features the language supports are directed edges, undirected edges, subgraphs, labels for nodes and edges, different shapes, colors and more. Despite the fact that the language is quite easy-to-use, and has a straightforward syntax, it tends to be very verbose when trying to create large and complicated graphs.

It is important to note that, unlike a typical aspect-oriented language, Dot does not have any notion of time or behavior as the result of the 'compiler' is the graph that the user wants. Because of this adding AOP may seem somewhat strange since join points are defined as points during the execution of a program. Since the program does not execute, but only describes a static graph structure, all the join points simply map to points in the code in a static way. However, most ideas of AOP can still be applied.

Because the syntax of Dot is that self-explanatory, the example showed in listing 5.1 and the resulting graph in figure 5.1 is enough to get an understanding of the language.

5.1 Join Point Model

The join points identified in Dot are all the elements of the graph, being the graph itself, its nodes and edges. This simple join point model is mostly caused by the lack of run-time behavior.

```
digraph G {
  subgraph cluster_0 {
    style=filled;
    color=lightgrey;
    node [style=filled,color=white
    ];
    a0 -> a1 -> a2 -> a3;
    label = "process #1";
  }

  subgraph cluster_1 {
    node [style=filled];
    b0 -> b1 -> b2 -> b3;
    label = "process #2";
    color=blue
  }

  start -> a0;
  start -> b0;
  a1 -> b3;
  b2 -> a3;
  a3 -> a0;
  a3 -> end;
  b3 -> end;

  start [shape=Mdiamond];
  end [shape=Msquare];
}
```

Listing 5.1: The code to create a graph.

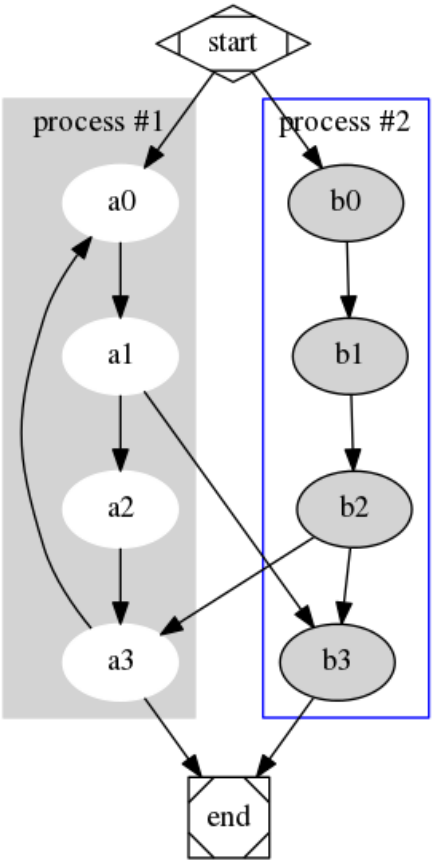

Figure 5.1: The generated graph by dot.

5.2 Base Language Compiler

The compiler for the base language contains a parser that makes it possible to request all the graphs, nodes and edges present in the file. Because of this no modifications were made to the base compiler.

5.3 Aspect Language

The aspect language mostly uses the same syntax as Dot itself to keep the language as simple as possible. There exists a wildcard to create more general rules, being '..'. By using this the programmer specifies that anything can take its place.

Just as in Dot itself a graph, node and edge can contain attributes. These attributes are considered to be the arguments of the element and are used to restrict the matching. An attribute needs to have a name, being the attribute itself, and a value. For this value the wildcard can be used. It is also important to note that the wildcard can be used to denote any other attribute. Without this only elements with exactly those attributes are matched. Note that these attributes only act as arguments for the element and not for the pointcutset.

```
1 node [label=..];
2 node [label="label", ..];
3 node [..];
4 .. [..];
```

Listing 5.2: Examples of using the wildcard.

It are the elements (node, edge, graph) that can be used as arguments, but to prevent having to make an entire type system for it, the id's of the elements are used instead. This makes it possible to keep the type system very basic, as can be seen in figure 5.2.

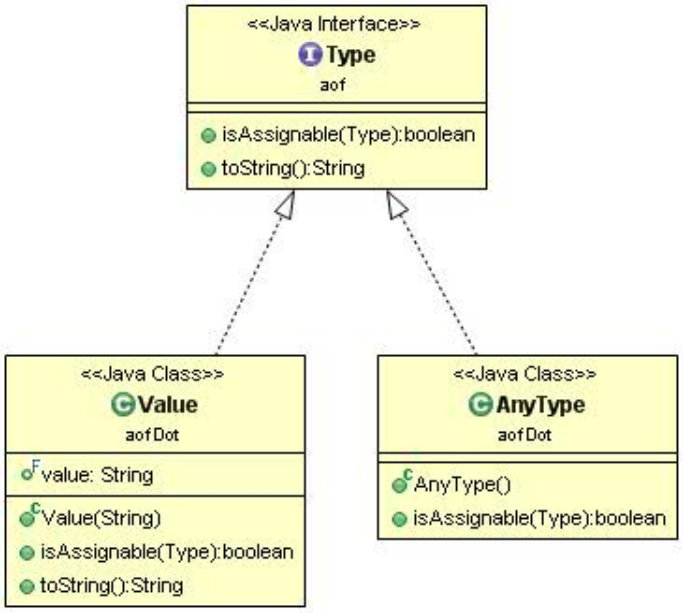

Figure 5.2: UML diagram of the type system for aspect-orientation in Dot.

There is no context information used, though it is possible to match elements based on the greater picture, being the graph they are part of. This will become clear when explaining the pointcuts and more specific the GraphPointcut.

A complete overview of the language in EBNF notation is shown in listing 5.3.

```
file ::= (WS | order | pointcut | advice)* EOF
order ::= 'order' WS? '{' WS? NAME (WS? ';' WS? NAME)* WS? '}'
pointcut ::= 'pointcut' WS NAME WS? '(' args ')' WS? '{' rules '}'
advice ::= ('advice' WS NAME WS? ':' WS? )? ('insert' | 'delete') WS NAME WS?
    '(' args ')' WS? '{' rules '}'
args ::= (WS? type WS NAME (WS? ',' WS? type WS NAME)* WS? )?

rules ::= (WS | graph | node ';' | edge ';')*
graph ::= ('..graph' | 'graph' | 'digraph' | 'subgraph') WS (NAME | '..') WS?
    '{' rules '}'
edge ::= (NAME WS? '=' WS? )? '(' WS? node WS? ')' WS? ('->' | '--') WS? '('
    WS? node WS? ')' (WS? '[' attributes ']')?
node ::= (NAME | '..') (WS? '=' WS? (NAME | '..'))? (WS? '[' attributes ']')?
attributes ::= (NAME WS? '=' WS? (NAME | '..') (WS? ',' WS? attributes)? |
    '..')
type ::= ('Node' | 'Edge' | 'Graph')

NAME ::= '..'? ((LETTER | DIGIT | '_' | '"' .* '"') '..'? )*
WS ::= (' ' | '\n' | '\t' | '\r')+
DIGIT ::= '0..9'
LETTER ::= ('a..z' | 'A..Z')
```

Listing 5.3: EBNF notation of the aspect language.

5.4 Pointcuts

There exist three kind of pointcuts being GraphPointcut, NodePointcut and EdgePointcut which represent respectively a graph, node and edge. All of these pointcuts can have Dot attributes, which will be stored as arguments and thus are part of the superclass Pointcut. Because each pointcut also has specific information I will now discuss these three pointcuts in more detail.

A NodePointcut is the easiest as it only has a name and attributes, and since attributes are already handled by the superclass all we need to do is add a name. A node matches a NodePointcut if it has the same name and all the attributes match. Two examples of such a node poincut are shown in listing 5.4 and 5.5.

```
pointcut n() {
  start [..];
}
```

Listing 5.4: A node pointcut matching any node start.

```
pointcut n() {
  .. [color=..];
}
```

Listing 5.5: A node pointcut matching any node with only a color.

An EdgePointcut isn't that much harder, as the compiler relates every edge to a source and target node. These two nodes are again represented by a NodePointcut, as this was the easiest solution. On top of that we also need to know whether the edge is directed or not, which is a simple boolean. An edge matches an EdgePointcut if they are both directed and the sources and targets match or if they are both undirected, the source and target node may be switched. Two examples of an edge pointcut are shown in listing 5.6 and 5.7.

```
pointcut n() {
  (start [..]) -> (main [..]) [..];
}
```

Listing 5.6: An edge pointcut matching any edge from start to main.

```
pointcut n() {
  (.. [..]) -> (main [..]) [label
    =.., ..];
}
```

Listing 5.7: An edge pointcut matching any edge to main with a specified label.

The GraphPointcut requires all the information of its nodes and edges. Instead of introducing new types for nodes and edges, it is easier to reuse the pointcuts to represent this. A graph also has a name, though this information is not used in the current version. Checking whether a graph matches with a GraphPointcut is very simple, we just need to compare the nodes and edges it contains. A wildcard can be used to indicated that the graph may contain more nodes and edges. Two examples of a graph pointcut are shown in listing 5.8 and 5.9.

```
pointcut n() {
  graph G {
    (start [..] ->  main [..])
    [..];
    .. [..];
    (.. [..]) -> (.. [..]) [..];
  }
}
```

Listing 5.8: A graph pointcut matching any graph G with an edge from start to main.

```
pointcut n() {
  graph G {
    start [..];
    main [..];
    exit [..];
    (.. [..]) -> (.. [..]) [..];
  }
}
```

Listing 5.9: A graph pointcut matching any graph G with only three nodes.

An overview of all the pointcuts and their interactions is shown in figure 5.3.

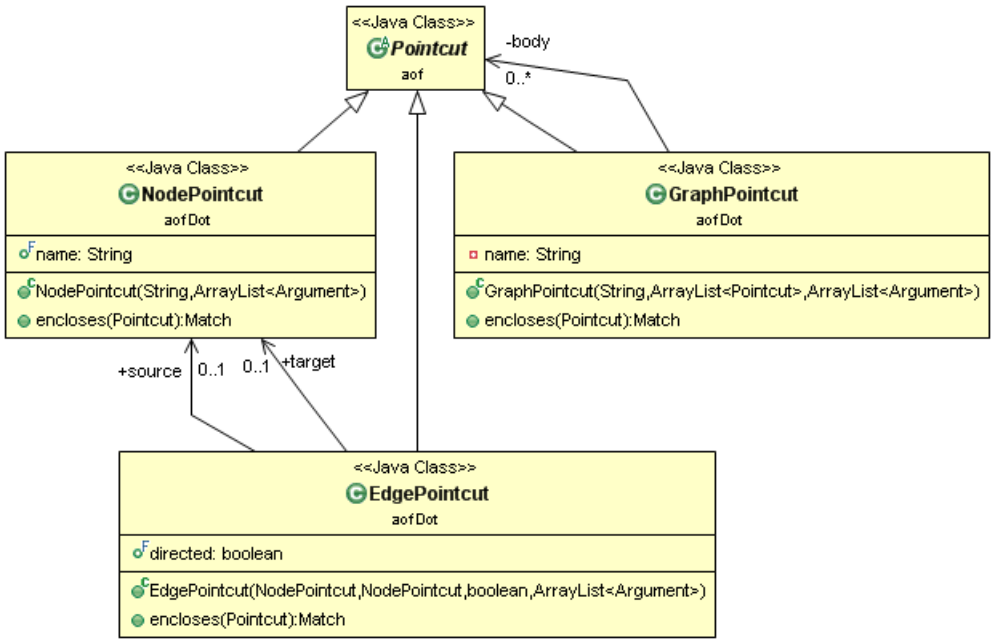

Figure 5.3: UML diagram of the three pointcut classes.

Arguments can be introduced on every level of the pointcut, which is shown by the examples in listing 5.10. Though the pointcut expresses the type of the argument, this is not really

used by the weaver. As mentioned earlier it is the id of the elements that are used internally.

```
pointcut n(Edge i, Node j) {
  graph .. {
    i = (main [..]) -> (j = .. [..]) [..];
    .. [..];
    (.. [..]) -> (.. [..]) [..];
  }
}

pointcut o(Node i) {
  i = .. [..];
}

pointcut p(Edge i) {
  i = (main [..]) -> (.. [..]) [..];
}
```

Listing 5.10: Examples showing the use of arguments.

The current version does not use the context mechanism offered by the framework. It is however possible for the user to specify a form of context by using a graph pointcut. We can use an element inside this graph as an argument. Because the pointcut only matches that certain graph, we have specified some context information for the element we can use as an argument.

5.5 Advice

There are two kinds of advice: insert and delete, which are meant to respectively insert elements or delete them from the graph. It is possible to add attributes to an element simply by creating an element that already exists but with the new attributes. Modifying an attribute is done in the same way. Deleting an attribute is done by specifying an attribute next to the element, if no such attribute is specified the entire element is deleted. By Using arguments it is possible to specify the elements that were matched in the pointcut. To provide this functionality the DotAdvice class is created, which contains the type of the advice and the body, being the elements that have to be inserted or deleted. These elements are expressed using the Pointcuts; this allows us to quickly reuse all the functionality for the pointcuts, among which the usage of parameters and wildcards. An example of an advice is shown in listing 5.11.

```
advice fn: insert n() {
  main [shape=box];
}

advice dn: delete n() {
  main [color=..];
}

advice in: insert n(Node i) {
  i [shape=box];
}

advice ie: insert n() {
  (main) -> (exit);
}
```

Listing 5.11: Some examples of advice.

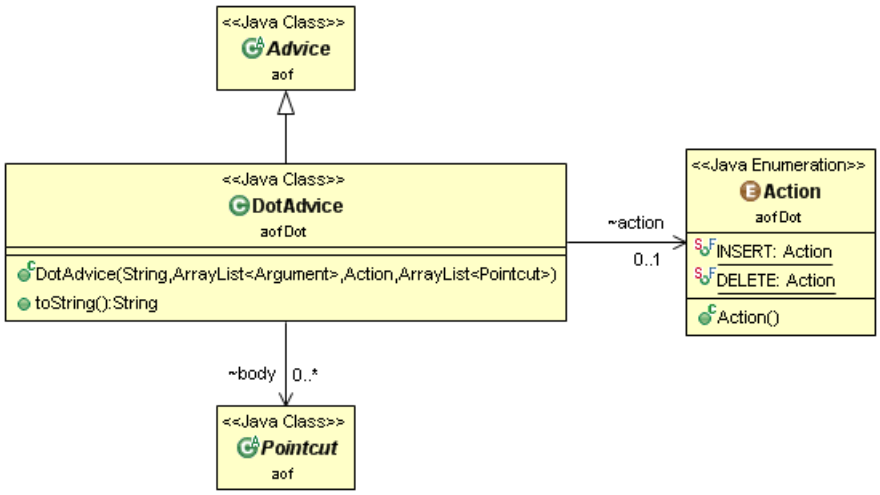

Figure 5.4: UML diagram of the DotAdvice class.

5.6 Order

Despite the absence of runtime behaviour it is still possible that conflicts arise if two advice modify the same element and the modifications overlap. Consider an advice that colors a node red, and another that colors it blue. By changing the order of these two advice we either get a red node or a blue one. To specify an order, we simply create a semicolon separated list of the order in which they have to be executed. If there are two advice without an order that do execute at the same time, the order is undetermined. The following example demonstrates the difference in output caused by the order.

```
1 digraph prof {
2   start [color="blue"];
3   main [color="green"];
4   exit [color="red"];
5
6   start -> main;
7   main -> exit;
8   start -> exit;
9 }
```

Listing 5.12: The source code.

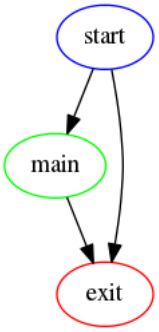

Figure 5.5: Graph corresponding to listing 5.12.

```
1 pointcut n(Node i) {
2   i = start [..];
3 }
4
5 advice dn: delete n(Node i) {
6   i [color=..];
7 }
8
9 advice fn: insert n(Node i) {
10   i [color="red"];
11 }
```

Listing 5.13: An example of the order.

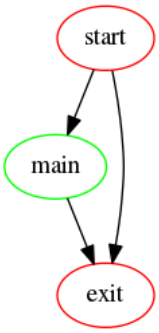

Figure 5.6: Result of executing dn before fn.

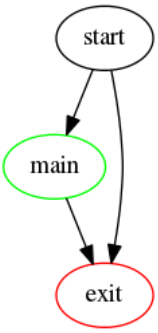

Figure 5.7: Result of executing fn before dn.

5.7 Weaver

The weaver will first gather all information, i.e. join points, pointcuts and advice. After that it will start weaving the code. It is important to note that the weaver will first find all matching pointcuts for all join points before weaving. This is important as it means that if one advice modifies an element in such a way that it no longer matches a join point, any change caused by advice linked to that pointcut will still be executed.

No other important notes are to be made. Thanks to the Dot compiler it is easy to get all the graphs, nodes and edges from a file, modify them and write the resulting graph back to file. The UML diagram of the DotWeaver is shown in figure 5.8.

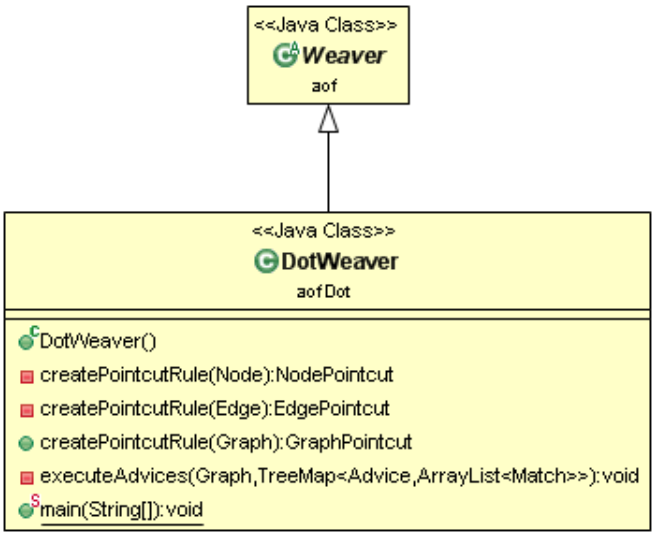

Figure 5.8: UML diagram of the DotWeaver class.

5.8 Example

Now that all the basic elements are explained, it is time to show a more complicated example. I will provide both the code and the visual graph before and after weaving.

```
digraph finite_state_machine {
  rankdir=LR;
  node [shape = circle];

  LR_0 [shape = doublecircle];
  LR_3 [shape = doublecircle];
  LR_4 [shape = doublecircle]
  LR_8 [shape = doublecircle];

  LR_0 -> LR_2 [label = "SS(B)"];
  LR_0 -> LR_1 [label = "SS(S)"];
  LR_1 -> LR_3 [label = "S($end)"];
  LR_2 -> LR_6 [label = "SS(b)"];
  LR_2 -> LR_5 [label = "SS(a)"];
  LR_2 -> LR_4 [label = "S(A)"];
  LR_5 -> LR_7 [label = "S(b)"];
  LR_5 -> LR_5 [label = "S(a)"];
  LR_6 -> LR_6 [label = "S(b)"];
  LR_6 -> LR_5 [label = "S(a)"];
  LR_7 -> LR_8 [label = "S(b)"];
  LR_7 -> LR_5 [label = "S(a)"];
  LR_8 -> LR_6 [label = "S(b)"];
  LR_8 -> LR_5 [label = "S(a)"];
}
```

Listing 5.14: The source code.

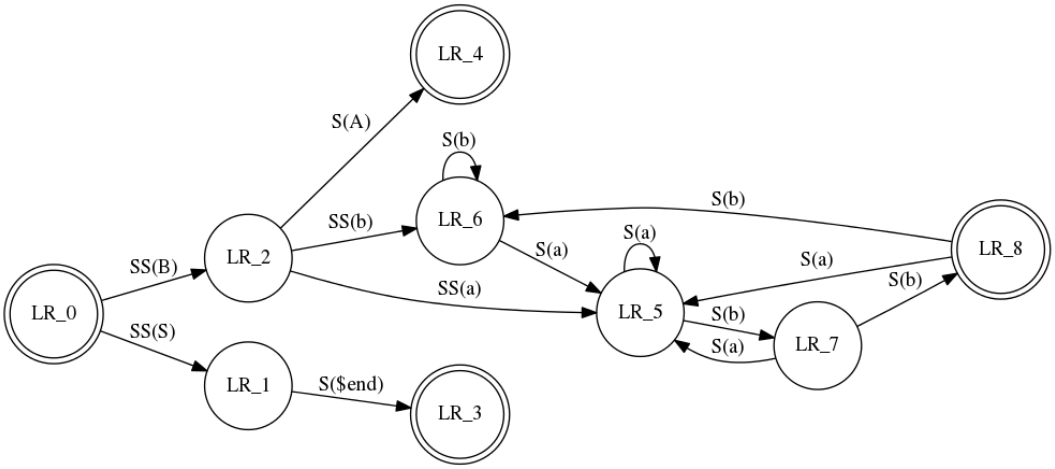

Figure 5.9: The graph before weaving.

```
order{fn; color}

pointcut n(Node j) {
  j = LR_.. [..];
}

pointcut m(Edge j) {
  j = (.. [..]) -> (.. [shape=
    doublecircle, ..]) [..];
}

pointcut end(Node n) {
  n = .. [shape=doublecircle];
}

advice fn: insert n(Node j) {
  j [color=blue];
}

advice mark: insert m(Edge j) {
  j [color=red];
}

advice color: delete end(Node n) {
  n [color=..];
}

advice shape: insert end(Node n) {
  n [shape=box];
}
```

Listing 5.15: Aspect code.

```
digraph finite_state_machine {
  rankdir=LR;
  node [shape=circle];
  LR_0 [shape=box];
  LR_3 [shape=box];
  LR_4 [shape=box];
  LR_8 [shape=box];
  LR_2 [color=blue];
  LR_1 [color=blue];
  LR_6 [color=blue];
  LR_5 [color=blue];
  LR_7 [color=blue];

  LR_0 -> LR_2 [label=SS(B)];
  LR_0 -> LR_1 [label=SS(S)];
  LR_1 -> LR_3 [color=red, label=S(
    $end)];
  LR_2 -> LR_6 [label=SS(b)];
  LR_2 -> LR_5 [label=SS(a)];
  LR_2 -> LR_4 [color=red, label=S(A)
    ];
  LR_5 -> LR_7 [label=S(b)];
  LR_5 -> LR_5 [label=S(a)];
  LR_6 -> LR_6 [label=S(b)];
  LR_6 -> LR_5 [label=S(a)];
  LR_7 -> LR_8 [color=red, label=S(b)
    ];
  LR_7 -> LR_5 [label=S(a)];
  LR_8 -> LR_6 [label=S(b)];
  LR_8 -> LR_5 [label=S(a)];
}
```

Listing 5.16: The woven code.

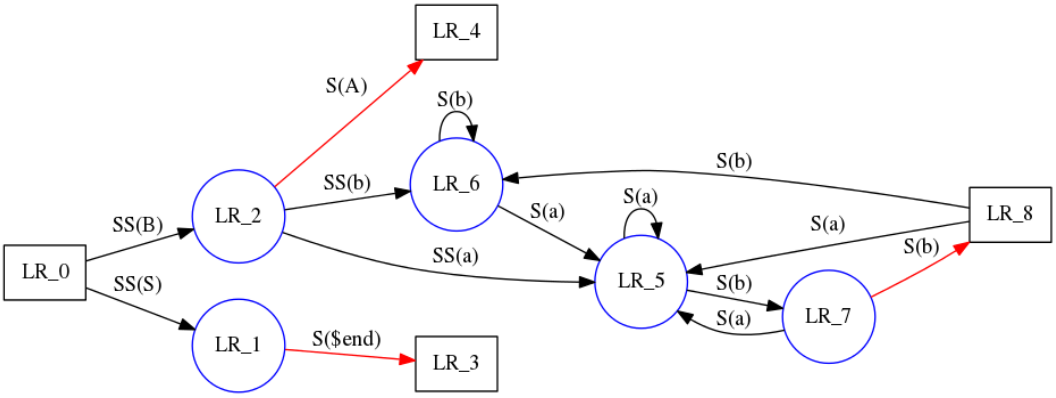

Figure 5.10: The graph after weaving.

5.9 Difficulties

The biggest problem encountered was how to decide on the syntax of the aspect-oriented constructs in our extension of Dot. Once this was done adding aspect-orientated programming went really fast, also thanks to the compiler which allowed it to easily modify the graph and write it back to file.

One small problem was that there were two minor bugs in the compiler which needed to be fixed first, but this could be done really fast due to the availability of the source code.

CHAPTER 6

AOWP

Due to the limited time available it is impossible to implement more languages using the framework. Instead I have chosen to examine existing languages and determine how they could be implemented with the framework.

AOWP [3] is a domain-specific language for web applications. Most web applications are developed with the Model-View-Controller framework which clearly separates the application into business logic (model layer) and the visual component (view layer), which communicate through the controller.

Although there exists an AOP language for PHP called AOPHP, this language has some flaws. First of all there is a lack of pointcuts to really cover all join points of web applications. Secondly there is no way to handle a history of page visits without modifying the web pages. Finally we note that the aspects can not be sufficiently separated from the crosscutting concerns.

6.1 Join Point Model

The join points that can occur in a web application consists of events based on HTTP and language specifications of well-known web programming languages such as PHP and Java/Servlet. Where the first consists of receiving HTTP requests, reading form data and reading or writing cookie data, the latter consists of reading the session data. An overview of all the events is shown in table 6.1.

6.2 Aspect Language

AOWP reuses the syntax of web programming languages such as PHP and Java/Servlet. It does this by mapping the aspect constructs onto object-oriented structures. By doing this

Abbreviation	Event
REQ	Receive an HTTP Request
F/R	Read the form data
C/R	Read the cookie data
C/W	Write the cookie data
S/R	Read the session data
S/W	Write the session data

Table 6.1: AOWP events.

the effort required by the user is minimized as he does not need to learn a new language.

Creating an aspect can be done by creating a class that extends the Aspect class, adding a pointcut and an advice is done by declaring functions with the name of respectively 'pointcut' and the name of the type of advice ('before', 'after', 'around'). An example is shown in listing 6.1 and 6.2.

This could also be accomplished if we were to use our framework, since it is required to create an aspect language of our own. This means that the same syntax could be chosen, though it has to be said that the framework has no notion of aspects, but this is not a problem as it is the responsibility of the compiler to extract the required information and provide it to the weaver in a format it understands.

6.3 Pointcuts

There are three categories of pointcuts, a first one selects events, a second selects event flows and a third selects based on usage contexts. Each of these categories has a set of sub-types, or so-called designators. An overview of all the designators are given in tables 6.2, 6.3 and 6.4. How these designators are used can be seen in table 6.5. From this table we also see that it is possible to use regular expressions as an argument of a designator.

Designator	Description
request	Select REQ with the specified URL
getget	Select specified F/R in a GET request
postget	Select specified F/R in a POST request
cookieget	Select specified C/R
cookieset	Select specified C/W
sessionget	Select specified S/R
sessionset	Select specified S/W

Table 6.2: Designators for selecting AOWP events.

I will now discuss how these pointcuts could be implemented with the earlier presented framework. Instead of creating a pointcut subclass for each category it is more interesting to only create one subclass being AOWPPointcut. I choose to take all the pointcuts together as there is no distinction in how these are represented. The AOWP designators shown in table 6.3 will not become part of the pointcut, but instead will be added to the context.

Designator	Description
withinrequest	Select all AOWP events in a REQ
pflow	Select all AOWP events in specified page transitions

Table 6.3: Designators for selecting AOWP event flows.

Designator	Description
accessnum	Select all AOWP events when the number of access users is the specified number or greater.

Table 6.4: Designators based on usage contexts.

Example	Description
request(/news.php)	Selects all events of receiving HTTP requests on news.php.
withinrequest(/login.php)	Selects all events generedted during the execution of HTTP requests for login.php
pflow(/news.php)	Selects all the events involved in handling HTTP requests when the user has seen news.php
accessnum(5)	Selects all events involved in handling HTTP Requests when the number of users if 5 or more.
request(/*Action.php)	Selects the REQ whose name starts with '/' and ends with 'Action.php'.
!pflow(/news.php)	Selects a negative set of events selected by pflow(/news.php)

Table 6.5: Examples of designator.

The class will rely on the arguments of the Pointcut superclass to implement the argument. The specific designator this instance represents has to be indicated by a special value, either an enum or an integer.

By doing this we can easily specify pointcuts, creating join points is however a bit more tricky due to capturing the event flow. We could do this naively and create a pointcut for each element we encountered along the flow, but that would cause a long creation and comparison delay. It is much better to capture the entire flow as a context and provide that along with the join point. This means that checking whether a pointcut matches a join point is a bit more work, but it allows us to generate only one instance for the entire join point.

AOWP allows combining multiple designators by using the negation (!), intersection (&) and union (|) operator. This is not possible with the framework, since this only supports union. It is however not a big problems as we can create a new type of pointcut, the OperatorPointcut. This pointcut will simply execute a certain operator (!, & or |) on its members, which are again pointcuts. Another approach could have been to use the union of the framework and using the OperatorPointcut only for the negation and intersection, but this would lead to some extra work to be done. We would have to rework some expressions

to make the unions the top-level operator or remove them completely. This can be seen in the following examples. In these examples I use simple letters to represent pointcuts.

$$!(a|b)\,\&c \Rightarrow (!a\&!b)\,\&c$$
$$(a|b)\,\&!c \Rightarrow (a\&!c)\,|\,(b\&!c)$$

To avoid this extra work it makes more sense to treat the union the same way as the negation and the intersection.

6.4 Advice

The types of advice AOWP supports is the same as AspectJ, being 'before', 'after' and 'around'. The body of an advice is normal code with extension of the **proceed()** call. In contrast to AspectJ, there are no variables for the context, all this information is provided as arguments of the advice.

6.5 Weaver

Due to the dynamic behaviour of web applications it is impossible to rely on compile-time weaving, and thus run-time weaving is required. The weaver itself works pretty straightforward by extracting join point shadows and injects code to call the weaver.

This would cause some problems when using the framework, as the current version does not support run-time weaving. A work-around is to set up the weaver each time it is required, but this introduces a large overhead for creating and loading the pointcuts each time.

6.6 Examples

```
1  class NewsAlertAspect extends Aspect {
2    /* !pflow(/news.php) & request(*) */
3    funcion Pointcut() {
4      $pc = new Pointcut();
5      $pc->addNotAnd(new PFlowPC("/news.php"));
6      $pc->addAnd(new RequestPC("*"));
7      return $pc;
8    }
9
10   function before($adviceContext) {
11     alertToViewNews();
12   }
13 }
```

Listing 6.1: An advice to alert the user to check the newspage.

```
1  class LoadAlertAspect extends Aspect {
2    /* accessnum(20) & request(*) */
3    funcion Pointcut() {
4      $pc = new Pointcut();
5      $pc->addAnd(new AccessNumPC(20));
6      $pc->addAnd(new RequestPC("*"));
7      return $pc;
8    }
9
10   function before($adviceContext) {
11     alertForAccessTooMany();
12   }
13 }
```

Listing 6.2: An advice that alerts on heavy load.

6.7 Difficulties

The language itself will not give rise to many problems if we would try to implement it with the framework. Only the requirement for run-time weaving is currently a problem, though there are some work-arounds these are not feasible and would require more work than if the framework would make it possible. I discuss this later again in chapter 10.

CHAPTER 7

AspectMatlab

In this chapter I examine how we could implement AspectMatlab by using the framework. AspectMatlab [1], as the name suggests, is an AOP extension to Matlab. Like other extensions it adds pointcuts and advice for method calls or execution and getting or setting of variables. It does however more than just this, because Matlab has a strong focus on loops, it also adds pointcuts for loops.

As presented in the paper on AspectMatlab, it performs source-to-source compilation taking Matlab and AspectMatlab source files as input and generating Matlab source files as output. The compiler was built using a couple of toolkits such as Natlab and MetaLexer to get a modular and extensible compiler.

7.1 Join Point Model

Besides the method calls and execution that are present in most languages, call and execution can also be applied for scripts which are commonly used in Matlab. Other commonly found join points such as get and set for arguments are also present. Aside from these, AspectMatlab also identifies three join points concerning loops, being the execution of the entire loop, the execution of the loop head and the execution of the loop body. This is because the loop is one of the main entities in a typical Matlab program.

7.2 Base Language Compiler

The base language compiler used in the paper is McLab, which is an extensible Matlab compiler. This comes in handy as we will want to modify it to report information and join points to the weaver.

7.3 Aspect Language

The main element of the language is an aspect. Such an aspect can contain properties and methods like a class, but more important it has patterns (pointcuts) and actions (advice). Each of these elements has a syntax similar to normal Matlab syntax as can be seen in listing 7.3.

The main problem that arises if we want to implement this with the framework, is that there is no notion of 'aspect', which makes it impossible to have properties or methods related to it. If we would want to implement something similar as in the mentioned example, we would have to implement a class or script containing the methods and attributes to which we could write the code for the actions during weaving. This is however not very clean code, and we would be undermining the main goal of AOP, which is to clearly concentrate and separate the concerns.

Other cases however where we do not rely on attributes can be done without a problem. Also the presence of methods in the aspect files is not a problem as we can simply copy them to a new script or even directly into the advice that needs it.

AspectMatlab provides the user with some context information in the form of context selectors. Which are present depends on which join point is encountered. Doing this with the framework can be done by specifying them as arguments. This is easier than using the context information as the Argument already has a name and a certain type. An overview of all the context selectors for different join point types are shown in table 7.1.

	get	set	execution	call	loop	loopbody	loophead
args	indices		arguments passed		-	-	-
obj	variable	variable before set	-	function handle	loop iterator variable		
newVal	-	new array	-	-	-	-	range expression
counter	-	-	-	-	-	current iteration	-
line	line number in the source code						
loc	enclosing function/script name						
name	name of the entity matched				-	-	-
varargout	cell array variable used to return data from `around` action						

Table 7.1: Context selectors for different join point types.[1]

7.4 Pointcuts

The pointcuts match the join points which were discussed earlier. Therefore I will not discuss them any further but just show an overview in table 7.2.

It is possible to further restrict the scope of matching by using the within pattern. On the other hand it is also possible to generalize pointcuts by using wildcards. The '*' wildcard can be used to match any function/variable or exactly one argument. The '..' wildcard can be used to match one or more variables.

Pointcut	Description
call	Captures all calls to functions/scripts.
execution	Captures the execution of function bodies.
get	Captures array accesses.
set	Captures array sets.
loop	Captures execution of all loops.
loophead	Captures the header of the loop.
loopbody	Captures the body of the loop.

Table 7.2: List of patterns.

Because of the Matlab syntax, where 'foo(1,2)' could either be a call to a function with arguments '1' and '2' or an array indexation, it is possible to create a general notation for calling a function or script and getting or setting a variable. A list of possible pointcuts is given in table 7.3.

Pointcut	Description
call(foo)	Matches all calls to foo.
call(foo())	Matches calls to foo with no arguments.
call(foo(*))	Matches calls to foo with exactly one argument.
call(foo(..))	Matches calls to foo with 1 or more arguments.
call(foo(*,..))	Matches calls to foo with 2 or more arguments.
call(*(*,..))	Matches any call with two ore more arguments.

Table 7.3: Selective pattern matching.

If we were to implement these pointcuts with the new framework we might end up with three subclasses for Pointcut. It is however possible to group them all together into one since there is no clear distinction between a function call and getting a variable. The within pattern would become part of the context provided to the pointcut.

7.5 Advice

AspectMatlab identifies three types of advice being before, around and after. The body of an advice is normal AspectMatlab code, possibly extended with the proceed() call, as can be seen in listing 7.3.

As mentioned before, is it possible to provide context information to the advice. It is also possible to use variables specified within the aspect to communicate between advice or between multiple executions of the same advice. This latter one will cause the most problems as it is currently impossible to do anything like this with the framework. To solve this a work-around could be used where we use data in the original matlab code. This is however bad coding and breaks the entire concept of aspect-oriented programming.

7.6 Order

AspectMatlab introduces a default ordering where the first encountered around advice becomes the outermost one, followed by the rest of the around advice in the order of appearance. After the around advice, all before advice are placed in order of occurrence before the actual join point. The same holds for the after advice that are placed right after the join point in the order encountered before the around advice. This relation between before, after and around advice is shown in figure 7.1.

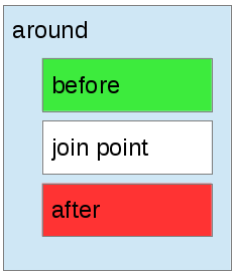

Figure 7.1: A representation of the order between before, after and around advice.

Since the framework does not provide such a default order, it is required for the AspectMatlab compiler to explicity add such an order to the weaver. This isn't very complicated though, it suffices to remember the last encountered advice of each type (before, around and after) so that when we read a new one, we can create a link between these two advice and mark the stored one as a predecessor of the new one.

This solves the ordering problem between advice of the same type, what still remains is the order between the different types. There are two different approaches to solve this:

1. Create a link between the types.
2. Handle the different types one by one in the weaver.

The first approach may sound like the best one, but at the same time it does some redundant work, simply because when weaving we will have to treat every type of advice differently. The different treatment is a direct result of the inherent difference between before, after and around advice. This means we could organize them during weaving according to their type.

If however it is still preferred to have an order between the different types, which is possible as it might be helpful with certain implementations, this isn't much work to be done. To execute all before advice after the last around advice, we need to store the first encountered before advice. We need it at the end to define the last around advice as a predecessor of this first before advice. To make the last after advice happen before the last around advice, since this will be the deepest one and thus the first to be executed after the after advice, we need to define the last after advice as a predecessor of the last around advice.

Because of the order behaviour caused by the around advice, which makes the order 'reflecting' over the join point. This leads to some confusing ordering and thus it can still not be used that straightforward.

7.7 Weaver

The AspectMatlab compiler (weaver) has been built using extensible toolkits, and aimed for a very clean and modular implementation. It takes a collection of Matlab and AspectMatlab source files as its input and produces a collection of woven Matlab source files. I will not discuss the entire process of how this is done, but an overview is given in figure 7.2.

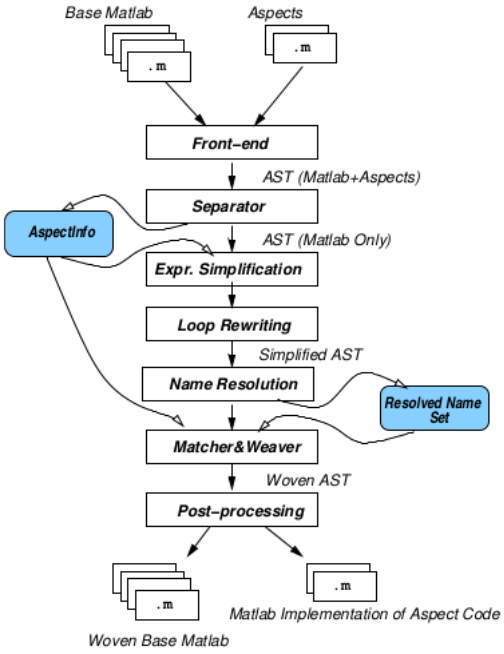

Figure 7.2: Overal structure of the amc AspectMatlab compiler. [1]

To be able to call all the advice at the right point, some transformations are required. All compound expression are disassembled into simple expressions as can be seen in the following example:

```
1 z = sum(x) / length(y);
```

Listing 7.1: Original code.

```
1 AM_CVar_5 = x;
2 AM_CVar_6 = sum(AM_CVar_5);
3 AM_CVar_7 = y;
4 AM_CVar_8 = length(AM_CVar_7);
5 z = (AM_CVar_6 / AM_C_Var_8);
```

Listing 7.2: Resulting code.

It is also important to note that, due to Matlab's semantics it is hard to determine at compile time the semantics of an expression. Because they wanted to minimize the amount of run-time checks, they introduced a name resolution analysis which adds extra information to the AST so they can determine the type.

We could implement this the same way using the framework, without requiring any changes to the process.

7.8 Examples

```
aspect myAspect

  properties
    count = 0;
  end

  methods
    function out = getCount(this)
      out = this.count;
    end

    function incCount(this)
      this.count = this.count + 1;
    end
  end

  patterns
    call2args : call(*(*, ..));
    executionMain : execution(histo);
  end

  actions
    actcall : around call2args : (name, args)
      this.incCount();
      disp(['calling', name, 'with parameters(', args , ')']);
      proceed();
    end

    actexecution : after executionMain
      count = this.getCount();
      disp(['total calls: ', num2str(count)]);
    end
  end

end
```

Listing 7.3: An aspect to count all calls made with at least 2 arguments.

```
aspect unit
  patterns
    loopheader : loophead(*);
  end;

  actions
    loop : around loopheader : (newVal)
      range = this.annotate(newVal);
      acell = {};
      for i = (range.val)
        acell {length(acell) + 1} = i;
      end
      varagout{1} = struct(this.annotated, true, 'val', acell, 'unit', range.
    unit);
    end
  end
end
```

Listing 7.4: Example of a units aspect.

7.9 Difficulties

The current version of the framework does not cover all features to realize the complete version of AspectMatlab as it is presented. This is a limitation that can easily be fixed by adding those features, as discussed in chapter 10. The features that are present in the framework are easy to use for AspectMatlab as the idea behind the elements is similar as those used in Small C.

CHAPTER 8

DiSL

In this chapter I examine how we could implement DiSL by using the framework. DiSL [6] is a domain-specific language for bytecode instrumentation. Designing such a language is a complex task as it needs to achieve three conflicting design goals:

1. High expressiveness.
2. High-level programming model.
3. High efficiency.

Currently there exist low-level languages that meet the first and third goal, but these are based on bytecode instrumentation, which makes it hard to develop, maintain and customize tools. Other AOP languages achieve the second goal but do not succeed in exposing important join points, easily providing access to information and mixing bytecode with aspect-oriented code.

8.1 Join Point Model

To achieve a high level of expressiveness DiSL has an open join point model, which means that any region of bytecode can be a join point, provided with any kind of static and dynamic information.

8.2 Base Language Compiler

DiSL uses Java as its host language and uses annotations to express the functionality required. To parse the base language source code DiSL relies on jBORAT, a lightweight toolkit providing support for instrumentation with complete bytecode coverage.

8.3 Aspect Language

First of all it has to be noted that DiSL's terminology differs from what is generally used in that they respectively use *instrumentation*, *marker* and *snippet* instead of aspect, pointcut and advice. Guards and scopes can be used to restrict the matching of snippets. The difference is that guards are implemented as a class with methods that will be evaluated and should return a boolean value, while scopes limit the execution based on the method signature of the join point.

As mentioned before the code is written as plain Java classes with annotated methods and fields. Besides having an open join point model, DiSL also allows the user to define his own static and dynamic information which can be used in the snippets. It is also possible to add argument processors which will be used to process arguments. Two examples of DiSL are shown in listing 8.4 and 8.5.

8.4 Pointcuts

To be able to handle the open join point model, DiSL provides an extensible library of markers/pointcuts. These markers are standard Java classes implementing a special interface for join point selection. It has to be noted that in contrast to many other AOP languages these markers are not part of the instrumentation, but are completely implemented on their own.

There are two possible ways to implement this with the framework. The first one is doing exactly the same as is done now, but instead of an interface the markers would have to extend the marker pointcut. A problem with this approach is to keep the weaver informed of all these new markers, but that's a problem that can be overcome. Another way would be to create a very general marker pointcut that is able to fit every specified marker such that the parser can create a pointcut with the specified data. This would however cause a lot of problems as the pointcut would be too general, making it hard to really suit the needs of the user. It is hard to tell what the best solution would be, as the paper does not provide a lot of information about how to specify own markers.

8.5 Advice

Advice is expressed in the form of code snippets that are inlined, giving the developer fine-grained control over the inserted code. The snippets are written as regular Java static methods and annotations are used to indicate the type, either 'before' or 'after', and other properties such as the marker, guard and scope.

Mainstream AOP languages support 'before', 'after' and 'around' advice, but since DiSL is meant for bytecode analysis and thus not meant to modify any code, there is no use for the 'around' advice. For that reason DiSL only provides a 'before' and 'after' advice. An example of a snippet is shown in listing 8.1.

```
1 @Before(marker = BodyMarker.class)
2 static void advice() {
3   System.out.println("Advice being executed.");
4 }
```

Listing 8.1: Example of a snippet.

Because snippets are inlined, they are able to efficiently communicate data through local variables defined in the instrumentation. It is also possible to communicate at a thread level.

A snippet can use both static and dynamic information, which is passed on as an argument, which are the only possible arguments. Once again it is possible for the user to specify their own information by creating a class that implements a certain interface. An example of a snippet using context information is shown in listing 8.2.

```
1 @Before(marker = BodyMarker.class)
2 static void advice(MethodStaticContext msc) {
3   System.out.println("Advice being executed for " + msc.thisMethodFullName());
4 }
```

Listing 8.2: Example of a snippet using context information.

An advice is pretty simple to implement as it is similar to what we did before. With the implementation of the static and dynamic information we face the same problems as with implementing the pointcuts. The framework does not provide a separate mechanism for guards or scope and thus it is required that this information is added to the context of the pointcut.

8.6 Order

Order can be specified for each snippet as a non-negative integer number. The rule is that snippets with a higher order are closer the the join point than snippets with a lower order. An example of this order can be seen in listing 8.3.

```
1 @Before(marker = BodyMarker.class, order = 0)
2 static void first() {
3   System.out.println("This advice is executed first.");
4 }
5
6 @Before(marker = BodyMarker.class, order = 1)
7 static void second() {
8   System.out.println("This advice is executed second.");
9 }
```

Listing 8.3: Example of order between two snippets.

The framework does not work with an order value, but instead requires a connection between advice. This means that the compiler needs to convert these values into an explicit link between advice with consecutive order values.

The implicit order of appearance between snippets with the same order value has to be made explicit to the weaver. The parser can do this by making an order while parsing where he states that the previous encountered one must occur before the next encountered.

8.7 Weaver

The DiSL weaver runs on top of jBORAT, which will help us by parsing the classes from the source code. The process of weaving with DiSL goes as follows:

1. First DiSL parses all the instrumentation classes.
2. When jBorat hands over a class to DiSL an internal representation for snippets, markers, guards, static contexts and argument processors is created.
3. Scopes are matched.
4. Join point shadows are created and evaluated by guards, and matching snippets are selected.
5. Static contexts are used to compute static information.
6. Argument processors are evaluated for snippets and matching snippets are selected.
7. Everything is woven together.
8. Static information and bytecode to access dynamic information is added.

This is shown visually in figure 8.1.

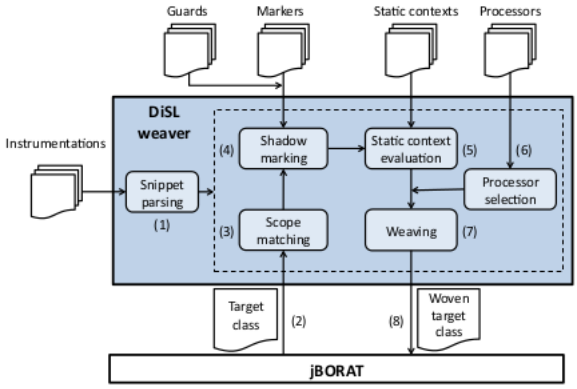

Figure 8.1: Overview of DiSL weaving process.[6]

8.8 Examples

```
1 public class CallingContextBBAnalysis {
2   @ThreadLocal
3   static CCTNode currentNode;
4
5   @SyntheticLocal
6   static CCTNode callerNode;
```

```
7
8   @Before(marker = BodyMarker.class, order = 1)
9   static void onMethodEntry(MethodStaticContext msc) {
10    if ((callerNode = currentNode) == null) {
11      callerNode = CCTNode.getRoot();
12    }
13    currentNode = callerNode.profileCall(msc.thisMethodFullName());
14  }
15
16  @After(marker = BodyMarker.class)
17  static void onMethodcompletion() {
18    currentNode = callerNode;
19  }
20
21  @Before(marker = BasicBlockMarker.class, order = 0)
22  static void onBasicBlock(BasicBlockStaticContext bbsc) {
23    currentNode.profileBB(bbsc.getBBIndex());
24  }
25 }
```

Listing 8.4: Aspect for calling context-aware profiling.

```
1  public class Senseo {
2    @ThreadLocal
3    static CCTNode currentNode;
4
5    @SyntheticLocal
6    static CCTNode callerNode;
7
8    @Before(marker = BodyMarker.class, order = 1)
9    static void onMethodEntry(MethodStaticContext msc, ArgumentProcessorContext
       proc) {
10     if ((callerNode = currentNode) == null) {
11       callerNode = CCTNode.getRoot();4
12     }
13
14     currentNode = callerNode.profileCall(msc.thisMethodFullName());
15     proc.apply(ReferenceProcessor.class, ProcessorMode.METHOD_ARGS);
16   }
17
18   @After(marker = BodyMarker.class, order = 2)
19   static void onMethodCompletion() {
20     currentNode = callerNode;
21   }
22
23   @AfterReturning(marker = BodyMaker.class, order = 1, guard =
       MethodReturnsRef.class)
24   static void onReturnRef(DynamicContext dc) {
25     Object obj = dc.getStackValue(0, Object.class);
26     currentNode.profileReturn(obj);
27   }
28
29   @AfterReturning(marker = ByteCodeMarker.class, order = 0, args = "new,
       newarray,anewarray,multianewarray")
30   static void onAllocation() {
31     currentNode.profileAllocation();
32   }
33
34   @Before(marker = BasicBlockMarker.class, order = 0) {
35   static void onBasicBlock(BasicBlockStaticContext bbsc) {
36     currentNode.profileBB(bbsc.getBBIndex());
37   }
38 }
39
```

```
40 @ArgumentProcessor
41 public class RefferenceProcessor {
42   static void objProc(Object obj, ArgumentContext ac) {
43     Senseo.currentNode.profileArgument(ac.getPosition(), obj);
44   }
45 }
```

Listing 8.5: Example of collecting run-time information.

8.9 Difficulties

DiSL is a language completely different from those discussed until now. Despite this difference the framework is still able to provide useful functionality proving its value. Besides some minor difficulties, mostly caused by the lack of functionality, there are no big problems if this were to be created with the framework. All the tools that are used to aid in creating this language could also be used when working with the framework.

CHAPTER 9

Discussion

As demonstrated by the languages presented and discussed in the previous chapters, the framework is general enough to be useful in a wide range of languages. The amount of work required to instantiate the framework depends on the complexity of the target language, and is mostly dominated by the effort required for the weaver. Thanks to the presence of 'helper' methods in the general Weaver class, the work that has to be put into the weaver is limited to putting everything together, and finding matches is already done for us.

In general we experience that the concept of pointcuts and advice, as it is represented by the framework, provides a good basis to start from. The extensions that are required are very limited. The entire type system however, because it can not be provided by the framework may cause to be problematic. Though if handled well, it is possible to simply reuse the existing type system of the base language eliminating the cost of creating a complete new one.

The poincutset which was meant as a way to simplify selecting multiple join points that could not be selected by wildcards proves to be very useful, though it still has room for improvement since the only supported operator is the logical or. As most other AOP languages support other operators as well, this may seem to be a very limited way to express a set of pointcuts. Upon closer inspection this is often not the case, since for most join points it makes no sense to require that it should happen together with another join point. If we look at the some examples of AspectJ, we note that the '&' is used with the **within** keyword. This however is not a join point but is part of the context in the framework.

The instantiated framework and the base language only interact with a compiler that extracts the join points from the source code, and with the weaver that adds the advice in the right place. This is however not the only coupling between the base language and the framework. Most components that are added to support AOP for the base language will carry some notion of the base language, being it implicitly. More coupling between the framework and the base language is the type system which is preferably the same.

Finally we note that modifying the parser for the base language to extract the join points can be quite cumbersome. One way to minimize the required effort would be to use an ex-

isting aspect-oriented language and inject the modifications as advice. This way of working is however very limited and can only be done if there is a match between the join point model of the aspect-oriented language and the interesting points in the parser. If we would for instance use AspectJ, we are limited to the beginning and end of method executions and calls. If the parser is not sufficiently organized it is impossible to add the code at the right place and the entire approach can not be applied without modifications to the parser.

CHAPTER 10

Future Work

The languages discussed in this thesis have shown some limitations of the framework. The most important one is the lack of communication between advice. This could be solved by introducing global variables, which would be similar to arguments currently present for advice. The weaving step would have to be extended depending on the way the variables are woven with the base language source code.

Many other AOP languages have an explicit notion of aspect, this is not the case in this framework but could be considered as a valuable addition. Before doing so, it is essential that some questions are answered. Will the aspect only be used to encapsulate variables, pointcuts and advice or will it be possible to use it like classes in an object-oriented language? If that is the case then we gain some valuable features, such as the reusability of pointcuts. However, inheritance would also apply for advice, which means we need a way to select the right advice in case they are overridden. Another problem would be to choose between multiple subaspects, certainly since this choice may not be the same for the entire application, but can differ from join point to join point.

Another problem with the current framework is the lack of choice between different ways of weaving. Though it is possible to weave the original files and the advice any way desired, it is the time of weaving that lacks freedom. The only real option is using the weaver as a pre-processor. This should be extended to provide more support for run-time weaving, since adding run-time checks isn't always feasible.

Another important feature still missing is decent support for debugging, as this is completely left to the compiler of the target language. This however means that all errors are reported based on the woven code instead of the original source and aspect files. The easiest way to do this is to encapsulate the final compiler and intercept the errors. By keeping track of the woven locations we can resolve the errors back to the original locations. This approach can not only be done for 'fixing' the reported errors, but could also be used to handle other tools that work on the woven code.

Currently the framework treats context and arguments in a different way, and the notion

of a context is closer to a type, as they both define an interface with just an **isAssignable** method. We could ask ourselves in what way a context would be different from an argument. Since in most cases a context variable will have a name and a value, just as an argument.

Finally also the weaving process could be optimized to handle quick re-weaving. Currently the weaver starts from the original files every time the process is started. This requires a lot of time for large projects and a lot of woven code will be the same as the previous, since often only small changes are made at a time. Achieving this involves adding links between the aspect code and source code, which is also required to 'fix' the error messages mentioned earlier.

CHAPTER 11

Related Work

Language-oriented programming [2] is similar in that way that it provides freedom to the user to create his own domain-specific language, just like the framework allows a user to specify his own aspect-oriented language. This is realized by implementing loosely coupled languages which can be combined as requested.

Pluggable AOP [5] aims to specify aspect-oriented languages largely independent of the base language, which would allow combining multiple aspect-oriented languages together. Though the approach is different as they don't use a framework but a description of the aspect-oriented language, both aim to get a loose coupling between the aspect-oriented language and the base language.

Higher order attribute grammars [10] are a new kind of grammar that enables the structure tree to be extended as a result of attribute computations. This allows languages that can be more easily modified and enable extensions. This is shown in Aspects as Modular Language Extensions [9], where they use a programming language that uses a higher-order attribute grammar to extend it with AOP. Where this thesis is about how we can add AOP to any type of language by having a general and modifiable framework, this paper is about how we can add AOP to modifiable languages. We might be able to use (parts of) this technique in the framework.

XAspects [8] is a framework for aspect compilation that allows to combine multiple domain-specific aspect extensions. It does this by reducing all extensions to a single general-purpose aspect extension (AspectJ). This is an alternative to the type of framework presented in this thesis, but is not as general, since not all aspects can be reduced into AspectJ.

In Towards An Aspect-Oriented Language Module: Aspects For Petri Nets [7], the concept of AOP is added to Petri Nets. Since this is clearly not a standard textual programming language such as Java, the entire idea is examined. It is shown that AOP is meaningful outside the current scope of its use, which supports the design of the framework in this thesis.

CHAPTER 12

Conclusion

The provided functionality of the framework is still somewhat limited. Nevertheless it is able to achieve an abstraction of the core elements of aspect-oriented programming and has been successfully instantiated to prove its abilities and limitations. Despite the limitations of the current framework, it is often already possible to solve the problem in a different way. Yet adding new features and extensions will allow serving an even larger amount of languages and make the framework more user-friendly.

Bibliography

[1] Toheed Aslam, Jesse Doherty, Anton Dubrau, and Laurie Hendren. AspectMatlab: an aspect-oriented scientific programming language. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, page 181192, New York, NY, USA, 2010. ACM.

[2] Sergey Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains*, November 2004.

[3] K. Hokamura, N. Ubayashi, S. Nakajima, and A. Iwai. Aspect-oriented programming for web controller layer. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 529–536, December 2008.

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

[5] Sergei Kojarski and David H. Lorenz. Pluggable AOP: designing aspect mechanisms for third-party composition. page 247. ACM Press, 2005.

[6] Luk Marek, Alex Villazn, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. DiSL: a domain-specific language for bytecode instrumentation. page 239. ACM Press, 2012.

[7] Tim Molderez, Bart Meyers, Dirk Janssens, and Hans Vangheluwe. Towards an aspect-oriented language module: Aspects for petri nets. In *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages*, DSAL '12, page 2126, New York, NY, USA, 2012. ACM.

[8] Macneil Shonle. XAspects: an extensible system for domain specific aspect languages. In *In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, page 2837, 2003.

[9] Eric Van Wyk. Aspects as modular language extensions. *Electronic Notes in Theoretical Computer Science*, 82(3):555–574, December 2003.

[10] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, page 131145, New York, NY, USA, 1989. ACM.