Dynamic Aspects

An AOP Implementation for Squeak

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Anselm Strauss

Bern 2008

Leiter der Arbeit

Dr. Marcus Denker

Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik

Further information about this work, the tools used and an online version of this document can be found at the following places.

Anselm Strauss
strauss@iam.unibe.ch
http://www.squeaksource.com/DynamicAspects.html

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://scg.unibe.ch/

Abstract

Cross-cutting concerns in OOP lead to scattered and tangled code that reduces transparency and maintainability of a program, and produces duplicated code. Common examples of cross-cutting concerns are logging, caching or database transactions in an application. Rather than trying to break down such concerns into classes and objects, the entity of modularization in aspect-oriented programming (AOP) are aspects [Kicz97]. An aspect in AOP is the pendant of an object in OOP. With aspects such problems can be solved much easier and can be isolated into single entities. AOP can be seen as programming paradigm that builds on top of an existing paradigm and language. Today most AOP is done on top of OOP.

DYNAMIC ASPECTS is a lightweight AOP implementation for Squeak Smalltalk that profits from the advanced reflection tools developed by the Software Composition Group. Sub-method level reflection allows to select single statements within methods. Annotation of such statements allows extrinsic addition of behavior at any location in code. Finally, the idea of partial behavioral reflection gives the system great flexibility. All those tools are the core DYNAMIC ASPECTS builds upon. Beyond the basic AOP implementation, DYNAMIC ASPECTS links to the field of context-oriented programming and shows how contexts can be used in aspects. Furthermore, the idea of control flow is generalized and generic flow is implemented with contexts.

Acknowledgements

Many thanks to my supervisor Marcus Denker who supported me in many ways, from writing and maintaining the REFLECTIVITY software I'm using, through having the right key ideas for the thesis and putting me to the right direction, to taking the time to discuss with me and reading over my work. Also I thank Philippe Marschall for PERSEPHONE and David Röthlisberger for GEPPETTO. Both are great pieces of software that I'm using with REFLECTIVITY.

Thanks to Prof. Dr. Oscar Nierstrasz, the head of the Software Composition Group (SCG), for giving me the chance to do my master thesis here.

Additional thanks go to all the people hanging around with me in the SCG, that contributed to my work, whether with smart, silly or sceptical comments, whether it was intentional or unintentional.

Contents

Chapter 1

Introduction

This chapter will give a short idea of what aspect-oriented programming (AOP) is about and presents a short practical example. Further it will state the goals and the value of this thesis, and give an outline of the rest of this document.

1.1 AOP in a nutshell

Aspect-oriented programming is a way of programming, most often mentioned together with others like object-oriented programming or functional programming. In terms of programming paradigms AOP is more a pseudo-paradigm than a stand-alone paradigm. AOP implementations usually build on top of OOP, and also the motivation for AOP comes from shortcomings in OOP.

Let us think of a feature we want to add to our code, that will force us to add code to multiple methods in a class. For example, we want to log all write access to private instance variables, so we place a log statement in the beginning or at the end of all our setter methods. Now, we could do that by pasting this log statement everywhere in these methods, or we could use the *"aspect way"* described in the following. The latter has the advantage that we write the statement only once, and that we can easily share the state between all the locations in our methods.

AOP is not so much different from OOP as one might think. Essentially we want to add behavior (functionality), but we want to do it in a different way than we do with classes and objects. Basically we go through the following three steps:

Where to add behavior? Rather than choosing a specific class and method to add the code, we specify this in a reflective manner. Here we would select all the set methods in a specific class. Usually, we have some kind of high-level language called *pointcut language*. When evaluated, it maps to points in the execution of a program incorporating the current context of the execution.

What behavior to add? This is normal code as we have it in methods, only that it applies to various locations in our program, the ones we selected with the pointcut. This is called an *advice*.

What arguments are needed? Advices are code specified outside classes, but we still may need access to the internals of some objects. In addition we also want to know from where exactly the advice was called. Therefore we need the possibility to specify parameters an advice is called with, e.g. the object an advice is executing for can be a parameter.

These three things together are encapsulated in an *aspect*. Inside an aspect we also have some mechanism to connect pointcuts with advices. Consequently, aspects are also subject to code reuse, so that we want to have something like inheritance between them.

Aspects are not defined stand alone. They are programed on top of some original code. On the low level, aspects eventually result in modifications in the executing code. How and when these modifications are made is completely up to the specific implementation.

1.2 A first example

Take the example from the previous section, annotating all setter methods in a class. The selection in Smalltalk would look about the following:

Listing 1.1: Selecting setter methods

```
MyClass methods select: [ :m | 'set*' match: m selector. ].
```

In an AOP implementation we will most likely have a more sophisticated interface for doing such queries:

Listing 1.2: The pointcut pendant of selecting setter methods

```
PC class: MyClass method: 'set*'.
```

That is it for the pointcut part. It is a description of where to add behavior, and when evaluated will yield to the appropriate join points. PC is

just a global variable pointing to an object that provides various pointcut instantiation methods.

The advice would just be like a normal method body, e.g.:

Listing 1.3: Added behavior for setters

```
self log: 'Value set'.
```

How we connect pointcuts and advices and pack them into aspects is subject to the specific implementation. In the case of DYNAMIC ASPECTS pointcuts and advices are both methods, the class they are belonging to is the aspect. Pointcut methods can use the global called PC to construct the descriptions and must return a pointcut. Compositions of pointcuts behave like regular pointcuts. Pointcut and advice methods are special methods, that are marked with *pragmas*. The pragmas have a special syntax and can be used to provide additional information about source code. Here they are used to mark methods as pointcuts or advices, and in the latter case they are also used to specify additional parameters of an advice.

If we reconsider the last example, we may come to the conclusion that the log's expressiveness is rather limited. We would like to know from each field write what field it is setting, and it would also be nice to have some sort of accounting info for each field for later evaluation. The first enhancement we can reach by using advice arguments and a more detailed pointcut. The second one is possible if we see the advice in context of the class it is a method of, and know that each object of a class has it's state.

First we enhance the pointcut, and at the same time we make it a method of our new aspect LogSetters:

Listing 1.4: Pointcut method for setters

```
LogSetters class >> setters

    <pointcut>

    ↑ (PC class: MyClass method: 'set*') & PC instanceFieldWrite.
```

Note that this is on the class side of our aspect. Pointcuts are the same for all instances of an aspect. instanceFieldWrite will search inside the method body for specific statements that are writing to a field, meaning variable assignments.

For the advice, we will provide it with a *control*, saying where exactly to add the code, and ask for some parameters to identify which field was set:

Listing 1.5: Advice method for setters

```
LogSetters >> logField: aNode

    <advice: #setters control: #after arguments: #(node)>

    self log: aNode variable.
```

The control here is *after*, meaning the code gets executed after the original statement. aNode will give us a reflective representation of the statement the advice is installed on. According to our pointcut definition this should be an assignment node. Note that the advice is now on the instance side of the aspect, since it operates with its state. How many instances an aspect has and per what object they are created is configurable and explained later.

Finally, log: will do something like:

Listing 1.6: Logging method for setters aspect

```
LogSetters >> log: aVariable

    log at: aVariable put: (log at: aVariable) + 1.
```

The result is an aspect that hooks into each setter method of MyClass and counts all assignments per variable.

1.3 Thesis proposition

REFLECTIVITY [Denk08a] brings advanced reflection with a powerful interface to the programer. However, it stays more on a base level and lacks a high-level interface that would make it directly suitable for the user. DYNAMIC ASPECTS is the effort to create such a high-level interface in the specific direction of AOP. It's a lightweight implementation for Squeak Smalltalk. It shows how REFLECTIVITY can easily be extended to support aspect-oriented programming.

Many existing AOP implementations introduce a set of new features around an existing language, like new syntax, compilers, IDE tools, and others. DYNAMIC ASPECTS shows that it is possible to do a powerful AOP implementation without all those features. It integrates nicely with existing concepts of Smalltalk and stays simple. Instead of using special constructs for advices and aspects, they are just mapped to methods and classes. This gives aspects all the benefits from the base system and keeps it uniform. Also, it makes aspects subject to their own application, since aspects are themselves presented as the units they operate on. The pointcut language

of DYNAMIC ASPECTS is so simple, one actually would not call it a separate language, yet it is more powerful and allows more fine-grained selection than others.

The combination of existing concepts in Smalltalk and REFLECTIVITY gives DYNAMIC ASPECTS truly some features not found in many other implementations. REFLECTIVITY's sub-method level reflections allows to annotate on single statements inside methods in a very clean and generic manner. Furthermore, DYNAMIC ASPECTS shows how control flow can be based on contexts instead of the classic approach of looking on the execution stack. And, it presents a more generalized version of control flow called *flow pointcuts*.

1.4 Document outline

The following provides some guidance on the important chapters of this document. For a full overview see the contents on page vii.

Chapter 2 (p. 7) gives a more technical introduction into AOP than this chapter, without focus on a specific implementation.

In **chapter** 3 (p. 15) shortcomings of existing AOP implementations will be discussed and regarded as motivation to create a new implementation.

Chapter 4 (p. 19) picks up the problems from the previous chapter, and presents DYNAMIC ASPECTS as a solution to them. It presents the architecture and design of it without going to much into the implementation details.

Chapter 5 (p. 39) addresses the user. It shows how to program aspects in DYNAMIC ASPECTS and presents its user interface.

Chapter 6 (p. 47) gives some introductory examples, applying what was presented in the previous chapter.

Finally, in **chapter** 7 (p. 57) there are some concluding thoughts on the implementation, including pointers for future work.

The appendix holds some related information on the work and on this document. There is also a glossary in **appendix** C (p. 67), that gives some short help on the most important technical expressions used during this document.

Chapter 2

Aspect-Oriented Programming

For the remaining chapters it is important to know a bit more about the technical details of AOP. Therefore, in this chapter the basics about AOP are presented, without focus on a specific implementation.

2.1 Motivation

AOP has originally grown from OOP. The main reasons by which it is motivated are *cross-cutting code* and *pluggable behavior*:

- In OOP the units of modularization are classes and their objects. Writing classes means encapsulating state (variables) and behavior (methods). Sometimes a problem arises with functionality that crosses these encapsulating units, leading either to cross-cutting code or to the need of adapting the code for different encapsulation. The former means that code for one enclosed feature is spread over multiple classes and methods, leading to tangled code and poorly following the proper rules of modularization in OOP. The latter may not always be possible, if there are multiple features contradicting each other in the way the code should be modularized. Apart from that, redesigning the code can also mean a lot of additional programming work.
- Through its reflective nature AOP also qualifies for annotating existing systems without touching the original code. This can be of great value if the the added code is only of temporary use. There is no need

to hack into the original system, and annotations can easily be identified, resulting in fast removal or deactivation. Some implementations also allow for dynamic addition of AOP code, eliminating the need to recompile or even restart the system.

So far, this puts AOP rather into the direction of modifying and enhancing existing programs. In the context of Aspect-oriented software development (AOSD) there are also ideas to incorporate AOP in the very early development process of software [Rash04]. This can mean that aspects become an integral part of the software and are explicitly planned for the design or architecture of an application [Hane]. Consequently the software then depends on AOP, in opposite to just using AOP on top of some application. It can also mean, that software is developed with the fact in mind that later it should be easily extensible with aspects.

The bottom line is that AOP stays a pseudo-paradigm and is used on top of another paradigm. AOP can be built not only on top of OOP but also on other paradigms. The common idea always seems to be that AOP should capture *cross-cutting concerns that cannot be modularized using the primary programming paradigm*.

2.2 Technical jargon

AOP brings a few special terms along, that are important to know before proceeding.

Aspect Aspects describe cross-cutting concerns, meaning code that violates the separation of concerns. As such they are regarded more a logical construct describing features or functionality in a program. In AOP they are substantiated as units of modularization the programer writes. Aspects usually cover a set of advices and pointcuts, and interconnect them.

Advice Simply a block of code that represents the behavior or functionality to be added. Advices are different from methods in that they are defined not for a specific class, but for an aspect. That can cause some difficulties when needing to access the internals of a class. An advice is in fact defined outside the code's class it operates on.

Join point Join points are places in the execution of the system where aspects meet the original code. For example a method execution or a field read can be join points. Join points are the locations where behavior is added by aspects. The available types of join points depend heavily on the language and AOP implementation that are used.

Join point shadow The shadow of a join point is the actual piece of code that corresponds to the join point. It is the static projection of a join point. This is used by the weaving process to know the exact location in the code where something needs to be modified.

Pointcut A pointcut is basically a specification of a set of join points. Usually it presents the user a more high-level interface for join points. Pointcuts can be composed and can describe more complex scenarios. Finally, they are mapped to a set of join points. Pointcuts have a more descriptive nature and do not directly point to concrete locations in the system. They can also contain non-locational specifications, like conditions.

Weaving This is the process of installing the aspects on the original code. Most often this is a far more complex operation than just loading classes or starting a program. The original code and the aspects are merged together, which is called *weaving*. Eventually, the executing code of an application is different from the original code. The weaving may happen on the binary code directly, or also on a higher level, like for example on the source code. Often this requires additional tools, like an aspect weaver, a tool similar to a compiler. Many implementations weave aspects after the compilation. This also means that weaving needs to be redone after every recompilation of the code.

Reflection In very short and abstract words reflection means making a program the domain of another program[1]. In OOP this would mean to present the concept of classes, objects and messages itself as classes and objects. The typical example here is, that classes are as well objects. They understand messages like any other objects, and know what their superclass is or how many methods they have. A lot of things in AOP resemble the mechanisms used in reflection. Even an AOP implementation itself can be fully based on reflection.

2.3 Overview

In this section we will pick up the terms from the last section and try to connect them, giving an overview of how AOP works.

Figure 2.1 shows the reflective representation of a class with its methods. Depending on the level of granularity allowed in the reflective system, there may also be a more detailed representation of methods on a sub-method level. We will assume that this is the case here. The pointcut specifies locations in the code where to add behavior, and the advice is just normal code as in a method body.

[1] or even itself

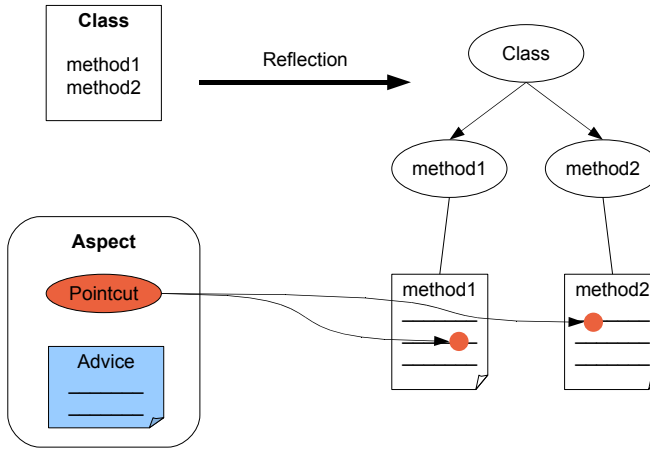


Figure 2.1: Overview of AOP

Figure 2.2 shows how the weaving process works and what the final result of installing aspects is. The ellipses in the figure represent the class and its methods as equal entities that can all be accessed in a reflective way. With sub-method level reflection even single statements inside methods can be accessed, as shown by the boxes with lines below the method objects. The aspect includes the pointcut description, which in some way points to methods or statements inside methods, and the advice that usually consists of some program code and is connected to a pointcut.

Through weaving the original method gets modified, so the resulting compiled code after installing aspects is not anymore the same. The advice code is not just inlined to the original code, but a special piece of code is inserted that will call the advice at the proper location during execution. Note that the advice is executed inside the aspect *and not inside the modified code*, it is only called from there.

2.4 Pointcuts and join points

So far the only examples of pointcuts were methods or single statements inside methods. To be honest, most AOP implementations do not support

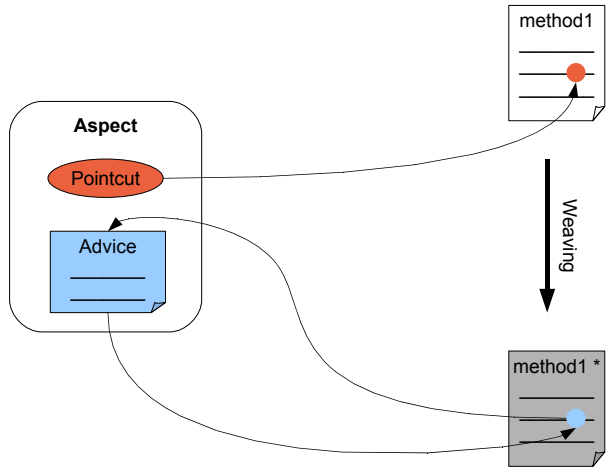


Figure 2.2: The weaving process

the selection of single statements. But there are various other pointcut types allowing more complex descriptions. The most important categories will be discussed in the following to give an idea of what is possible with pointcuts.

There are not only filter pointcuts (describing a location in the system or code). It is also possible to make conditional specifications. In this case the pointcut is not used to determine code locations to install aspects on, but it is used to determine at runtime if there is a match and only then the advice is called.

Methods Select whole methods to install advices on. Sometimes there is a distinction between *execution* and *call* of a method[2]. Also for some languages constructors and initializers are treated separately. The filtering can include various properties of methods: signatures, return and argument types, name, modifiers, and more.

Field access Reading and writing variables.

Condition Specify any block of code that returns a boolean value. The evaluation can happen when the pointcut is created or each time a

[2]The call happens before the method is executed. In some cases this difference can be of use.

corresponding join point is reached during execution. Mostly this is specified in combination with a filter pointcut. The advice is then only called if the condition yields true.

Control flow This is like a condition, but with focus on the method stack. It allows one to specify methods the execution should be in or not. Sometimes even the exact depth can be specified, interesting for recursive calls of the same method.

Reifications and runtime parameters In some situations one wants to access elements of the executing code that are not directly accessible from outside or are different for each execution. For example the user could define a condition depending on the sender of a method. In other cases, information first needs to be reified before it can be used by pointcuts. For example, the method stack. All these arguments can be very useful to define pointcuts.

Advices Since advices are very similar to method bodies, it is only natural that they are subject to their own application. When installing aspects on methods, sooner or later the idea of installing aspects on advices themselves comes up.

Filter pointcuts often accept wildcards or regular expressions to specify methods, classes and other code fragments. Besides the basic pointcut types it is also possible to combine multiple pointcuts into a single new one. Simple compositions with “and”, “or” and “not” are often all that is needed.

Some pointcuts can be specified in multiple ways. For example selecting two classes can either be done using a combination of two pointcuts each describing one class, or by using a single pointcut that uses a regular expression to match both classes.

2.5 Advices

As mentioned before advices are very similar to method bodies, and as such there is not much interesting to say about them. The first question that appears concerns code that is defined somewhere else as the state it operates on. Although advices can be seen as code that is appended to methods, they are executed outside of them. *So, how do advices access the original state?*

Weaving does not mean to inline code as a preprocessor in C can do. The execution of an aspect is merely a call to the advice. That means that advices need some possibility to access the object’s private data they are executing for. And that is exactly what advice arguments are about. An advice can

for example request the object it is executed for. This one is then given like a method parameter to the advice.

The second question arises when thinking further from the answer to the first question. Then, advices are like methods but defined outside the class they annotate. They also want to have a state they can manipulate, but it is not the same as the object has they are executed for. They can access the object's state but maybe they also want to have a separate one. *Where and how is this state managed?* The answer is the aspect itself. It acts as some kind of unit that provides a state for its advices.

Traditionally OOP manages instance and class variables. Of course there should be the freedom of making state available only to selected places in the code. Namely, we are talking about *scope* here. In OOP there is instance and class scope, and maybe also a global and a special scope. For aspects a scope makes sense as well. The aspect's state could for example be managed per object that calls an advice from the aspect. Or maybe it should be managed per call.

Coming to the conclusion, stateful aspects should also have some kind of scope. It might be even more sophisticated than the one in OOP. Whether it is implemented with the same mechanisms as in OOP, namely instantiation, depends on how aspects are implemented.

2.6 Connecting pointcuts with advices

One thing that has not been mentioned so far is how pointcuts and advices are interconnected. Some AOP implementations pay greater attention to modularity of pointcuts and advices, like for example JAsCo [Vand], others do less. JAsCo uses the term *connector* that fits very well.

Connecting a pointcut with an advice means that the advice is executed for each runtime event that is matched by the pointcut. High modularity allows to connect pointcuts and advices in any many-to-many relation, and maybe also share them among different aspects.

Another thing a connector specifies is the *control* of an advices. This says when exactly the advice is executed. Normally the possible values for the control are:

before Execute before the original code, after advice execution the original code is resumed.

after Execute after the original code. For a method this can mean that the return value is then the one from the advice, and not from the original code.

instead Execute instead of the original code. Often there is an option to proceed the original code from within the advice[3].

around Depending on the implementation this can mean the same as instead, or just before and after.

2.7 Weaving

Weaving is something that heavily depends on the specific AOP implementation. The original term comes from the idea of low-level byte-code weaving. Aspect code is woven directly into the byte-code with the help of external tools. It seems a natural approach to modify the code just after the normal toolchain of compiling code has finished. But thinking of reflection, there is actually a second way to do it. Reflection also covers the possibility to modify code in a reflective manner. One could modify the code in the reflective representation, and the changes are then automatically applied. This would maybe give a nicer interface, depending on how reflection is implemented. [4]

There are many different approaches how aspects are defined. Some implementations rely on additional syntax and language constructs. Consequently the aspect code is strictly separated from the normal code, and also the installation is most likely a separate process. In highly dynamic languages, AOP can go as far as being just another programming library. Of course this presumes dynamic compilation and loading of code, a good reflection interface, and more. But eventually it gives the user an AOP implementation where all he has to do is using an additional API. As can be seen by reading on, DYNAMIC ASPECTS is exactly such a solution.

[3] Some implementations also call this *"replace"*, despite the fact that it is not replacing the original code in every situation.

[4] There are also other techniques for weaving. For example, it is possible that aspects make their changes directly to the source code and use the normal compile process to become effective.

Chapter 3

Problem discussion

In this chapter we will discuss the drawbacks of current AOP implementation and then give a short motivation for creating DYNAMIC ASPECTS.

3.1 Problem description

Most AOP implementations come with a bunch of new tools and changes to the existing development environment. The main components are:

- A special weaver program. The usual write-compile-run cycle is lengthened by an additional explicit weaving step.
- Additional syntax to define aspects. This brings additional effort for the user to learn it. Also, aspect code is not compatible with old tools, e.g. the source code editor first must be enhanced.

Weaving is preferably something that is done without a lot of interaction needed from the user. Unlike compiling there are not that many options to this process, so it is suited for automation. To avoid having an extra weaver program that operates on compiled code, the weaving can be done before compiling. That means aspects are installed directly on the source code and get automatically active by just compiling source. Yet it is more comfortable if this source code modification can be done in a reflective manner, which implies a good level of reflection in the language. Hence, the weaving also gets more dynamic, since it can be done directly from within a running application.

Defining new syntax for aspects has the disadvantage that all tools around a language must be adapted. It also makes it more complicated to use

aspects not on top of objects, but again on top of other aspects. In the case of an already very complicated syntax of the base language, the aspect language may inevitably also become very complex.

3.2 Related work

In the following we will shortly look at two AOP implementations with their characteristics and level some criticism on them.

3.2.1 ASPECTJ

ASPECTJ [AspJ] is an AOP implementation on top of Java. It is a very popular and mature implementation that has even obtained good acceptance in the industry. Its characteristics are mostly derived from the ones of Java. The pointcut language includes many types of pointcuts. This is necessary since there are so many language constructs in Java, like control structures, access modifiers, abstract elements, instance initialization, etc.

ASPECTJ comes with a set of new tools. These cover weaving, generating documentation out of aspect source code, Ant support and a special code viewer. All these tools existed before ASPECTJ and then had to be adapted or rewritten to become usable also for aspects. All these tools blow up the installation of ASPECTJ and bring lots of additional work. To be fair, one must say that ASPECTJ has reached a good integration into the Eclipse IDE, which makes it again easier to use.

ASPECTJ is an example of a more static implementation. Weaving is done on byte-code level, after normal compilation. At runtime no dynamic modification of aspects is possible. The language includes a lot of specialities that are caused by and bound to Java. For example, there are *inter-type declarations*, that make it possible to add elements to classes without modifying their original source code. In Smalltalk, this is not part of AOP, but part of the normal language, and it is just called *extensions*.

3.2.2 AspectS

ASPECTS [Hirs03] is an AOP implementation for Smalltalk. The whole aspect language is mapped to a set of new classes instead of new syntax. Defining aspects is done just by doing normal OOP programming. Weaving is accomplished by using method wrappers and is a very dynamic process. Together with PERSPECTIVES [Hirs02] it even supports contexts. ASPECTS shows how easy an AOP implementation can be that is built on top of such

a dynamic and uniform language as Smalltalk is. It also shows that pre compile time weaving with reflection is a good way to go. Traditionally Smalltalk's reflection is on a higher level than the one in Java, although work has been done to enhance Java's reflection capabilities [Tant01].

However, what AspectS still misses is an explicit way of defining aspects. There is no predefined style of clearly structuring join points and advices. The user ends up with a huge block of code instantiating classes, specifying attributes and glueing them together. What AspectS also lacks is a real powerful pointcut language. The user merely deals with join points directly. In addition, join point granularity stops at the level of methods, there is no sub-method level at all.

3.3 Motivation for a new implementation

AspectJ is a major and productive implementation. On the other hand, AspectS shows the great potential of Smalltalk in this field, but it lacks the maturity AspectJ has. Also, AspectS has limited power in specifying pointcuts and defining aspects.

A lot of AOP implementations know the idea of conditions and control flow. But the integration of contexts is not widespread. Contexts are a consequence of complex conditions. They are globally addressable conditions, that can be composed and have an object structure. Like that it is much easier to express complex conditions. Control flow is as well a context. It is a condition that depends on the execution context — the method stack — a program is currently in. So, what we are also missing is an explicit implementation of contexts, that would help is in superiorly expressing conditions and execution flow.

The way to go seems to follow the idea of AspectS and make more out of the potential of Smalltalk. Reflection turned out to be a very good instrument to base AOP upon. Yet, something on a sub-method level would be welcome. Partial behavioral reflection has been used to realize AOP [Rodr], but it provides only a limited form of dynamic AOP, as the underlying Reflex system does not support *unanticiapted* partial behavioral reflection [Röth05]. Also, there should be some constructs to clearly structure aspects, that help to keep the code clean. We are looking for a lightweight implementation, without the need of lots of additional tools, that integrates nicely into an existing Smalltalk environment, and allows the user to do really powerful and dynamic AOP.

In the next chapter, we will see how such an implementation could look like.

Chapter 4

Solution: DYNAMIC ASPECTS

In this chapter we will present DYNAMIC ASPECTS as a solution for what we encountered as problems in the last chapter. This will cover the very basic architecture of DYNAMIC ASPECTS and some of it's design decisions.

4.1 Architecture of DYNAMIC ASPECTS

Before we go into the gory design details, we will highlight some of the basic characteristics of DYNAMIC ASPECTS. The following points will give some ideas on how DYNAMIC ASPECTS fits into the typical AOP implementation.

- Aspects are all subclasses of DAAspect, which is in turn a subclass of Object.
- Pointcuts are class side methods of an aspect, that return an instance of DAPointcut.
- Advices are instance side methods of an aspect, their pragma indicates the advice control, references a pointcut name (a pointcut method on the class side) and specifies the advice arguments.
- DAPointcut instances can be evaluated to return instances of DAJoinPoint , which return nodes in the reflective representation of methods.
- Installing an aspect means calling DAAspect class >> install, which is normally done automatically every time anything changes on an aspect class, unless the aspect is disabled.

This already indicates a lot about how DYNAMIC ASPECTS works. Aspects as classes has the advantage that inheritance can be used in aspects for code reuse just as with any other classes. Hosting pointcuts and installation on the class side makes a specific aspect globally addressable. An aspect can create any number of instances depending on the scope, and advices will directly act on the instance respecting the particular scope.

4.2 Reflection with REFLECTIVITY

The special thing about DYNAMIC ASPECTS's implementation is that it fully relies on reflection. This includes the weaving process, as well as the pointcut language. REFLECTIVITY is a system that brings together various tools in the area of advanced reflection and is suited for DYNAMIC ASPECTS's implementation. REFLECTIVITY is really decisive for the design of DYNAMIC ASPECTS.

Traditionally Smalltalk has good reflection capabilities. The basic reflection in Squeak is *structural* (representing the static structures of programs) and allows *introspection* (*"look but don't touch"*). There is also *behavioral* reflection (representing runtime abstractions of a program) and *Intercession* (modifying the program in a reflective manner). The last two are especially important for DYNAMIC ASPECTS. Intercession allows the code to be modified, which is what aspects do, and behavioral reflection gives more details of the code, e.g. message sends or variable access [Tant03]. REFLECTIVITY takes partial behavioral reflection to a new level by making it completely dynamic [Röth08].

Another unique feature of REFLECTIVITY is sub-method level reflection [Denk07]. It allows one to operate on single statements inside methods in a reflective way. This allows aspects to really affect any place in the code, and that in a very uniform way.

As we can see in figure 4.1 there are basically three layers below DYNAMIC ASPECTS. Without knowledge about REFLECTIVITY, some explanation and clarification of how they are related to DYNAMIC ASPECTS is needed. The fourth layer on top of REFLECTIVITY is DYNAMIC ASPECTS itself. We will start from the bottom layer and continue to the top:

1. The **Persephone**[Mars06] framework manages a *reflective method* for each *compiled method*. Reflective methods are high-level descriptions based on an extended AST (abstract syntax tree). Compiled methods are lazily created when they need to be executed. If the reflective method changes, the compiled method is discarded and recompiled on the next execution. This can happen either by changing the source code, or by directly changing something on the AST of a reflective

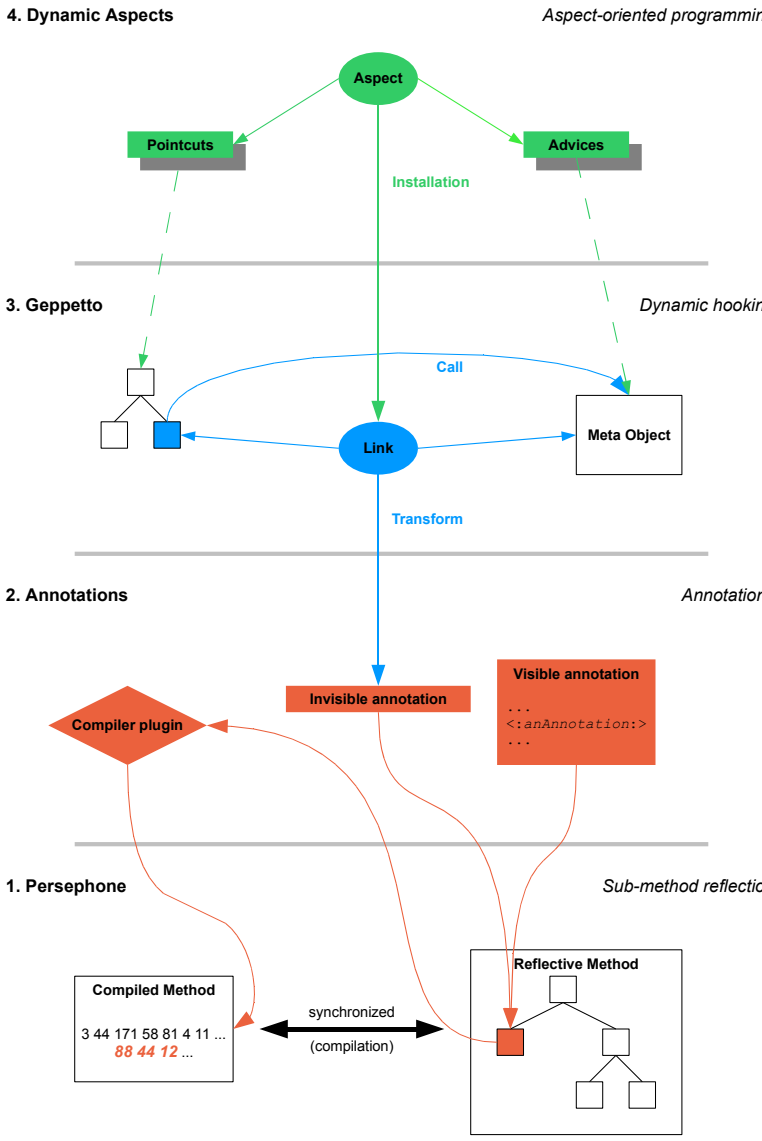


Figure 4.1: DYNAMIC ASPECTS on top of REFLECTIVITY

method. From the perspective of execution, both methods are always kept in synchronization.

The level of granularity in sub-method reflection is very high. Every single statement, like messages, returns, sequences and also variables, are reflectively represented. The elements of reflective methods are *nodes* and build a part-whole hierarchy. As we will see later in the other layers, reflective methods are used in many locations. There is an interface to search through the nodes, to modify them, and we can even programatically generate methods with reflective method nodes, e.g. create an object tree instead of writing source code.

2. **Annotations** are a part of PERSEPHONE. The idea is to implement annotations on top of reflective methods. These *annotations* can either be visible or invisible. Visible annotations are written in the source code in a special syntax, like <**:anAnnotation:**>, invisible ones are added directly to reflective method nodes. With the help of a compiler plugin annotations are then translated into code. Eventually annotations manifest as additional byte-code in a compiled method.

   The annotation framework is very extensible. We can define whatever types of annotations we like, and write the appropriate compiler plugin that interprets them. Annotations do not necessarily need to be translated into compiled code, they can also for example only do checks on the code during compilation.

   Through sub-method reflection PERSEPHONE can install annotations on every single statement of the code.

3. **Geppetto** uses invisible annotations to add its so called *links* to nodes of reflective methods. Links are annotations that extend the original code, and then end up in *hooks* in the compiled code. The hook of a link calls an external object, the so called *meta object*, with appropriate arguments from the currently executing environment. GEPPETTO defines a protocol how these meta objects are called and provides a good set of reification arguments that can be passed. The meta object can be any object in the system to call a message on. It is important to note, that the execution here takes place on some kind of meta layer, instead of the base layer[Denk08b]. The normal execution is interrupted by the hook, and code from the meta layer is executed.

   On the other end GEPPETTO directly operates on the nodes of reflective methods. When installing a link we directly pass it the nodes to be installed. Installing involves annotating the appropriate nodes and discarding the currently cached compiled method, so it is recompiled on next use and includes the annotated code.

   One way to look at it is that actually the idea here comes very close to

AOP, just that we are missing the concepts of pointcuts and aspects. So this is practically a low-level interface for AOP.

4. Finally, **Dynamic Aspects** enhances GEPPETTO by two integral parts. First, it introduces *pointcuts*, a high-level interface to describe reflective methods nodes and more. Second, it encapsulates pointcuts and advices into aspects and presents them easy programmable units. Additionally it cares for the integration in the programer's user interface.

   In the course of applying aspects to the system, one could abstract and say that *aspects* become *links*, *pointcuts* become *reflective method nodes* and *advices* become *meta objects*. Although in reality the process of installing aspects is a bit more complicated as shown in the following.

So, DYNAMIC ASPECTS's implementation takes place exclusively on the topmost layer. All other parts are given by REFLECTIVITY and its tools. DYNAMIC ASPECTS makes heavy use of reflective methods, to evaluate pointcuts, and links, to actually install aspects.

4.3 Aspects are classes

When comparing aspects with classes, one can actually discover a lot of similarities. Both are the units of modularization in their respective programming paradigm. In OOP it is the class, in AOP it is the aspect. In DYNAMIC ASPECTS *aspects are classes*. From the implementation point of view, DAAspect is just a subclass of Object, with the indirection of DAProtoAspect. This is necessary to introduce some things that can not be directly implemented in DAAspect. All aspects are then defined globally like classes.

The fact that aspects should be classes, and wanting to encapsulate advices and pointcuts in aspects, almost inevitably leads to the decision to implement pointcuts and advices either as methods or variables. These two are basically the only elements we have available in classes. For advices the decision is simple. They resemble methods so much that they *"actually are methods"*. A pointcut method is just a method that returns a pointcut object. The connection between pointcuts and advices is given by the pragma in the advice, as we will see later in this chapter.

The implementation of aspects takes place on the *instance side* as well as on the *class side*. The former hosts the advices with the state they manipulate, the latter is used as a global access to the aspect, allowing to query and install/uninstall the aspect. The pointcut methods are defined on the class side since they have a rather global scope compared to the advices.

All these design decisions bring a set of advantages to aspects:

- Aspects are perfectly integrated into existing concepts. Coding aspects is not much different than coding classes. Also the implementation is simple and very lightweight.
- Aspects profit from inheritance. It makes them modular and reusable, just as classes are.
- AOP normally addresses the things we have in OOP, say classes, methods and such. Addressing itself can be difficult since it must be expanded to support its own special constructs, that are different from OOP. But here, applying aspects to other aspects is just as easy as applying aspects to classes. To say it again, *aspects are classes*. It is similar to the often mentioned fact that in Smalltalk classes are objects. Sure, they are special objects, but *"they can do everything normal objects can do"*. This results in great uniformity and simplifies things, eliminating a lot of special cases.

A definition of an example aspect might look like this:

Listing 4.1: Creating aspects means subclassing

```
DAAspect sublass: #MyAspect
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'MyCategory'
```

This defines a new aspect with name MyAspect. It is completely the same as defining subclasses. The pointcut is then defined with a class method:

Listing 4.2: Defining a pointcut

```
MyAspect class >> transcriptShow

    <pointcut>

    ↑ (TranscriptStream >> #show:) asPointcut.
```

This returns the method we use when doing Transcript show: ... as a pointcut. The advice on the instance side will make #show: always put a carriage return after a printed string:

Listing 4.3: Defining an aspect

```
MyAspect >> extraCR: transcript

    <advice: #transcriptShow control: #after arguments: #(object)>

    transcript cr.
```

Note that in this case we are putting the advice code and its connection to the pointcut into the same method. For better reusability one could also extract the code into a new method and just forward the call from the advice to it. If we want to connect multiple pointcuts with the same advice, we just write a new composed pointcut, that includes all pointcuts we want, and connect the advice to this one.

Finally, one may doubt with good reason that it is a good way to code this behavior like that, but we remember that it is only an example.

4.4 Advices are methods

As mentioned before advices are instance methods on aspect classes. The reason why they are on the instance side is, that advices may want to have a state that may also be shared with other advices. Pointcuts on the other hand, are a more static thing. They are needed by the aspect class at installation time, before instances are created, and not needed anymore until an aspect needs to be reinstalled or is uninstalled.

To distinguish advice methods from other methods in the class, they are declared with an advice pragma. The minimal form of the advice pragma is:

```
<advice: #pointcut control: #control>
```

The symbol #pointcut must match the name of a pointcut on the class side. #control is one of the valid control symbols like #before or #after. There are currently two options that can be added to the pragma: arguments: and disabled:. So a comprehensive advice method might look like this:

Listing 4.4: Full advice pragma

```
advice: obj node: n

    <advice: #pointcut control: #before arguments: #(object node) disabled:
      true>

    ...
```

This defines an advice to be installed “before” the pointcut with name “pointcut”, that has access to the object and node reifications by the names “obj” and “n”, and that is currently disabled, meaning it is ignored on aspect installation. What follows the pragma directive is the normal method body, the actual advice code.

4.5 Pointcuts are descriptions

For an aspect pointcuts are class methods, marked with the <**pointcut**> pragma, that return a valid pointcut structure. In this section we will look a bit closer at pointcut objects and how they are actually used.

From the implementation point of view there are actually only two types of pointcuts. *Filter pointcuts* and *condition pointcuts*. As a user we want to say *where* to install aspects, but we also want to specify *when* (in the meaning of "*if*") the code of an aspect gets executed. Beyond that, there are special pointcuts that make additional specifications. They are distinct from any others and therefore not categorized.

It is a requirement of a pointcut language, that it presents a simple interface to the user, which allows him to express precisely what he has in mind. Pointcuts are high-level descriptions, that are understandable by the user, and that can be modified and even extended. Consequently pointcuts are merely a data structure to hold the user's specifications and to store/order them in a generic form. There are different pointcut classes, to indicate the different pointcut types, that just store the user's specified values and arrange them in a tree structure. The tree structure comes from the requirement to freely combine any pointcuts with each others.

4.5.1 From pointcuts to reflective method nodes

Figure 4.2 shows a simplified UML with the relations between pointcuts, join points and links. Pointcuts form a part-whole hierarchy following closely the composite pattern. Their evaluator is a visitor that walks over the tree to generate the corresponding join points. All subtrees or compositions of pointcuts behave the same.

Thinking of the final goal to translate pointcuts into something that can be mapped to GEPPETTO links, it is clear that some functionality to evaluate them is needed. The first step is to reduce them to join points. Join points keep some characteristics of pointcuts, but also lose some and gain others. They are no more as comprehensible as pointcuts, and not intended for the user to deal with directly. A tree of pointcut objects transforms into a set of join points. So now we do not have anymore a tree, but just a set of objects. Therefore all objects are of the same class. The reason to do this is mainly to eliminate compositions and come closer to the form of a link.

The second step is to evaluate join points. Like pointcuts they still have some descriptive form, meaning they do not directly reference classes or methods in the system. They hold a description of it, and when evaluated return the concrete references. This can help when code in the system

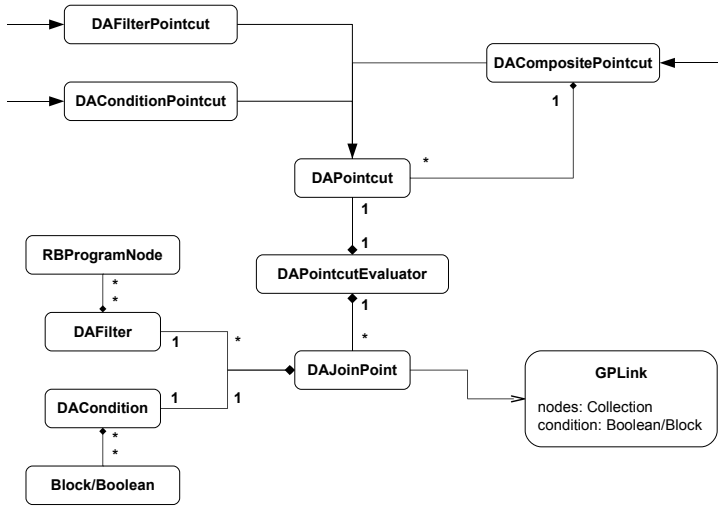


Figure 4.2: Simplified UML of pointcuts and join points

changes and for example method objects get replaced. In this case join points just need to be re-evaluated instead of re-created.

A join point then comes very close to a link. It evaluates to a set of nodes and contains exactly one condition, that was combined from possibly multiple conditions. This is what can directly be translated into a link.

DAFilterPointcut and DAConditionPointcut have various specialized subclasses. All pointcuts are either instances of concrete subclasses or compositions of them. The various pointcut language constructs found in other languages are here all implemented as instantiation methods on the class side of DAPointcut. Some of them directly instantiate a specific pointcut class, others already return compositions.

4.5.2 Filtering

Filtering is heavily oriented on what links can do. The only places links can be installed on are statements in methods. In Smalltalk source code is actually only written for method bodies. So this is all we need. It ultimately gives us two kinds of filters, selecting statements inside methods and selecting the method as a whole. Furthermore, methods are organized into

classes, and again classes are put into categories. This finally gives us four types of filters.

In the chain from statements, over methods, classes, up to categories, each element can be assigned exactly to one higher ordered element. Meaning, every statement is located in exactly one method, each method has one class, and each class is assigned one category. This makes filtering rather easy. Each join point hosts a set of filters for each type.

The evaluation of join points only yields nodes, but not categories or classes. All statements in a method are organized below one top node. That is why selecting a method is the same as selecting statements inside methods. The filtering starts with the categories, selecting only those that match all category filters of the join point. Then, all classes within the remaining categories are filtered with the class filters, and so on and so forth, until filtering single nodes of a reflective method.

Filters can be specified either on the name of an element, or with a customizable block, that is executed for each element, checking whatever we want, and returning true or false. The name can also be given as a simple regular expression.

If no filters for a specific type are specified, by default all elements are selected, except for the statement filter, that by default just selects the top method node.

4.5.3 Compositions

The composition of pointcuts is what makes them modular and reusable. The composition operators implemented so far are #and:, #or: and #not. Since composition should work uniformly on all pointcut types, these operations must work for all types and also for merges of them. The interpretation is different for each type. Combining filter pointcuts is translated to set operations. For example, combining two method filter pointcuts with "or" means taking the union of methods they select. Whether for condition pointcuts composition is simple translated to a logical operation. Table 4.1 summarizes what the different possible combinations mean.

F: Filter, **C**: Condition	**F**	**C**
F and	Intersection	Add C to F
F or	Union	New default F, add C
C and	-	Logical AND
C or	-	Logical OR
not	Complement	Logical NOT

Table 4.1: Pointcut compositions

On evaluation all pointcuts must somehow result in join points; that is the only form that can then be translated to links. Each join point has multiple filters for each element type, like classes, methods, and so forth. "Anding" two filter pointcuts just means adding multiple filters in one join point for one element type. "Oring" means creating a new join point with new filters. When evaluating join points, each one returns only nodes that match all its filters, but the nodes of different join points are again unified.

Conditions are as well set for each join point. Since links only support one condition, multiple conditions are combined in DACondition itself, which any condition used should be a subclass of. In contrast, oring two conditions is not done in the condition itself, but on the level of the join point. That means, a new join point is created and the condition is set.

Each join point only has one condition. Anding a condition with a filter pointcut just means setting the condition on the corresponding join points. Oring a condition with a filter pointcut means creating a new filter pointcut with default filters and then adding the condition. It's the same as specifying just a condition as pointcut without filters.

Specifying the *"if"* of a pointcut without saying *"where"* makes no sense. The *"where"* must always be given. So, for each pointcut at least one join point is created. Anding can be done within one join point, by further limiting it's filters or condition. Oring means to introduce new join points.

Combining condition blocks

The composition of conditions is not as trivial as it may appear first. Most conditions are given as blocks, since the code should first be evaluated at runtime. Unfortunately Squeak does not have a programmatic interface to create blocks; there is only the literal way of using the block syntax ["..."]. The disadvantage is that the code inside the block must be known when writing the source. Creating and compiling it at runtime from the combination of two other blocks is possible, but maybe a bit ugly. That is why a new class DACondition was created, that mimics the interface of a block closure, so it can serve as replacement for traditional blocks.

Since conditions can have arguments, and some of them might be the same for all condition blocks, it is important to consider them as well. A combined condition takes the union of all arguments from it's member conditions. On evaluation it evaluates all it's members in turn, with the appropriate subset of arguments, and returns the combined value. Of course, in our case the return values of the blocks must be booleans.

As an example we consider the following two conditions:

Listing 4.5: Wrapping block conditions

```
c1 := DABlockCondition block: [ :x | x > 1. ].
c2 := DABlockCondition block: [ :x | x > 10. ].
```

The combination then takes only one argument:

Listing 4.6: Combining conditions with equal arguments

```
c3 := c1 and: c2.
c3 value: 5. "⟶ false"
c3 value: 11. "⟶ true"
```

If we rename the argument from the second condition:

Listing 4.7: Explicitly specifying condition arguments

```
c2 := DABlockCondition block: [ :x | x > 10. ] arguments: #(y).
```

The parameters of both conditions are no longer regarded to correspond to the same argument. The combined condition then takes two arguments:

Listing 4.8: Combining conditions with different arguments

```
c3 := c1 and: c2.
c3 numArgs. "⟶ 2"
c3 value: 2 value: 11. "⟶ true"
c3 value: 1 value: 11. "⟶ false"
```

Of course, we can always combine the condition ourselves in the code blocks. But DACondition is for the case where conditions are created independently and then must be combined without rewriting the block code.

4.6 Weaving aspects

For the user, defining aspects is all he has to do. But of course just defining them does not yet change anything in the system. So far, we only looked at the coding process; we did not yet see how the actual weaving works. In DYNAMIC ASPECTS *"weaving"* is the same as *"installing"* aspects.

The reason why installation usually plays no role for the user is that aspects are automatically installed. The aspect class monitors itself for changes and reinstalls itself when necessary. There are already predefined methods how a class can react on its own changes. The different cases that occur and the methods that are used are shown in table 4.2.

Change	Method
Add/change aspect	#doneCompiling
Remove aspect	#unload
Add/change pointcut/advice	#noteCompilationOfSelector:meta:
Remove pointcut/advice	#deregisterLocalSelector:

Table 4.2: Installing and updating aspects

That is the way it is already implemented in Squeak. A class is a reflective representation from the system. Creating a new class means calling the subclass: message on an existing class. Adding new methods is also done via message sends to the class. Figure B.1 in the appendix will give some overview of the base hierarchy of classes in Squeak.

Adding or changing a class or a method comes to the same. Each time the new version of the class or method source code is compiled, a new object is created and the old one gets replaced. So, there is no difference in reacting on these two types of changes. Removing the aspect class has to make sure that it is first uninstalled, otherwise it might leave some installed links that then can not be assigned to any aspect anymore.

Adding, removing or changing selectors, is only of interest if it is a pointcut or advice method. It is difficult to treat selector changes as partial updates of an aspect. Virtually, only affected links of an aspect need to be updated. Moreover, advices only need an update if their interface changes, but not when their method body does. But on the other hand, pointcut methods, for example, might call other pointcut methods of the same class to reuse them. That is what makes it very difficult to track all dependencies, and that is also the reason why currently aspects are always updated by reinstalling them as a whole.

Installing or reinstalling aspects is done on the class side of each aspect itself. The user has the control to disable the aspect, or disable single advices. The automatic installation of aspects always performs silently. If there was an error during installation, the necessary steps to clean up already installed parts should be taken automatically as well. The aspect stores the error from the most recent installation to indicate the user of the failure although the silent operation did not interrupt. The user can then manually reinstall an aspect, to see what the exact failure was.

4.7 Conditions and contexts

A valuable extension to pure filtering are conditions. They make pointcuts more dynamic. A condition does not influence where aspects are installed,

but controls *if* they are executed. Each time the execution reaches the hook of an aspect this one is only executed if the condition evaluates to true.

To make conditions even more dynamic they can use reification arguments. The alternative would be to put the condition code into the advice, but this would limit the reuse of advices.

Most times conditions are given as blocks containing a few statements. Another way to define conditions more globally are contexts. Internally, contexts are translated to link conditions, but other than the locally scoped blocks their code is defined globally as a subclass of DAContext. This is a better way to structure complicated conditional code, and make it globally accessible by a name. Furthermore, contexts can be combined like pointcuts. Table 4.3 shows the hierarchy of standard context classes.

DAContext	Standard contexts
DAActiveContext	Activatable contexts (singleton by default)
DAPerProcessContext	Instances are created per Process
DAComposedContext	Combine any contexts with each other
DAAndContext	Logical and
DANotContext	Logical or
DAOrContext	Logical not

Table 4.3: Context types

Normally contexts are derived directly from DAContext. The only thing one has to care is to overwrite the #isActive method. In some cases the state of a context is modifiable and that is what DAActiveContext is for. Such contexts manage a state for each instance and can be called activate and deactivate. In that case, it also matters how instances are created. The default is to have just a singleton, which is returned by #current from the class.

Composed contexts are never active contexts, there is no clear meaning of mapping the activation to the it's children. Composed contexts are like views, they return new instances for each composition operation and just alter the return value of #isActive by analyzing all their children.

To make an abstract example, we think of the fact that contextual logic often results in scattered code and a lot of if statements. This is a cross-cutting concern. And as such it is subject to be solved with aspects. Consider the following code of if statements:

Listing 4.9: Example of contextual if-code

```
Store>>store: arg

    self loggingEnabled ifTrue: [
        self isProductive
```

```
        ifTrue: [ self log: 'Storing'. ]
        ifFalse: [ self log: 'Storing: ', arg. ].
]

"..."
```

If logging is enabled, it makes a log entry that it is storing something, e.g. in a database. If the system is not running in production mode we even want to log what object exactly it is storing. For a non-production environment this might be a bit dangerous; who knows if there is any private data stored?

If we transform this to aspects, we first create some pointcuts for the method and the contexts:

Listing 4.10: Context pointcuts

```
storeMethod

    ↑ PC class: Store method: #store:.


normalLogging

    <pointcut>

    ↑ self storeMethod &
        (PC context: LoggingContext & ProductiveContext not).


productiveLogging

    <pointcut>

    ↑ self storeMethod &
        (PC context: LoggingContext & ProductiveContext).
```

Now we have two pointcuts normalLogging and productiveLogging that reflect our two possible logging conditions in the original code. storeMethod is not a pointcut but only reused in both pointcuts. The good thing with conditions is that we can compose them directly.

For the advice code, we also define two advices:

Listing 4.11: Transforming scattered if-code into advices

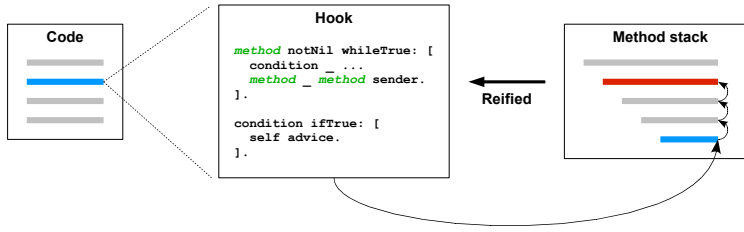
```
logNormal: arg

    <advice: #normalLogging control: #after arguments: #(arg1)>
```

```
    self log: 'Storing: ', arg.

logProductive

    <advice: #productiveLogging control: #after>

    self log: 'Storing'.
```

So, we have successfully transformed all of the if-code into aspects. Moreover, if we wanted to do this kind of logging in other locations in the code, we can just extend the pointcuts. Finally, our result is a basic implementation of context-aware aspects [Tant05].

4.8 Flow

A first application of pointcut compositions and contexts are *flow pointcuts*. To understand what the idea behind flow pointcuts is, we start with a slightly more specialized pointcut, the *control flow pointcut* or *cflow pointcut*, that is also more widely known.

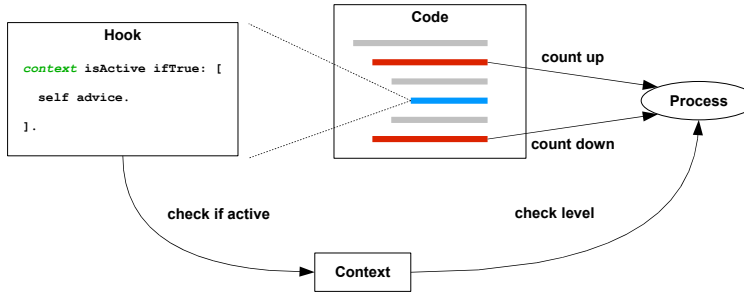



```
pointcut cflow: method.
```

Figure 4.3: Classical control flow pointcuts

The canonical control flow pointcut is illustrated in figure 4.3. The basic idea is easy to understand. A pointcut is given an additional condition, that checks what method stack the execution is in. Depending on what the condition computes during walking up the reified stack, the advice is executed or not. The use of this is that the pointcut can depend on the current execution stack. For example, an advice on a method should only be executed if this one was called from a certain other method. But not when it is called in another context. The expensive thing here actually is, that the whole stack must be reified. This is also why some implementations switch to a different technique of keeping track of the execution context. It can also be done with counters, as described later.

Figure 4.4 shows a different version of the pointcut; it is the one that is implemented in DYNAMIC ASPECTS as flow pointcut. What has changed from the classical version are two things:




```
pointcut1 in: pointcut2.

pointcut1 in: pointcut2 min: x max: y.
```

Figure 4.4: Generic flow pointcuts with counters

Counters The method stack is no longer reified. Instead some counters in combination with additional hooks are used. An alternative to using the stack is to simply install additionally one hook before, and one hook after the method we are looking for. To formulate this in AOP, we install an aspect on the method in question with a before advice that increments a counter, and an after advice that decrements the same counter. If the counter is greater than zero, we know we are

inside the execution of the method. If it is greater than 1, we even know that there is some recursion in the call. Of course, the counter must be managed per process, since multiple executions might run on the same code. Using counters also simplifies the case we want to pick a specific recursion depth. For example, we might want to limit the depth to at least 2 but no more than 4. This just corresponds to the value of the counter. With the old method of walking up the method stack, each increase in depth lengthens the walk, costing time and also memory for the reifications.

General flow Using counters the technique is no longer bound to the method stack. We can actually install counters on any pointcut and then check if the execution has already entered or left it. So the idea is to abstract the control flow to a general flow. The flow pointcut can be set on any other pointcut, it is a very generic form of the control flow pointcut. Additionally the flow is named and can be combined with any other pointcuts.

Coming back to the statement in the beginning of this section, the flow pointcut is actually a composition of two pointcuts. When looking at DAPointcut>>#flow:min:max:, we will see what the instantiation of a flow pointcut does.

Listing 4.12: Flow pointcut instantiation

```
flow: aDAPointcut min: minNumber max: maxNumber

    | context connectors |

    context := DAFlowContext
        pointcut: aDAPointcut
        min: minNumber
        max: maxNumber.
    connectors := {
        DAFlow before: aDAPointcut.
        DAFlow after: aDAPointcut.
    }.

    ↑ DAFlowPointcut
        context: (DAContextPointcut context: context)
        dependency: (DADependencyPointcut connectors: connectors).
```

It creates a context, two connectors and returns a flow pointcut containing all that. A connector is what aspects build directly on top. A connector is basically a wrapper around multiple links. Instead of taking nodes like a link, it understand pointcuts. It creates as many links as needed for the pointcut and tries to keep their installation consistent. Like a link it takes one meta object with one selector. The aspect uses multiple connectors,

sets the meta object to itself and the selector to the advice method. A connector can be regarded like an aspect with just on advice. It is a limited programmatic way to do aspects.

Dependencies are nothing else than connectors with the requirement that they are installed before the aspect. In this case, the connector is a special flow connector. On aspect installation, dependencies are just gathered over the whole pointcut structure and installed before the aspect.

DAFlow is just there to manage the counters per process. The second part is the context. This one serves as condition on the pointcut that actually checks on execution what the counter of the current process is, and if the condition is true or false. Looking closely at the implementation, one will notice that the counters are actually stored in the process object. This seems to break encapsulation a bit, but has the advantage that the counters are initialized/destroyed when the process is created/terminated. This removes the need to clean up by hand.

What a flow pointcut actually does is refer to the history of past join points and use it as a condition [Herz]. One issue with generic flow pointcuts is that one must think more carefully of how to use them. If for example, a pointcut with a condition is given as flow specification, it might happen that the condition is true on the before link, but false on the after link. That of course, leads to improper counting and can easily break things.

Chapter 5

Using DYNAMIC ASPECTS

This chapter addresses the user who wants to write aspects with DYNAMIC ASPECTS. It gives an introduction to defining and using aspects and shows what is possible to do with DYNAMIC ASPECTS.

5.1 The Aspect Browser

The aspect browser is a modification of the Squeak system browser that brings one simple enhancement for programming with aspects. It is registered in the world's open menu. As shown in figure 5.1, it adds a simple button to indicate the state of the aspect. The various colors and the label show if an aspect is currently selected, and if yes, if it is installed and enabled. The text changes depending on if an advice or pointcut is edited.

Pressing on the button opens a small menu for (re)installing and disabling the aspect, as shown in figure 5.2.

All other functionality of the aspect browser is the same as in the system browser. The button should update automatically every time the state of an aspect changes.

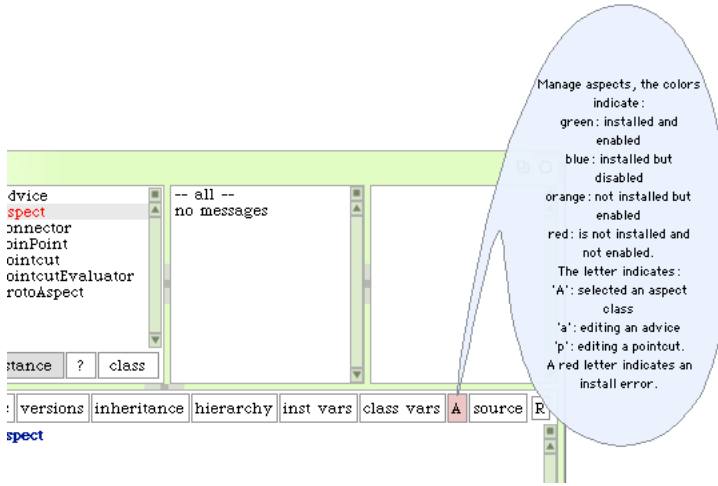


Figure 5.1: The aspect browser button

5.2 Defining aspects

5.2.1 Creating a new aspect

Creating new aspects is as simple as subclassing DAAspect. By default, a "*subaspect*" of DAAspect has no pointcuts and advices. Aspects can form arbitrary hierarchies as normal class hierarchies do. Like variables and methods an aspect inherits all the pointcuts and advices of its superclasses. To modify certain elements, the appropriate methods just need to be overwritten.

DAAspect has some special class methods in the options protocol that configure the behavior of an aspect.

#scope Must return a reification symbol that is valid for all pointcuts. For each new value of the reification a new object of the aspect is created. The advice is always executed on the corresponding aspect instance created for the specific value. By default nil is returned, which means to create only a single instance.

#cleanMetaObject This will clean up meta objects (aspect instances) when the aspect is reinstalled. By default, this is set to true. To preserve the state in aspect instances, we can set this to false. But it's likely that when reinstalling aspects (which can happen a lot if we make frequent changes to the aspect), there will be some old instances for scope values no longer used. So, manual cleanup of these instances is highly recommended.

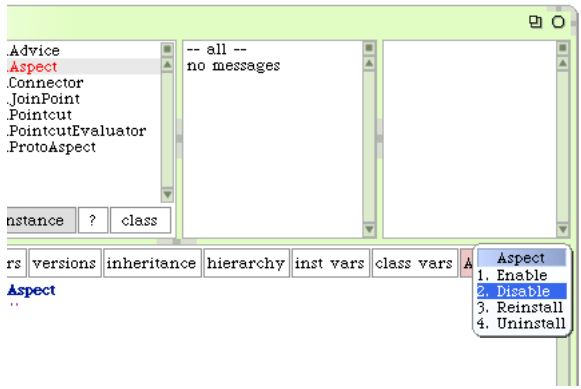


Figure 5.2: The aspect browser menu

#defaultDisabled This is by default true. It means, that initially an aspect is disabled. This can be handy if we create new aspects by subclassing or with yet incomplete pointcuts and advices. Later, we can enable and disable aspects, or even single advices, at any time.

#level This states the exclusive execution level at which the advice code is executed. Normal code is executed at the base level 0. Code called from advices is executed on the meta level, which is normally 1. In the case that code already executing on the meta level again reaches a location where an aspect is triggered, the level is increased again by 1. It is possible that infinite recursions happen when base and meta level execution is not carefully distinguished[Denk08b]. But with the level option this can be avoided. For example, to make an aspect only execute when it is called from the base level, but not when it is called from a meta level itself, set this to 1.

5.2.2 Adding pointcuts

Pointcuts are added on the class side of an aspect. Each method marked with the <**pointcut**> pragma specifies a pointcut. It then must return a valid pointcut structure, meaning an object of type DAPointcut. The name of a pointcut is the same as the name of the method. This is important since advices will connect to pointcuts by referencing this name.

In general, the same pointcut should not be used twice in compositions. Pointcuts are modifiable and changing one might affect various other pointcuts. No two pointcut methods should return pointcut structures that have

identical nodes. That means that pointcuts are never stored, but are always created dynamically from code.

For the three currently available composition operators, there exist binary and keyword messages that can be called on any pointcut object:

- AND: #and:, #&
- OR: #or:, #|
- NOT: #not

The result for AND and OR is a DACompositePointcut that can be handled just like any other pointcut.

Instantiating new pointcuts can either be done by using the specific pointcut classes directly, or by the convenience methods on the class side of DAPointcut. Some of them already return compositions of pointcuts. The global variable PC is just a reference to DAPointcut, that was introduced as a shortcut.

5.2.3 Adding advices

Advices are added on the instance side of an aspect. Each method that is marked with the <**advice: #pointcut control: #control**> pragma is an advice. This is the minimal version of the advice pragma. #pointcut must match the name of a valid pointcut method. The current possible values for #control are:

#before Execute the advice before the hooked code. After the advice the original code continues, returning the value of that code.

#after Execute the advice after the annotated statement. This also allows one to inspect the result returned by the original code. However, the final result will be that of the original code.

#instead Execute the advice instead of the annotated statement. The original code is not executed, unless explicitly requested by the advice. Multiple instead advices on the same code are currently not possible. The final return is the one of the advice.

#beforeafter Execute advice before and after the hooked code.

There are two more options to the advice pragma, arguments: and disabled:. The former allows one to specify reifications an advice requests in form of an array of symbols, like #(sender receiver). The advice then also must accept these as method arguments. The latter allows one to selectively disable certain advices with a value of true, or enable them with false or when the option is not present at all.

The two first parameters advice: and control: must be given and must be in this order. The optional parameters arguments: and disabled: may be missing and given in any order after the non-optional parameters.

5.2.4 Reifications

DYNAMIC ASPECTS makes use of reifications [Tant04] from GEPPETTO at several places. To be precise, the places are:

- Advice arguments
- Block arguments to pointcut conditions
- Aspect scope

The supported reifications by GEPPETTO can be queried with GPParameter allKeys. Some important ones are:

#argX Argument X of a method for $X = 1..4$

#arguments All arguments of a method

#object The object where the code is currently executing on

#class The object's class

#node The node in the reflective method from which the aspect was called

#operation The original statement the aspect was installed on, can for example be used to invoke the original operation in the case of an #instead control.

#sender The object that sent a message

#receiver The object that received a message

All these reifications (also called "parameters" in GEPPETTO) are identified by their corresponding symbol. Not all reifications make sense and can be used on all statements. For example, we cannot reify the arguments on a return statement, but we can do so for a block or a method. Each reification has its own class, usually named GP...Parameter. On the class side #key gives the symbol to identify the reification, and #nodes returns valid node types it can be used on.

GEPPETTO checks the use of reifications and complains about misuses with exceptions. DYNAMIC ASPECTS will just pass these exceptions along.

5.3 Installing and disabling aspects

There are various class methods in DAAspect that manage installation and updating of an aspect:

#enable Enables the aspect. If the aspect is not yet installed this also installs it. An enabled aspect calls reinstallSilently every time it is changed. This is the case, if:

- An advice is added/removed/changed
- A pointcut is added/removed/changed
- The aspect is created/removed/recompiled
- One of the special methods, e.g. #scope, is changed

#disable Disables the aspect. If it is installed it will also be uninstalled. A disabled aspect is not updated automatically. It can, however, be installed and uninstalled manually.

#install Installs the aspect; has no effect if the aspect is already installed.

#uninstall Uninstalls the aspect; has no effect if the aspect is already uninstalled.

#reinstall Simply calls first #uninstall then #install.

#...Silently Some methods also have silent versions, that catch any exception that may be thrown during execution, and keep silent about it towards the user. This is useful for automated tasks that are often run and which should not display an error to the user each time they fail.

#failure Inspect the failure that occurred in the latest silent installation attempt.

Normally, it should not happen that an aspect is in an inconsistent state. It should either be installed, meaning all its enabled advices have been installed successfully, or uninstalled, meaning it has no installed advices at all. If it happens that an aspect failed during installation, but also couldn't go back to the uninstalled state, it will stay inconsistent. Any further installation attempts will fail, and it must be cleaned up by hand. The only thing to do in such a case is to look at its connectors and make sure they all get cleanly uninstalled.

Currently aspects do not update if code they are installed on changes. This is currently the same behavior as GEPPETTO also has and is important to note for the user. *If code changes that has links installed on it, they are silently lost, and must explicitly be reinstalled on the new code.*

5.4 Programatically defining and using aspects

The way of programing aspects with classes and methods surely has its advantages. It allows the user to clearly structure his aspect code, just like in OOP. But it also forces the user to do so. Sometimes it might be desired to create aspects in a more lightweight fashion, without creating new classes. It is about the same when creating a block, which is anonymous, instead of adding the same code as a method to a class. Or when directly executing code from a workspace without the need to put it first in a well formed structure.

Something similar is possible with aspects. Aspects themselves use connectors, instances of DAConnector, on installation. A connector is a limited version of an aspect, that enriches the concept of a link in GEPPETTO. It understands how to install itself directly on a pointcut, and therefore it wraps multiple links. A connector can have just one meta object and one selector that it calls. We take again the transcript example from chapter 4:

Listing 5.1: Using connectors

```
c := DAConnector
    on: (PC class: TranscriptStream method: #show:)
    do: [ Transcript cr. ].
c install.
Transcript show: 'test'.
c uninstall.
```

This installs a small hook that adds a carriage return after each #show: on the transcript. The next step towards a lower level would be to use GEPPETTO directly.

5.5 Extending pointcuts

There are several possibilities when coming to the point where sophisticated pointcuts are needed. Pointcut expressions that have a rather local use, should always go directly to the pointcut methods in an aspect. If they should be shared among different aspects, they can be implemented in a top aspect class and then be inherited by subclasses. If the compositions have a more global scope, they can either be put into a globally accessible object, like the class side of a class, or as extension to DAPointcut.

Beyond that, tailored pointcut classes can be written. In consequence, also DAPointcutEvaluator must be adapted, so that the new types are correctly evaluated.

Chapter 6

Examples

This chapter applies what we have seen in the last chapter by the means of concrete code examples. It presents some simple and introductory examples, that show how to work with aspects.

6.1 The sushi store

In this section we will take the sushi store application from the Seaside examples to make some very basic demonstrations of how to use DYNAMIC ASPECTS. The sushi store is a simple web application that allows one to browse and order sushi in a store application running in the web browser.

6.1.1 A simple start: Modifying the title

A first very simple thing we want to do is to modify the title of the original application. This will show how the basics of DYNAMIC ASPECTS work, and will also show *that things really work*.

In a first step we create a new aspect for our title change. We call it DAStoreTitleExample and make it a subclass of DAAspect. On the class side we filter the #title method from WAStore in a pointcut also called title:

Listing 6.1: Pointcut for the store title

```
title

    <pointcut>
```

```
↑ PC class: WAStore method: #title.
```

On the instance side we define the associated advice that just returns an alternative title:

Listing 6.2: Advice for the store title

```
title

    <advice: #title control: #instead>

    ↑ 'sushiNet Pro'.
```

Compared to the original title:

Listing 6.3: Original store title

```
title

    ↑ 'sushiNet'
```

If we verify this in the browser we can see that the title changes from "SushiNet" to "SushiNet Pro". Note, that this happens without us needing to touch the source code of the original title method. A simple as this example may be, it's useful to show two other important things in the next two subsections.

6.1.2 Installing aspects on aspects

Since in DYNAMIC ASPECTS advices are normal methods, and they are the only place where added behavior is specified, we can easily install aspects on aspects, which actually means to install aspects on advices. We modify the example from the previous section, so that the already modified title is again altered. For that we first define an additional pointcut that is similar to the first, but now selects the advice method instead of the original title method:

Listing 6.4: Selecting the store title advice with a pointcut

```
titleAdvice

    <pointcut>

    ↑ PC class: self method: #title.
```

Then we again add an advice, like the first, but with a different name, so that we can distinguish it from the first:

Listing 6.5: Advicing the store title advice

```
title2

    <advice: #titleAdvice control: #instead>

    ↑ 'sushiNet Pro 2.0'.
```

So defining aspects on aspects is just as easy as defining them for any other method. Note, that when we disable the first advice the second also loses its effect, since it is installed directly on the first one and only has effect if this one is called.

6.1.3 Reusing the original operation

Thinking closely of what we have done in the last section by altering the title, we can notice that we actually overwrote the title although we just wanted to add something to the title. Doing the latter is also possible with the help of reifications as advice parameters. The final and maybe best solution could look like this:

Listing 6.6: The operation argument in advices

```
title: operation

    <advice: #title control: #instead arguments: #(operation)>

    ↑ operation value, ' Pro'.
```

We can request a reification of the original operation. Then, in the return we execute the operation, take the value, add some string and return the whole. If now the original title changes the advice does not need to be adapted. So, in fact, we eliminate duplicated code.

6.1.4 Cross-cutting code

We now want to use aspects to do something that really shows the cross-cutting characteristic of code, and how we can easily avoid it with aspects. An interesting thing would be to log all instance variable writes in the application. For this we first must filter all possible statements in the code that write to instance variables:

Listing 6.7: Filtering instance variable writes

```
fieldWrites
```

```
<pointcut>

↑ (PC className: 'WAStore*') & PC instanceWrite.
```

Luckily, there is already a predefined pointcut for this called instanceWrite. What it actually does can be seen in DAPointcut>>#instanceWrite:

Listing 6.8: Instance field write pointcut

```
instanceWrite

    ↑ self node: [
        :node |
        node isAssignment and: [ node variable isInstance. ].
    ].
```

It simply defines a block filter that selects all nodes in a reflective method that are assignments of instance variables.

In the advice, we just log every write to the transcript, but including the location in the code, so that we get some information where the variable is written. This is also the reason why we use the node reification.

Listing 6.9: Logging field writes with their location

```
logFieldWrites: node

    <advice: #fieldWrites control: #before arguments: #(node)>

    | method |

    method := node reflectiveMethod.
    Transcript show:
        'Writing field '',
        node variable name,
        '' in ',
        method methodClass name,
        '>>',
        method selector.
    Transcript cr.
```

If we now look at the transcript, shown in 6.1, we see a trace of writes. The blank lines were inserted manually after each click in the store application. For example, the third block shows what happens if we add an item to the cart.

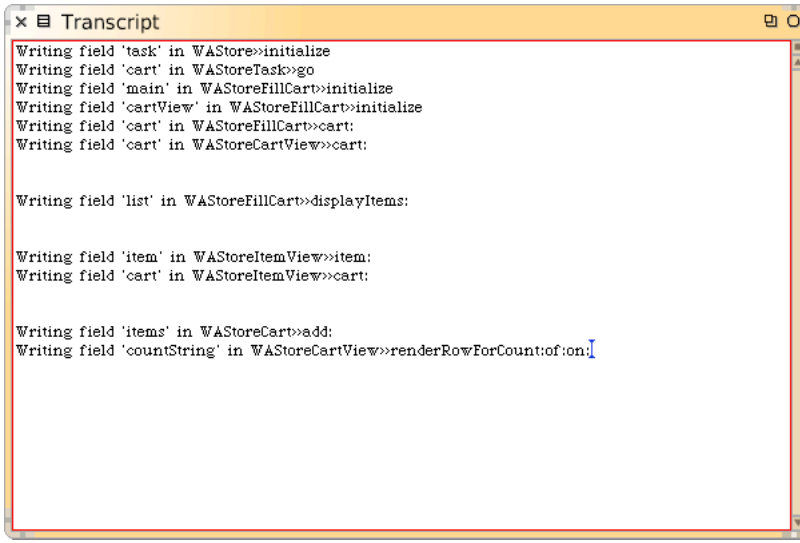


Figure 6.1: Field write log in the transcript

6.1.5 Modify the rendering

This is similar to the last example. It shows again the cross-cutting idea, and modifies the application's rendering. First, we select all render methods with a pointcut:

Listing 6.10: Selecting all rendering methods

```
rendering

    <pointcut>

    ↑ (PC className: 'WAStore*' methodName: #renderContentOn:).
```

Render methods in Seaside have the form renderContentOn: html. html is a canvas that can be used similar to a graphics object to produce HTML. We reify this argument to add a halo before each component:

Listing 6.11: Adding halos to the rendering

```
renderHalos: html object: component

    <advice: #rendering control: #before arguments: #(arg1 object)>

    html div id: 'halo'; with: [
        html small: [ html text: component asString. ].
    ].
```

Some additional code for modifying the style sheet is need, but not shown here. 6.2 shows a screenshot of how the modified rendering looks in the browser. Each component is preceded by a red bar showing the component name.

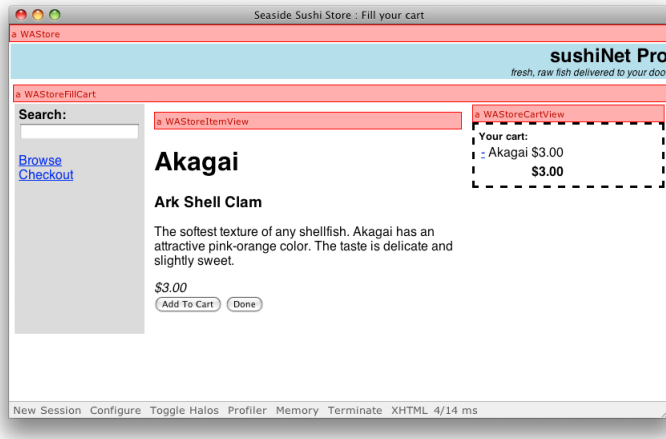


Figure 6.2: Rendering halos

6.1.6 Closing the store

Finally, we want to include a conditional factor in our examples. We will define a simple context DAStoreClosedContext that should state the condition when the store is closed. Its isActive message looks like:

Listing 6.12: Store closed condition

```
isActive

    ↑ Time now seconds even.
```

In reality, we would make a time specification that makes more sense. But this here ensures that we will quickly see the different behavior for both situations, when the context is active and when it is not, since it changes its state every second.

The pointcut selects the entrance of the store application:

Listing 6.13: Store application entrance

```
entrance
```

```
<pointcut>

↑ (PC class: WAStoreTask method: #go) & (PC context:
  DAStoreClosedContext).
```

The composition with the context assures that our advice is only activated when the store is closed. In the advice we cut the original application flow and render a closed message instead:

Listing 6.14: Store closed message

```
closedEntrance: task

    <advice: #entrance control: #instead arguments: #(object)>

    task call: DAStoreClosedMessage new.
```

If we hit a bunch of times on the "New Session" button, we will see that every even second the closed message will be rendered instead of the original store front page, which then looks like in figure 6.3.

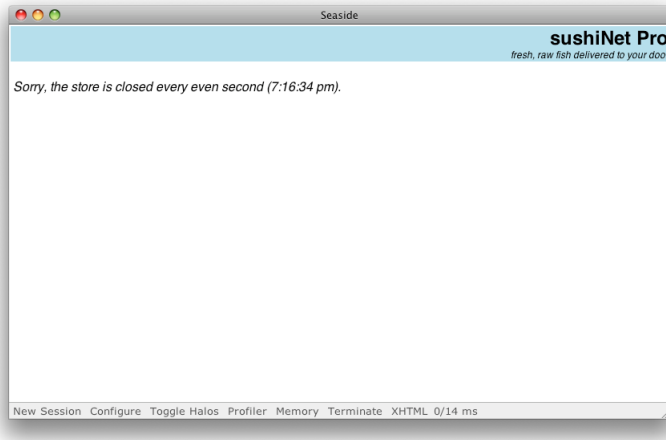


Figure 6.3: Closed store message

6.2 Private methods

One thing that is missing in Smalltalk compared to many other languages is access modifiers on methods. With aspects we can implement this feature to some degree. We think of some methods in a class that we want to

protect from being called from the outside, meaning not within the class itself. Smalltalk does not provide any protection mechanism by itself for that. Methods can be called from anywhere.

We take the simple case of one public method and one private method. The public method just forwards to the private method, which is fine, but calling the private method directly should be forbidden. We first create a new aspect DAPrivateMethodExample and define the private/public methods directly in the aspect, for the ease of use:

Listing 6.15: Public and private method

```
private

    "Should only be sent within the class."

    Transcript show: 'private'; cr.


public

    "Can be sent from anywhere."

    ↑ self private.
```

To capture the case where private is called from outside we will make use of a flow pointcut. We define pointcuts for selecting both methods and for the case of violating the access:

Listing 6.16: Pointcuts for capturing private method access violation

```
privateMethod

    ↑ PC class: self class method: #private.


publicMethod

    ↑ PC class: self class method: #public.


violatingPrivate

    <pointcut>

    ↑ self privateMethod & (PC flow: self publicMethod min: 1) not.
```

We limit the example to just one private and one public method. We could also declare multiple methods as public, but we will keep it simple

here. Actually, this allows us to define an arbitrary set of methods that are allowed to call #private, every other method is then automatically forbidden to do so. We are not limited to the scope of a class; we can define our own scope depending on what methods we select.

In the pointcut method we select the private method and combine it with a flow pointcut. The flow is then like a condition. It is true when the execution is within the public method. Since we want to express the case when access is *violated* we negate the flow condition, meaning it is true whenever the execution is *not* within the public method. Note, that this is different from defining the flow on every method that is not violating the access. That would result in many more methods, and to manage the flow for all of them would be much more expensive.

The advice should do something to state the violation of access:

Listing 6.17: Advice for method access violation

```
guardPrivate

    <advice: #violatingPrivate control: #before>

    self error: 'calling private method from outside'.
```

It executes before the private method and throws an exception informing that calling the private method directly is not allowed, as shown in figure 6.4.

Figure 6.4: Access violation error

Chapter 7

Conclusion

Here, we will finally revisit DYNAMIC ASPECTS as a competitive AOP implementation and see what problems it solves from the ones we have stated in the beginning. DYNAMIC ASPECTS is not a completely finalized project. There are small things, that could greatly enhance the current implementation, and bigger ones, that are clearly out of the scope of this work and targeted for future work.

7.1 Revisiting DYNAMIC ASPECTS

When comparing DYNAMIC ASPECTS to other implementations, there are some clear benefits in this implementation:

No special language constructs Extending the language syntax and semantics, adding new compiler tools, big adaptations to the IDE; all of that is just not needed. New concepts are mapped to existing ones. This proves again how flexible Smalltalk is, and how easy it is to build upon existing things. All in all, this makes DYNAMIC ASPECTS easy to learn and use.

Simple weaving Although not yet perfect, the weaving process is automatic and also dynamic in the sense that aspect functionality can be enabled/disabled on the fly on running code.

Modularity and reusability The decision to keep the implementation model of AOP very close to OOP gives the system great modularity. One of the most powerful techniques in OOP to reuse code is inheritance, and that is just as true for aspects. Moreover the freedom to combine pointcuts and advices allows great modularity and subsequent

extensions.

Contexts DYNAMIC ASPECTS provides an class structure for contexts and incorporates them in the pointcut language. This allows one to express complex conditions in a better way, and gives new possibilities as shown with the implementation of flow pointcuts as an abstraction from control flow pointcuts.

Dynamics Aspects can be installed/uninstalled and enabled/disabled on the fly. Aspects form a kind of pluggable behavior that can quickly and easily be unplugged on request, or silently be plugged into already running and uninterruptible applications.

Some advantages come clearly from the underlying Squeak and REFLECTIVITY platform, but nevertheless consequently using them in DYNAMIC ASPECTS gives it some truly unique features.

If we look at DYNAMIC ASPECTS in the context of our initial problem statement, we can now say that we have a lightweight AOP implementation that does not introduce new language or tools, yet it is powerful and even surpasses the functionality of ASPECTJ in some points with contexts and flow pointcuts.

7.2 Enhancements

7.2.1 Adapting dynamically to code changes

A lot of AOP implementations have a very static weaving process. Aspects are applied after the code is compiled. Each change of the original code requires a complete repetition of this process. The dynamic nature of Squeak is not only a big advantage, but can in some cases also make high expectations on its applications.

Adapting aspects automatically to code changes puts some challenges on the implementation:

- When adding new code, at least for some pointcuts, the only way to find out if the new code is affected by them is to re-evaluate them. There should be a way to only do some checks instead of needing to fully evaluate them again. Like this some pointcuts can be completely excluded or only partially evaluated. Producing a lot of additional computations at every code change can significantly reduce Squeak's performance, which will ultimately influence the usability in such a highly integrated and interactive system.

- Modifying code can completely or partially break the installation of aspects on that code. There need to be some strategies on how to inform the user about that, being aware that one change can cause a lot of failures, and what to do with such aspects. Aspects that are only partially installed can result in a totally different behavior, and also foreseeing the impacts of aspects on new code can be difficult.

7.2.2 Filtering with true regular expressions

Although filtering directly on elements with filter blocks is much more powerful than just filtering on the name, the latter still has an important role. Selecting elements by name when actually having certain requirements in mind that are not directly related to the name, can be fragile in that the filter will easily break when code changes. But mostly a user chooses the names in a way so that they reflect the meaning of the classes and methods. And that is why filtering by name is still very important.

The current implementation uses only very primitive regular expressions, the ones that are directly supported in strings. For true regular expressions the pointcut evaluation turned out to be very slow. To support full regular expressions with decent performance the filtering mechanism may need to be enhanced. Currently, when building up complicated pointcut compositions, a lot of different filters can get combined. Through the generic evaluation of pointcuts there might be a lot of unnecessary filtering. For example, duplicated filters are not eliminated and some filters are evaluated multiple times, even though they yield exactly the same.

A good package for true regular expressions is "Vassili's Regex" that is available from Squeak's package loader. With some optimizations to DYNAMIC ASPECTS this should be able to replace the currently very limited regular expressions.

7.2.3 Caching for performance improvements

Join points and pointcuts do already work with caches, and both support a reset message. The original idea was, that pointcuts can be changed without needing to re-evaluate all of their join points, and that join points can be re-evaluated when code changes without the need to recreate them.

Changing pointcuts is something that is maybe not so much of use for aspects directly. In aspects, pointcuts are defined directly as code in methods and the aspect automatically instantiates them when needed. But pointcuts can also be used out of aspects. For example, when working directly with connectors. In DAPointcut some more work is needed so that the already

implemented and used caches are properly discarded. Currently only the node that is asked to evaluate will create an evaluator and this one stores all evaluated join points. This could be extended, so that each pointcut caches its corresponding join points or subtrees. On change all caches upwards to the top pointcut node must be invalidated and recreated on next use.

Currently an aspect re-evaluates all its pointcut methods on an update, causing all old pointcut objects to be discarded and the new ones being required to evaluate first. An alternative would be, that an aspect holds the pointcut instances in a collection and on a specific update of a pointcut re-creates just that single pointcut. The update methods already support the notification of a single pointcut selector. The difficulty is that a pointcut update can only be noticed if the method marked with the pointcut pragma is changed. If it collects pointcuts from code outside itself, this will not work.

Join points still have a descriptive characteristic like pointcuts as well. This means, that they do not directly reference elements, but hold descriptions of it. When code changes, and that happens often in such dynamic systems as Squeak is one, join points just need to be re-revaluated. The challenge here is, that not only existing code changes, but also new code can be added. The former can easily be tracked, since all installed links know their connector, and each connector knows its pointcut and aspect. But in the latter case, the only solution to check if a join point matches new code is to re-evaluate it. This results in re-evaluating all join points over all aspects for any code change, which is hardly ideal at the moment.

7.3 Future work

7.3.1 Contextual aspects

DYNAMIC ASPECTS provides stateful, composable contexts and context pointcuts. However, some improvements can still be made. Real *context-aware aspects* should be implemented [Tant05, Cott]. Some things that are currently missing in DYNAMIC ASPECTS are:

- Parameterized contexts. This can for example be useful to have contexts with thresholds. Currently a context just supports the message isActive, which makes it rather inflexible.
- The possibility to define dependencies with pointcuts not only on current context states, but also depending on what contexts an application has been in earlier.

Also, there are valuable ideas in the field of Context-oriented programming [Hirs08] that should be thought of in combination with AOP.

7.3.2 Pointcuts as general query language for reflection

DYNAMIC ASPECTS's pointcut language is designed specifically for the needs of aspects. But some parts of it could actually also be used in another context. The filtering part could be changed into a generic query interface for reflection. This should be coordinated with the already existing reflective interface in classes, and especially in reflective methods. Then this could be integrated directly into REFLECTIVITY, and on top of it DYNAMIC ASPECTS could extend this language with aspect-specific things, like conditions, contexts, and others.

The benefit is a more powerful reflective query interface that can be used by various other parts in Squeak. This surely requires some changes on the current model of pointcuts.

Appendix A

Quickstart

DYNAMIC ASPECTS *requires* a Squeak image with REFLECTIVITY installed. The standard images downloadable from Squeak *will not work*.

REFLECTIVITY is availabe from the REFLECTIVITY website[1], either as a prepared image, or as instructions how to create one from the standard image yourself.

All DYNAMIC ASPECTS packages are available from SqueakSource[2] and can be loaded with Monticello.

As of REFLECTIVITY version 0.16, there are some fixes in DAReflectivityFixes−XXX.mcz that should be loaded first. Currently DYNAMIC ASPECTS does not work with REFLECTIVITY 0.17, hopefully it will with the next version which should also include these fixes.

Installation:

1. Download or prepare a REFLECTIVITY 0.16 image and open it
2. Load DAReflectivityFixes−0.16.mcz
3. Load the latest DA−as.XXX.mcz
4. Optionally load the latest DAExamples−as.XXX.mcz

The store example requires Seaside (available from the integrated package browser in Squeak) and the the sushi store example (available in the "Seaside Examples" project also on SqueakSource) to run properly, but the examples will install without these packages available.

[1] http://scg.unibe.ch/Research/Reflectivity/
[2] http://www.squeaksource.com/DynamicAspects/

Aspects are normally disabled on first initialization. The can be enabled by sending enable to an aspect class. The following will enable all aspects:

```
DAAspect allSubclasses do: [ :a | a enable. ].
```

Appendix B

Squeak core classes

When dealing with reflection and meta classes one often comes across the core classes like Behavior or Metaclass. Especially when coding the core aspect classes, which are basically like other classes in Squeak but still a bit different, I found it very useful to have *"the big picture"* of the core classes and its relations, namely inheritance and instantiation, in front of me. Finally, making the picture myself, I got a bit more insight which helped me to remember it better.

Here are some points of figure B.1 to think about:

- Inheritance forms a spiral.
- Instantiation forms a star.
- `Smalltalk allClasses select: [ :c | c class includesBehavior: c. ].`
  "⟶ an OrderedCollection(Behavior Class ClassDescription Object ProtoObject)"
  Meaning, some classes have instances which are then also among their superclasses. This forms some kind of loop.
- The bootstrapping question: How is it possible that every class eventually inherits from ProtoObject, but this one first needs its meta class to be present to be instantiated from.
- Every class has a superclass *except* ProtoObject, whose superclass is nil.
- MyClass indicates where the ordinary programmer's code usually attaches to the core system.

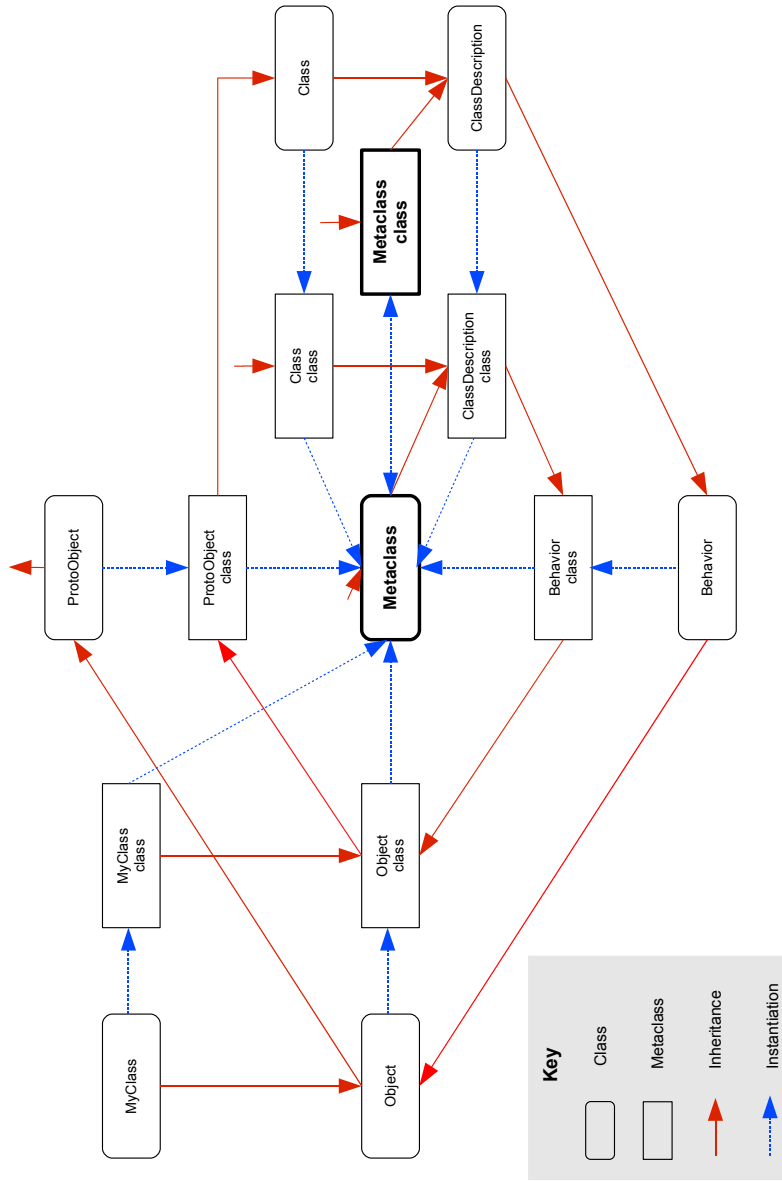


Figure B.1: Relations of core classes in Squeak 3.9

Appendix C

Glossary

Programming paradigm The fundamental techniques that are used to program. Most languages allow and prohibit certain techniques to force a specific programming paradigm, some languages also support multiple paradigms. Often this is also referred to as programming style or methodology.

AOP Aspect-oriented Programming. A programming paradigm, often based on object-oriented programming (OOP), but here the units of modularization are aspects instead of objects.

COP Context-oriented Programming. Yet another member in the family of *OP paradigms. COP focuses on writing code that behaves differently depending on what context the program is currently in. A context is usually not something abstract but rather very concrete to the domain of the program, e.g. the authorization level the currently executing user has, or if the program is run within a development or production environment.

Cross-cutting code Code that spreads over various locations, but regarding it's functionality, should rather be packed in one or a few locations. The locations may be whatever the units of organizing code in the underlying programming style are, e.g. in OOP these are usually classes.

AOSD Aspect-oriented software development. AOP plus all sorts of things that come along when using it as part of the original software development process, like requirements engineering, modeling, refactoring, maintaining, IDE support and lots more, all focused on what is special for aspects.

Pragma Pragmas are compiler directives to provide additional information

about source code. In Squeak pragmas have the form <**keyword**> or <**keyword1: arg1 keyword2: arg2 ...**>. They are set in the beginning of a method body and are scoped on a per-method level. The compiler transforms them into method properties that can be queried on the method objects.

Reflective method In REFLECTIVITY there is the concept of having for each compiled method a reflective method. This is a reflective representation of the method presented as a tree of nodes, giving a very detailed object structure for methods. It is similar to an abstract syntax tree, but in addition it allows search queries and annotating the original code. The compiled method is generated from the reflective one, and both are always kept in synchronization.

Annotations Annotating means adding additional information to the code, either visible by a special syntax in the source code, or invisible by annotating for example a reflective method. Annotations are evaluated by compiler plugins that can perform various operations depending on the annotations.

Sub-method level reflection Reflection that works on the sub-method level. The system can reflect entities inside methods, e.g. single statements.

Reification In the context of reflection reifying means generating the reflective representation of some element. Since reification can be very costly, partial reflection, where only the parts currently needed are reified, is a good choice.

AST An abstract syntax tree is a syntactical representation of the source code as a tree. This is usually the first step of transformation after having parsed the source code and before compiling it. Further representations, like for example the intermediate representation (IR), may follow, when finally the code is compiled into bytecode or machine code.

Hook A hook is a piece in the compiled code that was inserted by some other means than the classical way of writing source code and compiling it. In DYNAMIC ASPECTS each aspect finally results in one or more hooks. These are the concrete places in the executing code that make the calls to the aspect and it's advices.

Pointcut, Join point, Join point shadow, Advice, Weaving, Aspect See **section** 2.2 (p. 8) for explanation of these terms.

REFLECTIVITY, PERSEPHONE, GEPPETTO PERSEPHONE and GEPPETTO are parts of the REFLECTIVITY system, an enhanced Squeak Smalltalk image that supports advanced reflection. For more documentation see **section** 2.2 (p. 8) and the references there.

Appendix D

Document Version

This states the latest subversion revision and date when any `.tex`-file of this document was changed.

SVN Revision 22676
SVN Date November 11, 2008

For more information on using subversion keywords in LaTeX-documents see the `svn-multi` package in the CTAN[1] archive.

[1] http://www.ctan.org/tex-archive/macros/latex/contrib/svn-multi/

List of Figures

Listings

List of Tables

Bibliography

[Denk08a] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, Switzerland, May 2008.

[Denk08b] Marcus Denker, Mathieu Suen and Stéphane Ducasse. *The Meta in Meta-object Architectures*. Software Composition Group, University of Bern, Switzerland.

[Denk07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard and Philippe Marschall. *Sub-Method Reflection*. Journal of Object Technology, vol. 6, no. 9, pp. 231–251, 2007.

[Röth08] David Röthlisberger, Marcus Denker and Éric Tanter. *Unanticipated Partial Behavioral Reflection: Adapting Applications at Runtime*. Journal of Computer Languages, Systems and Structures, Elsevier, vol. 24, no. 2-3, pp. 46–65, July 2008.

[Röth05] David Röthlisberger. *Geppetto: Enhancing Smalltalk's Reflective Capabilities with Unanticipated Reflection*. Master thesis, University of Bern, Switzerland, 2005.

[Mars06] Philippe Michael Marschall. *Taking Smalltalk Reflection to the sub-method Level*. Master thesis, University of Bern, Switzerland, 2006.

[Vand] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, Viviane Jonckers. *Adaptive Programming in JAsCo*. Vrije Universiteit Brussel, Belgium.

[Kicz97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. *Aspect-Oriented Programming*. Xerox Palo Alto Research Center, USA, 1997.

[Hane] Stefan Hanenberg, Pascal Costanza. *Connecting Aspects in AspectJ: Strategies vs. Patterns*. University of Essen and University of Bonn, Germany.

[Tant05] Éric Tanter, Kris Gybels, Marcus Denker and Alexandre Bergel. *Context-aware Aspects*. Center for Web Research/DCC, University of Chile, Santiago, Chile; PROG Lab, Vrije Universiteit Brussel, Belgium; Software Composition Group, University of Bern, Switzerland; Distributed Systems Group Trinity College, Ireland.

[Tant04] Éric Tanter, jacques Noyé, Denis Caromel, Pierre Cointe. *Partial Behavioral Reection: Spatial and Temporal Selection of Reification*. University of Chile, Chile, École des Mines de Nantes, France, Campus Universitaire de Beaulieu, France, Université de Nice, France, 2004.

[Tant03] Éric Tanter, Jacques Noyé, Denis Caromel and Pierre Cointe. *Partial Behavioral Reflection: Spatial and Temporal Selection of Reification*. Proceedings of OOPSLA '03, ACM SIGPLAN Notices, November, pp. 27–46, 2003.

[Tant01] Éric Tanter, Noury M. N. Bouraqadi-Saâdani and Jacques Noyé. *Reflex — Towards an Open Reflective Extension of Java*. Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, LNCS 2192, Springer Verlag, pp. 25–43, 2001.

[Rash04] A. Rashid, A. Moreira and B. Tekinerdogan. *Editorial: Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*. IEE Proceedings — Software Special Issue, vol. 151, no. 4, pp. 153–156, 2004.

[Hirs08] Robert Hirschfeld, Pascal Costanza and Oscar Nierstrasz. *Context-oriented Programming*. Journal of Object Technology, vol. 7, no. 3, pp. 125–151, March–April 2008.

[Hirs03] Robert Hirschfeld. *AspectS — Aspect-Oriented Programming with Squeak*. Objects, Components, Architectures, Services, and Applications for a Networked World, LNCS 2591, Springer Verlag, pp. 26–232, 2003.

[Hirs02] Robert Hirschfeld. *PerspectiveS — AspectS with Context*. In *OOPSLA 2002 Workshop on Engineering Context-Aware Object-Oriented Systems and Environments*, Seattle, WA, United States, 2002.

[Rodr] Leonardo Rodríguez, Éric Tanter and Jacques Noyé. *Supporting Dynamic Crosscutting with Partial Behavioral Reflection: a Case Study*. Universidad de la República, Uruguay, Universidad de Chile, Chile, Ecole des Mines de Nantes, France, INRIA Rennes, France.

[Herz] Charlotte Herzeel, Kris Gybels, Pascal Costanza, Coen De Roover, and Theo DHondt. *Forward Chaining in HALO — An Implementation Strategy for History-based Logic Pointcuts*. Vrije Universiteit Brussel, Belgium.

[Cott] Thomas Cottenier, Aswin van den Berg and Tzilla Elrad. *Stateful Aspects: the Case for Aspect-Oriented Modeling*. Illinois Institute of Technology, Motorola Labs.

[AspJ] *AspectJ — Crosscutting objects for better modularity*. http://www.eclise.org/aspectj/.