Patterns and an Antiidiom for Aspect Oriented Programming (AOP)

Arno Schmidmeier

AspectSoft
Lohweg 9,
91217 Hersbruck,
Germany
Arno@aspectsoft.de

Abstract.
There are many different options when using AOP tools or languages to solve similar problems. Modern AOP languages offer new features and design ideas, through the combination of object and aspect oriented features. Patterns are therefore required to guide AOP developers and architects around the common pitfalls and unleash the power of Aspect Oriented Programming.
This paper describes three patterns and one common antiidiom. The first pattern PROCEED OBJECT deals with the problem how to delegate the execution of an around advice. The second pattern EAGER POINTCUT DECISION deals with the problem, how to avoid or reduce the performance hits of advices, which may be enabled or disabled at runtime. The third pattern notbelow describes a pattern, which handles the forces which the antiidioms NOTWITHIN should often resolve.

Introduction

Objects have been quite successful in the past to modularize and organize the implementation of the core requirements. Object Oriented technology created new architectures and solved problems. It proved also quite successful in creating new architectures. However due to the increasing complexity of modern programs non business concerns got more and more important. Typical examples include transaction handling, persistency, tracing, logging, and contract checking, etc. Object technology failed to handle these additional concerns in a modular way. Aspect oriented programming (AOP) proved quite successful in modularizing these concerns [5][6]. However AOP is quite a new programming paradigm, which has not jet entered mainstream software development. This paper contains a short introduction to the features of AspectJ in order to support the reader, who may not jet be familiar with AOP.
Most AOP based applications are currently built on top of an existing Object Oriented -code base. Sometimes an existing Object Oriented or component framework is enhanced by some pluggable or optional aspects, and sometimes they are even evolved to an Aspect Oriented

Framework. This paper describes three patterns and one idiom, which provided quite useful in these scenarios.

- The PROCEED OBJECT pattern is often used to convert a proceed call in an advice to an object. This object can participate in "classical" object oriented approaches, idioms and patterns like the famous command pattern[1].
- The EAGER POINTCUT DECISION pattern can be used to eliminate potential performance bottlenecks, typically caused by optional advices, which might be turned on or off, often even after the weaving took place. (e.g. for a configurable logging and tracing framework),.
- The NOTWITHIN antiidiom describes an often used and suggested idiom to ensure that a very unspecific advice is not applied in infinite recursion.
- And the third pattern NOTBELOW describes, how to ensure that an advice is executed only once in a call stack, even if aspect inheritance is used to decide where an aspect should be applicable. This pattern provides a usable solution for the antiidiom NOTWITHIN.

Although all the samples and code fragments are written in AspectJ and the names are influenced deeply by AspectJ terms, the patterns and idioms are useful outside AspectJ.

All patterns are applicable to other AOP Frameworks (e.g. aspektwerks[10], JBoss-AOP[11], Nanning[12]) and languages like Sally ([13]). The PROCEED OBJECT pattern is even quite useful in plain old java programs and architectures, which rely heavily on dynamic interceptors, a feature in the J2SE since its Version 1.3 [18].

The antiidiom itself can be applied in all AOP Frameworks and languages, which offer a capability to advice its own code. However I have seen this antiidiom currently only in AspectJ programs, but I expect, that this antiidiom will also appear often in other AOP languages as soon as they mature and gain a wider audience.

Applications of the PROCEED PATTERN are also described in literature. Two [8][9] of the currently published AspectJ books describe the applications build on this pattern. I observed all patterns in the wild from where I minded them. All these idioms and patterns have been extracted from commercial AspectJ projects in which I was involved.

AspectJ did have a great influence on my choice for the names of the participants and the patterns. However I do not consider it as a drawback, for the following reasons:

- Most current AOP adopters have at least a basic understanding of AspectJ and of its terms,
- Quite a lot of AOP languages and Aspect Oriented Software Development (AOSD) tools reuse also the AspectJ terms.
- Quite a lot of AOP papers use AspectJ-Terms.

Even more, I consider the AspectJ based naming a plus in practical commercial projects. Most commercial AOP projects I am aware of are based on AspectJ. If new developers are added on this project, they have to learn the new language, its terms and its ideas. A pattern vocabulary quite close to the AspectJ language benefits from AspectJ learning and training and these patterns and their names teach new AspectJ adopters patterns and samples for easier adoption. As mentioned above a short introduction into the ideas and AspectJ terminology can be found in the Appendix to this paper to offer an easy transfer of these pattern to other AOP platforms and languages.

Proceed Object

Motivation

Quite a lot of architectures and programs use the command pattern. The command pattern [1] is the base for several modern GUI frameworks. Single threaded GUI-Frameworks like Java Swing [14] or SWT [15] are very often based on this concept. They use a command pattern to serialize all GUI-invocations by dispatching all invocations in a task chain. Only the GUI update thread can executes each task.

The Java Swing Framework requires, that all calls, which update the GUI, must be performed inside the GUI update thread. Code, which may execute in other threads, must be encapsulated inside a `Runable` Object. This `Runable` Object must be passed over to the GUI update thread either with the invokeAndWait or invokelater- method.

Following simple call

```
tableModel.setValueAt(value, 0, 0);
```

must be transformed to:

```
EventQueue.invokeAndWait(new Runable(){
      public void run(){
            tableModel.setValueAt(value, 0, 0);
      }
}
```

Correct Java Swing-GUI code contains quite a lot of crosscutting code fragments, which deal with synchronization of the GUI-Code. Aspect oriented programs often want to move these crosscutting code fragments to aspects. Selecting the relevant code fragments with aspects is quite easy. However we need to hand over the proceed call to a different thread. Unfortunatly AspectJ does not have a proceed method in its thisJoinPoint object. (Quite a lot of other AOP frameworks do not have it either.)

A very related motivation is the desire to advice the proceed call. Assume you have an aspect which implements a try- reconnect-retry algorithm around important business code. Assume also that the algorithm and its results should be traced and benchmarked. The natural idea is to advice the code of the advice, but AspectJ has no pointcut for the proceed call.

* * * * *

Problem

How can you use the proceed call of an advice in an object oriented context?

How can an application execute the proceed call in a different thread, or can pass the proceed call around.? How can you advice then proceed call?

* * * * *

Solution

Wrap the proceed call in an object (PROCEED OBJECT), which implements a given interface (the PROCEED INTERFACE).
All necessary information for the call to proceed is now stored in the context of the PROCEED OBJECT. The proceed call can be executed by invoking the PROCEED METHOD of the PROCEED OBJECT. The PROCEED OBJECT is a plain old java object (POJO) which implements the PROCEED INTERFACE. Therefore, it can be passed around, stored, advised and invoked like any other (POJO). It is in the responsibility of the capture advice to place the PROCEED OBJECT in a well known location, handle it over to an implementation of a command pattern or start its execution by invoking the PROCEED METHOD.

Examples

AspectJ can be easily used to modularize the crosscutting code fragments of the sample code of the motivation. Following code fragment in an aspect extracts this crosscutting code and serves as a marvelous example for the need of an inner class.

```
void around(): swingMethods(){
      EventQueue.invokeLater(new Runable(){
                  public void run(){
                        proceed();
                  }

            }
      )
}
```

Discussion

The PROCEED OBJECT has often a very simple interface, which often contains only the method signature of the PROCEED METHOD. Quite a lot of implementation of this pattern reuse very simple but widely known interfaces like the `java.lang.Runnable` as a PROCEED INTERFACE. The PROCEED INTERFACE is quite often not a real interface but an abstract class, where the PROCEED METHOD is the implementation of the abstract method. I observed this design especially in the cases where this pattern is combined with other patterns like the famous command pattern.
It is sensible to keep the PROCEED METHOD as simple as possible. It is most useful, if the PROCEED METHOD contains only the proceed call. An advice of the PROCEED METHOD is then equal to an advice of the PROCEED METHOD. This fact is helpful, if this pattern should be combined with other aspect patterns or if the aspect and especially the proceed call should be adviced during the software evolution.
If the PROCEED METHOD is used to group the joinpoints of the creating advice, the used PROCEED INTERFACE specifies often a getter and a setter methods for a joinpoint variable. These methods are used to set and retrieve the joinpoint of the creating advice. This information can´t be accessed otherwise from a second aspect, which advices the PROCEED METHOD.
This pattern is currently often used to refactor crosscutting code of the command pattern. We saw that usage in the first sample fragment. These scenarios are currently the dominant usages of this pattern. Also most of the

current uses of this pattern fall into this category. Whenever I present the sample to AspectJ newbies, the newbies tend to classify it as an interesting implementation of the command pattern. But the pattern stands for its own, because it is also used for other scenarios e.g.:

- It is often used to implement or emulate extensible pointcuts.
- It is often used to implement a workaround for a potentially missing pointcut for the proceed statement.
- It is often used to combine two aspect oriented patterns, which both needs the extends relationship of an aspect.
- It is often used to store and save the pointcut (and maybe reuse later of course)
- I saw several usages, where this pattern was only used to extract code blocks from a very complicated advice.
- This pattern is often used to implement the wormhole [7] pattern, if an percflow or percflow below aspect is not applicable. (e.g. The used aspect language or framework does not provide it, or some of it contracts do not fit).

Additionally the PROCEED OBJECT pattern can be only used to implement the task object of the command pattern. I found this currently popular combination of the command pattern and the proceed object mostly where aspect oriented languages or frameworks cooperate with legacy object oriented architectures, programs, APIs or libraries. Aspects are used to fulfill the architectural contracts of the design patterns. The Use of the command pattern changes in “pure” aspect oriented programs. Commands tend to be much more complex. (e.g. Implementing complex undo redo functionality) Simple PROCEED OBJECTS can not be used to implement them easily.

The PROCEED OBJECT often holds a reference to the invoked objects, and to the other involved objects at the pointcut. (e.g. through a Joinpoint Object). A garbage collection can not collect these objects until the PROCEED OBJECT itself is freed. So ensure that the lifecycle of the PROCEED OBJECT is shorter or equal than the involved objects. Or ensure that the memory overhead of the delayed garbage collection is acceptable.

Implementation Notes

Some AOP languages and AOP frameworks have good support for the PROCEED OBJECT. The object, which represents a joinpoint instance has a PROCEED METHOD. In these languages it is sufficient to wrap the joinpoint object with the PROCEED OBJECT.

However there are other AOP languages like AspectJ, whose joinpoint objects of an around advice do not have an API for the invocation of the proceed statement. The reflective information of the joinpoint object and the power of the refection API of the Java platform offer only limited support to implement a proceed call outside the advice. The joinpoint object does offer all necessary information required to select the class, the method or the attributes and the object, which was used at the joinpoint. So all information is available to invoke the method or access the attribute hidden by the advice at the joinpoint with the reflective API. However all advices which are dominated by the around-advice would be bypassed. This side effect is neither possible nor desired.

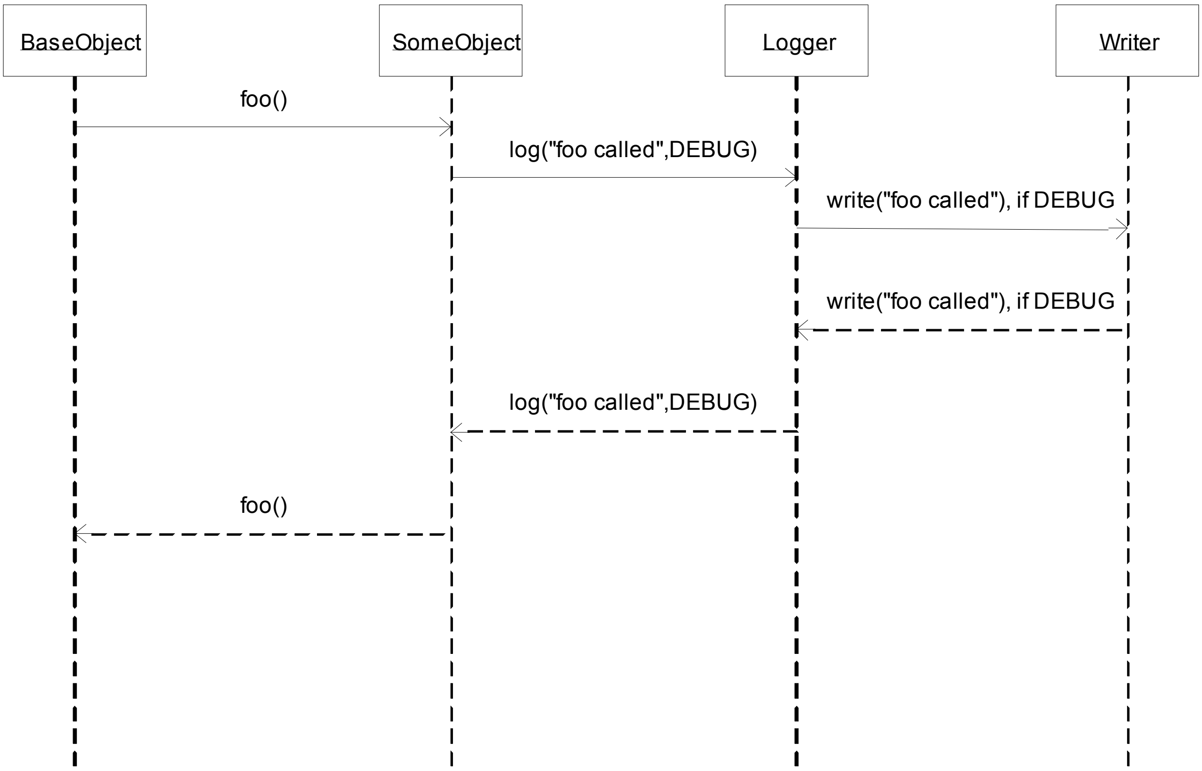
The use of an inner class inside the advice overcomes this problem. A method in this inner class can contain a call to proceed. This inner class can implement the PROCEED INTERFACE. As soon as the advice is executed, a new instance of the PROCEED OBJECT is created. The instance of the inner class is the PROCEED OBJECT.

Eager Pointcut Decision

Motivation

Current Implementations of Aspect Oriented Languages provide sufficient efficiency for the combinations of the advice code and the base code. An advice is implemented as a method in the aspect class in AspectJ. The weaving is implemented by a simple function call. More Details about the implementation an its efficiency of in AspectJ can be found in [17]. This efficiency is required for quite a lot of typical applications of AOP. Especially, if Aspects are used to advice plenty of points at a very fine grained level, e.g. at a huge number of intra method level concerns. Typical examples are very fine-grained tracing, variable and contract conformance validation. The critical performance impact is in such cases the execution time of the advice body.

Often typical non-functional concerns like tracing, pre- and postcondition checking, compete with performance. Typical libraries for these non functional concerns have a possibility to turn them on or off. If they are turned off, the performance impact is one or two function calls, plus the creation and handover of the parameters. In a “traditional” manual approach the creation and handover of parameters is in general very cheap, because most of the parameters are simple handcrafted constants. Following diagram illustrates the typical call stack of a manual use of a logging Framework:

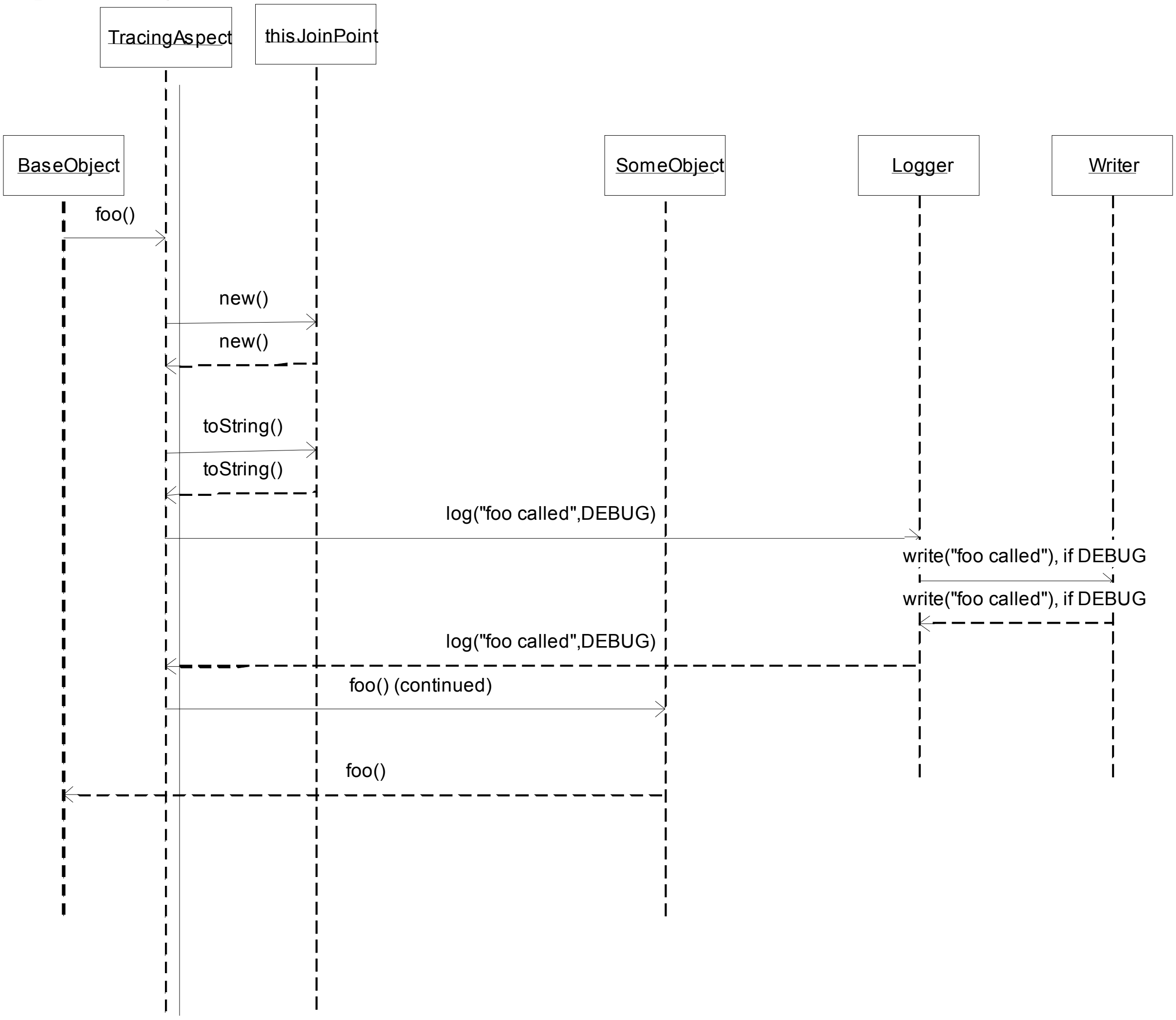


However, this picture changes if aspects are involved. A clean aspect oriented solution requires, that it calculates the parameters dynamically

based on reflective information, which is much more expensive and often even, too expensive.
A simple advice like,

```
        before():execution(public void *.foo()){
              Logger.log(thisJoinPoint.toString(),DEBUG);
        }
```

which realizes similar functionality, to the manual written code from the diagram above, requires more calls. It must generate at first a new joinpoint object and it must convert the thisJoinPoint object to its string representation. Somewhere later the garbage collection has to spend some extra time to clean up the intermediately created objects. The user can easily spot the additional calls (and the additional work for the CPU) in the next sequence diagram.



The typical solution to add or remove the additional functionality by adding or removing the aspect from the build set, may be acceptable for aspects which are only used during development time on the development machines, but this approach is in general not applicable in practical deployed code for several reasons like:

- A full rebuild or deployment takes too long.
- It is often necessary to enable or disable this functionality dynamically.
- The administrative overhead of a rebuild, may be too big. (e.g. Any change performed by a developer or by the project builder may need official approval.).
- The configuration management may be to inflexible, to support these optional featured applications.
- Quality assurance may prohibit a re-deployment without a complete testing cycle, if the aspect is added or removed.
- A 99,9% availability or higher QoS guarantee can even prohibit to shutdown and restart the application to redeploy the aspect solution.

* * * * *

How can the execution of an advice, which implements optional functionality, be turned on or off dynamically?

* * * * *

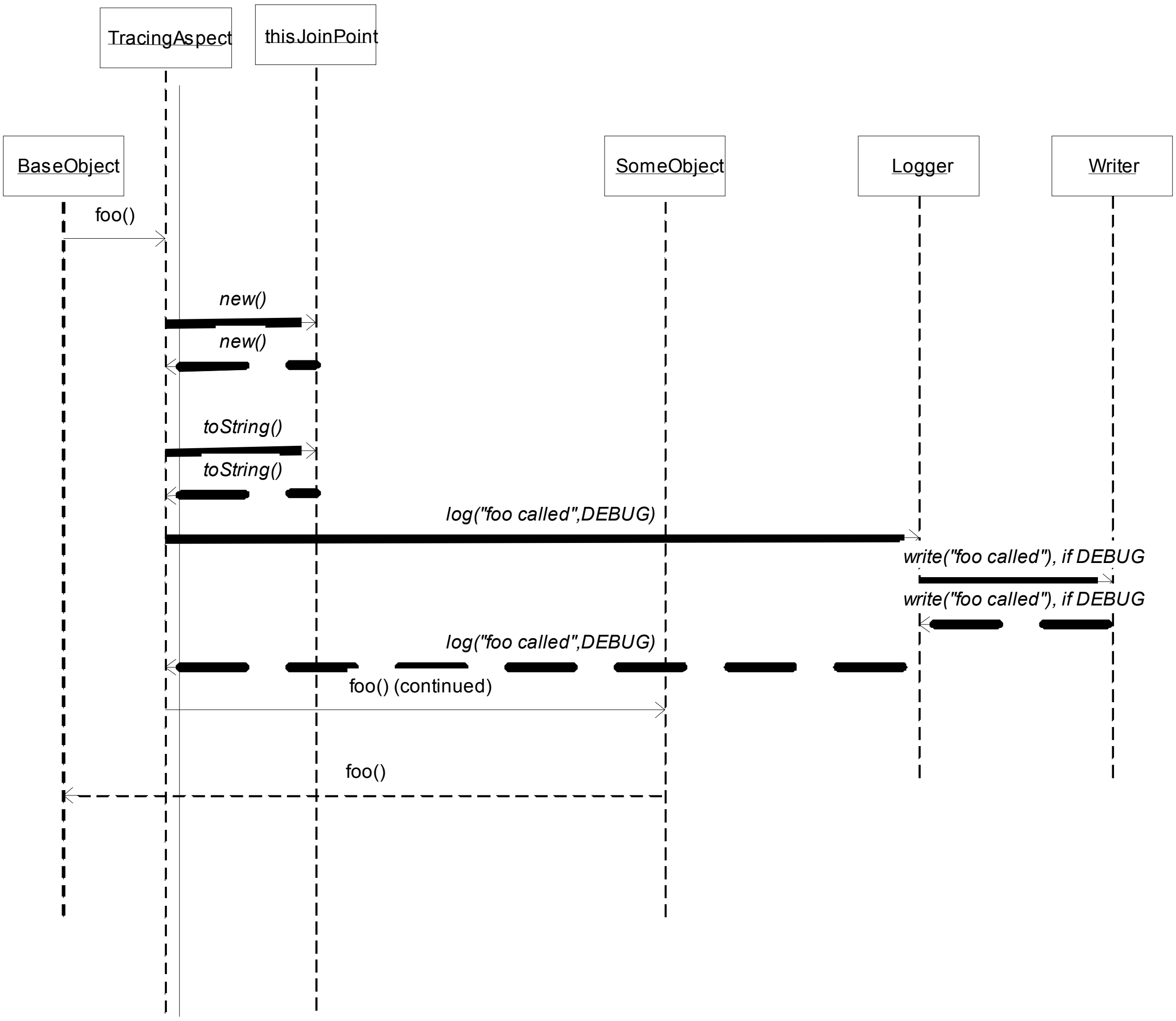
Solution

Extract the decision, if the optional functionality is required, from the advice and add it to the pointcut. The pointcut can now decide without unnecessary processing unnecessary work, if the additional functionality is need, and invoke the advice only if this kind of functionality is really required.

Sometimes it is easy to spot the DECISION right in the code of the advice. More often the DECISION must be extracted from the library itself. In addition, the library should expose the relevant code fragments, especially if there is an if-statement depending on a simple comparison, hidden in the library. In two cases I have reimplemented the DECISION, based on the published algorithm of the library.

Discussion

Following diagram shows in bold lines, which method calls of the above diagram are executed, if we are in the DEBUG mode.



This pattern is often used in conjunction with the pattern POINTCUT METHOD[4]. These combination must be used, if an “if pointcut” is not available in the AOP language or framework in use. The JBoss- and the Nanning framework do not provide an “if-pointcut” yet. An implementation of this pattern uses there the POINTCUT METHOD.

If the AOP environment does support dynamic aspect composition, dynamic aspect composition competes with this pattern. If the dynamic composition mechanism is efficient enough, the dynamic composition mechanism could be used. If the rest of the pointcut statement is quite expensive to evaluate this pattern, could be used to speed the decision process of the pointcut evaluation.

Example

A very detailed tracing can easily degrade the performance of an application by a factor of 1000. Most tracing and logging API have therefore a possibility to decide as early as possible, if some information has to be traced. Typical java code fragments like

```
logger.warn(“myclass”,”mymethod”,”entering”);
```

require normally only the very limited performance overhead of an additional function call, which gets additionally often inlined by current VMs. The performance overhead is neglect able, if tracing is disabled. If someone wants to extract the same functionality to an aspect, the logger call gets much more expensive because reflective AOP-code must be used, or complex data structures must be generated, etc. The corresponding code:

```
before():execution(* *(..)){
            Signature s=thisJoinPointStaticPart.getSignature();
            logger.entering(s.getDeclaringType().getName(),s.getName());
}
```

requires much more overhead. A Signature Object must be created through an invocation through the static JoinPoint Object. Plus several methods must be invoked, some are known for not being quite fast. These overhead must be executed for each call to the logging API, before the logging framework does even have the change to decide if it has to log. Additionally all intermediate variables have to be collected by the garbage collection later. It is therefore obvious that tracing implemented with advice like this runs much slower than the straightforward implementation. But the straight forward manual implementation creates a nightmare for code evolution and is therefore not acceptable either.

However following advice with its pointcut runs nearly as fast as the manual instumented code. Detailed performance information can be found in [17].

```
before():execution(* *(..))&& if (logger.isLoggable(Level.FINER)){
      Signature s=thisJoinPointStaticPart.getSignature();
      logger.entering(s.getDeclaringType().getName(),s.getName());
}
```

Antiidiom NotWithin

Motivation

Someone wants to apply an advice quickly around a large number of joinpoints, e.g. with the intention to audit the behavior of some code fragments. Typical examples are tracing, gathering profiling information, collecting test data for mock objects, etc. In these scenarios, the advice appears easily to some joinpoints, which are within the advice body. The result is often infinite recursion. Advice with its anonymous pointcut is a typical problem. The following demonstrates this problem:

```
public aspect InfiniteTracer {

      before():call( * *.*(..)){
            System.out.println(thisJoinPoint);
      }
}
```

This aspect advices all calls to all methods. Unfortunately this advice does contain some calls (e.g. System.out.println()) and thisJoinPoint.toString() in its advice. As a result, the VM aborts without any error messages.

* * * * *

How can a developer prevent the recursive application of an advice to its own advice body?

* * * * *

Naïve Antiidiom Solution

It is obvious that the selected joinpoints of this pointcut has to be limited. So it is quite natural to eliminate all Joinpoints which are inside the advice itself with a NotWithin pointcut. So the above code is often rewritten with the help of the within pointcut.

```
public aspect InfiniteTracer {

      before():call( * *.*(..))&&!within(InfiniteTracer){
            System.out.println(thisJoinPoint);
      }

}
```

Discussion

However this commonly seen “idiom” is a common Antiidiom, which is unfortunately often used to explain the recursive advice behavior. (e.g. in the AspectJ documentation.[9]) Unfortunately it is only rarely called as an antiidiom, because this approach works only for small toy samples. Consider the following situation some other developer creates a second aspect to redirect all calls from System.out. to a more sophisticated logging API, e.g. log4j [15].

```
public aspect LoggerRedirector {
      public static final Logger
            logger=Logger.getLogger(LoggerRedirector.class);

      void around(String param,PrintStream p):
            call(public void java.io.PrintStream.println(String))&&
            args(param)&&target(p)
      {
            if (p==System.out)
                  logger.info(param);
            else
                  proceed(param,p);
      }
}
```

Now if this second aspect is added an infinite indirect recursion will happen. The aspect InfiniteTracer advices the first call in the program. Unfortunatly the call to System.out.println(String) will be advised by the advice from the aspect LoggerRedirector.

In most cases where this antiidiom is used, it should be used to prevent recursive execution of an advice in a callstack. However this idiom prevents ONLY the recursive application of an advice to all advices in the same aspect, which is a clear subset of the problem which the developer tried to solve. Unfortunatly in small examples he will be often lucky enough, that this small subset is sufficient to keep the initial test up.

There are several other problems beside the easily occurring indirect infinite recursion. Code, which does not really need the advice gets advised, which can easily degrade the performance. The advised code generates often some unexpected behavior, because quite a lot of Joinpoints are hidden behind the syntactic sugar of the java programming language. E.g. following method contains not only Joinpoint which can be easily spotted.

```
      public void concatString(){
            String x=System.currentTimeMillis()+" "+this;
      }
```

The joinpoint which can be easiest spotted by looking at the code, is the call to System.currentTimeMillis().
The “syntactic sugar” operator “+” hides several joinpoints. The advice at these joinpoints can easily get into conflicts with some other advice.

There is no real motivation for the use of this anti idiom in code from real projects. From my experience, this antiidiom can be used with care in following situations similar to following:

- Someone wants to benchmark the compile process or the execution of an adviced program.
- Someone wants to gather easily statistical information about the structure of a program. By capturing the weaving information from the compiler someone can easily collect these data.
- Somewants wants to experiment with some aspects at some codefragments.

In most practical situations, the drawbacks easily outrun the advantages. This anti pattern is often caused by sloth (much to general pointcuts) and ignorance (Not thinking and reading how AOP really works). A detailed description of these root causes can be found in [16]

Currently I have spotted this antiidiom only in AspectJ code, I identified following reasons for this fact:

- This antiidiom is currently only advocated in the AspectJ community.
- Several other AOP languages or Frameworks do not permit that an advice or an aspect may advice itself. [1]
- AspectJ is currently the “plug and pray” AOP newbie language. Other AOP languages or framework seem to be used by more experienced AOP users.

Possible Solutions for the intended use

This antiidiom is often used in “ugly” code, with very few long methods, with a few classes, which do only use libraries. These methods cry after some Refactorings. The well known refactoring pattern “extract method” [4] is strongly suggested. As soon as this code is split up into several smaller methods and brought into an acceptable shape, execution pointcuts limited to methods in some classes or packages can be used, which avoid typical recursive situations.

Another approach which often is even more then sufficient, is to limit the pointcut to more concrete types, methods and packages. Most often this narrowing of the scope of the pointcut is fully sufficient.

The best approach to eliminate the need of this antiidiom is to improve the pointcut definition and to improve the structure of the program, in order to be able to use more appropriate pointcuts.

In all of the interesting cases there are several interesting options, the most common are:

If top performance is not a key concern, a combination of cflow and cflowbelow pointcuts could be used. Another option is the use of patterns like the ADVICE FLOW POINTCUT (see below).

[1] I consider the lack of this feature a very strong drawback of any AOP tool which lacks this functionality for several reasons, e.g. Several important patterns are not possible, without this feature, Implementing pure AOP a program written only in AOP is not possible.

If the code executes in a fully synchronized environment (e.g. inside an appserver), a pointcut method or an if pointcut, combined with a local variable, which holds a recursion flag, can be used to solve the problem. In a multithreaded environment the recursion flag can be implemented in a threadlocal variable or in a custom implementation of a kind of threadlocal variable.

However there are several different other options. A detailed discussion of all possible solutions is a whole pattern collection or pattern language of its own.

advice flow pointcut

Motivation

I mined this pattern in two typical situations. A reliable solution for the forces of the Antiidiom NotWithin is one of them. This motivation is not duplicated here. The other one is discussed in this section.

Abstract Aspects contain often abstract pointcuts and advices, which must not execute twice in a call stack. A typical example for such a scenario is opening a transaction, if the transaction system does not support nested transactions. Often the software is build up recursively of components, where each component must be fully wrapped by the critical advice. The subaspects define normally the wrapping joinpoints. The aspect on the other hand must also ensure, that the advice will be executed exactly one. A naïve implementation would use pointcuts similar to

```
Abstractpointcut()&&!cflow(Abstractpointcut())
```

in the abstract aspect. However this idiom does not work. It is at least a common pitfall for abstract pointcuts, if not an antiidiom for aspect oriented programming with AspectJ. The concrete subaspects of the abstract aspect form an individual aspect class. The abstract pointcut from above is expanded to following statements in its instantiation process:

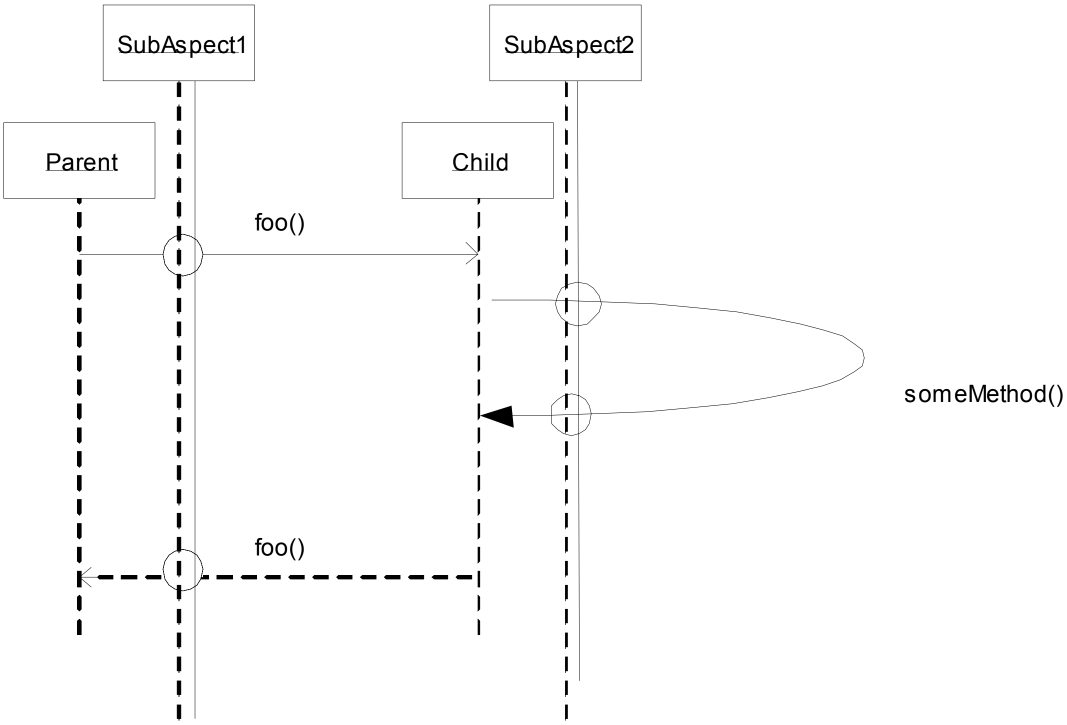
```
    SubAspect1.Abstractpointcut()&&!cflow(SubAspect1.Abstractpointcut())
```

And

```
    SubAspect2.Abstractpointcut()&&!cflow(SubAspect2.Abstractpointcut())
```

And so on.

And these fragments do not match the intentions, because the pointcut from SubAspect2 can still select joinpoint in the cflow of the pointcut from SubAspect1 and vice versa. Following Sequence diagram illustrates this behavior,



SubAspect1 intercepts the call from the Parent and SubAspect2 intercepts the call from the Child.

Problem

* * * * *

How can someone guarantee, that an advice is only executed once in an execution flow?

* * * * *

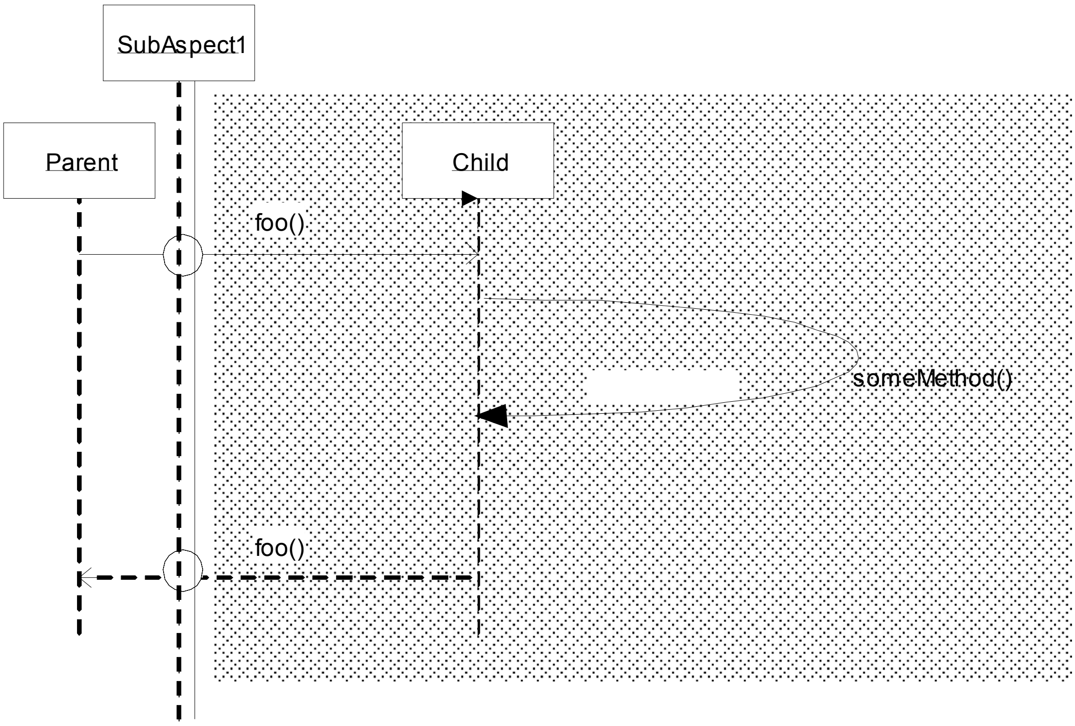
Solution

Use an pointcut, the “ADVICE FLOW POINTCUT”, as the body of the cflow statement.
The “ADVICE FLOW POINTCUT” is an adviceexecution()-pointcut around the “SINGLETON ADVICE” in the abstract aspect.

So the ADVICE FLOW POINTCUT looks like:

```
  Abstractpointcut()&&!cflow(adviceexecution()&&within(Abstract
Aspect))
```

The behavior of the pattern is illustrated by following diagram, where the shadowed region is the cflow of the adviceexecution, which prevents, that SubAspect2 can advice the someMethod execution.



Discussion

The joinpoints selected by the ADVICE FLOW POINTCUT, are not in the original code base. However these joinpoints are in the code base of the added aspects. This idiom is used to ensure that an advice executes exactly one. Most often other idioms, which do decide where the advice is bound to are involved. E.g. ABSTRACT POINTCUT, MARKER INTERFACE, etc. [4].
This pattern prevents safely advice recursion and is prefereable over the common NOTWITHIN antiidiom

call(* *.*(..))&&!within(MyAspect)

which does not prevent safely infinite recursion.
The ADVICE FLOW POINTCUT uses a cflow pointcuts, which needs some time to evaluate at runtime, especially on old VMs. So it is not sensible to use this kind of advice in very fine grained pointcuts, which do map to thousands of places in the code bases, and which will be often passed in large iterations. If you want to use the ADVICE FLOW POINTCUT pattern it is quite useful to limit the potential joinpoints as much as possible. It is safe to use this pattern from a performance standpoint at all joinpoints at exposed places of your application or component, e.g. at the border for interprocess calls, e.g. before calls to middleware, IO, transaction monitors, etc, because the already existing overhead at these places easily outruns the additional created by the cflow statement().

Acknowledgements

There are several people who helped me writing this paper. I want to say thanks to my shepherd Uwe Zdun for his supportive and insightful comments. I want to say thanks to

Brandon Glenk for correcting several bugs in my use of the English language. Finally I want to say thanks to my wife and my daughter for their patience, while I was writing this paper.

References

[1] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J: "*Design Patterns: Elements of Reusable Object-Oriented Software*," Addison-Wesley, 1995.

[2] Hanenberg, S.; Costanza, P.: "*Connecting Aspects in AspectJ: Strategies vs. Patterns"*, First Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD'01, Enschede, April, 2002

[3] Schmidmeier, A.; Hanenberg, S.; Unland, R.: "*Known Concepts implemented in AspectJ*", 3rd Workshop on Aspect-Oriented Software Development of the German Informatics Association, March, 2003

[4] Schmidmeier, A.; Hanenberg, S.; Unland, R.: "*AspectJ Idioms for Aspect Oriented Software Construction"*, EuroPLoP'03 June, 2003

[5] Schmidmeier, A: "*Using AspectJ in Component-Based Architectures on the Server Side", *Invited talk at AOSD'01, Enschede, April, 2002

[6] Schmidmeier, A.: "Using *AspectJ to Eliminate Tangling Code in EAI Activities*", practitioners report at AOSD´04, Boston

[7] Laddad R: "*AspectJ in Action"* Manning Publications

[8] Gradecki J.D, Lesiecki N.: "*Mastering AspectJ*", Wiley, March 2003

[9] AspectJ, http://www.eclipse.org/aspectj, June 2004

[10] AspectWerks, http://aspectwerkz.codehaus.org/index.html, June 2004

[11] JBoss-AOP, http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop, June 2004

[12] Nanning, http://nanning.snipsnap.org/space/start, June 2004

[13] Hanenberg, S.; Unland, R.: *Parametric Introductions*, 2nd International Conference of Aspect-Oriented Software Development (AOSD), Boston, MA, March 17-21, ACM Press, 2003, pp. 80-89.

[14] Sun: Java Foundation Classes/Swing *http://java.sun.com/products/jfc/,* June 2004

[15] Eclipse.org: SWT, http://www.eclipse.org/articles/main.html, June 2004

[16] Fowler, M: *Refactoring*,, Addison-Wesley Professional, June 1999

[17] Hugunin J., Hilsdale E.: *Advice Weaving in AspectJ*, AOSD'04, Lancaster April 04

[15] Apacher.org: Log4j, http://logging.apache.org/log4j/docs/ June 2004

[16] Brown W., Malveau R., McCormick H., Mowbray T.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley, March 1998

[18] Sun : Dynamic Proxy Classes http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html , June 2004

APPENDIX Aspect oriented programming with AspectJ

In addition to Java AspectJ provides a number of new language features which will be explained here in more detail: aspect, pointcut, advice and introductions. The intention of this section is to give people which are relatively new in the area of aspect-oriented programming a brief introduction into the programming language AspectJ.

1.1 A-1.1 Joinpoints

Joinpoints are these points in the program, where someone wants to add new code or replace existing code with new one, in order to simplify, improve or enhance the functionality, design performance, fault tolerance, etc. In AspectJ joinpoints are points in the execution flow like:

- calling
- or executing a method,
- executing an advice,
- reading or writing a variable,
- handling an exception,
- initializing an object
- or class.

1.2 Pointcuts

A *pointcut* selects a collection of join points. To specify pointcuts AspectJ provides a number of pointcut designator like `call`, `execution`, `adviceexecution`, `get`, `set`, `initialialization`, `handler` which select a joinpoints based on the defined program flow. However these Joinpoints are not sufficient to select all relevant joinpoints successfully in all non trivial applications. One way to overcome this solution is the pointcut-method pattern presented in [4]. AspectJ and AOP languages with an even more sophisticated and powerfull pointcut languages like [13] offer additional pointcuts to improve the usability and overcome drawbacks of the pointcut method pattern. Some are based on the lexical structure of the application, like within and withincode. Both select all joinpoints inside a class, package or a method. Some other define the joinpoints based on type of the caller, callee or the arguments. These pointcuts are called `this`, `target` and `args`.

Each pointcut can be combined using Boolean operations. For example the following pointcut has the name `callFooFromAToB. It` describes all join points where a call to the method `foo` of class B is performed within the lexical scope of A.

```
pointcut callFoofromAToB(): within(A) && call(void B.foo());
```

The pointcut consists of two pointcuts which are combines by the logical operator `&&`. The pointcut `within(A)` desribes all join points in class A, `call(B.foo())` describes all join points where the method `foo` of class `B` is called. Named pointcuts (like pointcut `callFooFromAToB`) can itself be used in other pointcut definitions.

AspectJ distinguished between static and dynamic pointcuts. A static pointcut describes join points which can be determined by a static program analysis. In the example above all join points can be determined statically. On the other hand there are join points which cannot be statically determined. For example the following pointcut determines all join points where a message foo is sent from an instance of A to an instance of a subclass of B.

```
pointcut dCallFoofromAToB(): this(A) && call(void *.foo())
                             && target(B+);
```

In contrast to the previous pointcut definition it now depends on the participating objects whether a certain line of code represents a join points described by this pointcut or not. For example a call of foo in a superclass of A might now be a valid join points as long as the object is an instance of A. The pointcut **call**(void *.foo()) now determines all call join points to all existing foo methods independent of in what classes a method of this signatures occurs.

Pointcuts permit to export parameters. For example the following pointcut binds the runtime object of the sending object which is of type A to the variable a.

```
pointcut boundA(A a): this(a) && call(void *.foo())
                      && target(B+);
```

Often someone wants to select only joinpoints which happen in the callstack of a method, or an advice. These joinpoints can be selected by with the pointcuts cflow and cflowbelow. E.g. following sample selects all joinpoints which haben in the callstack of the call to a public method foo in any class.

```
cflowbelow(call public void *.foo())
```

Please note that cflow or cflowbelow pointcuts are nearly always used in combination with other pointcuts to limit the scope of the pointcut. The if pointcut is another popular way to limit a pointcut. The if pointcut requires a Boolean expression, which can be statically evaluated. AOP languages or frameworks, which do not have an if pointcut can easily use the pointcut method pattern to select the appropriate pointcut.

1.3 Type Patterns

AspectJ allows the use of type patterns whenever a single type or a type is required. Table ::: contains an overview of the valid type patterns

*	A sequence of any characters, except .
..	A sequence of characters, starting and ending with a .
+	The type itself or any subtype
A||B	Type pattern A or type pattern B
A&&B	Type pattern A and type pattern B
!A	Not type pattern A

1.4 A-1.2 Advice

A piece of advice specifies the code that is to be executed whenever a certain join point is reached. It represents the crosscutting code because it may be executed at several execution points in the program. The declaration of a piece advice needs to specify at what joinpoints, it is meant to be executed, in order to select these joinpoints it uses a single or a combined pointcut statement. Additionally, it must specify at what point in time it is supposed to be executed: an advice may either be executed before or after the original code at a certain joinpoint or may even replace it.

In AspectJ pointcut methods are defined as follows:

```
before(): aPointcut() {...do something...}
```

This method is executed before a join point determined by the pointcut aPointcut is reached (the modifier before() is responsible for deciding, at which point of the interaction the method should be invoked). Furthermore an advice can be executed around or after a join point. An advice which is declared around an joinpoint, replaces the original code at this joinpoint. Most often the advice adds only some additional functionality to the original code, (e.g. some caching, some exception handling). AspectJ offers the keyword proceed for these scenarios. The keyword proceed executes the original code (or the next advice in the advice chain) at this pointcut.

A piece of advice can refer to pointcut parameters. For example the following piece of advice

```
before(A a):boundA(a) {
  System.out.println(a.toString());
}
```

imports the pointcut parameter a of type A and sends in its body the message toString. Inside a piece of advice the keywords thisJoinPoint or thisStaticJoinPoint can be used which permit to reflect on the current join point.

1.5 A-1.3 Introductions

Introductions permit to add new members to classes (which is similar to open classes or mixins [3]) or to add new interfaces or superclasses to classes. Syntactically, introductions consists of the member definition and the name of the target type. To add new interfaces to a target type, AspectJ provides the keywords declare parents.

```
class A {  ... }
interface NewInterface {...}
aspect MemberIntroduction {
  public String A.newString;
  public void A.doSomething(){...}
}
aspect InterfaceIntroduction {
  declare parents: A implements NewInterface;
}
aspect TypePatternIntroduction {
  public void (A+).doSomething2() {...}
}
```

The code above contains an aspect MemberIntroduction that adds a field newString and a method doSomething to class A. The aspect InterfaceIntroduction adds the interface NewInterface to class A. The target type can be specified using so-called type patterns that permit to apply an introduction to several types at the same time.

1.6 A-1.4 Aspects

Aspects are class-like constructs which permit to contain all of the above mentioned constructs. In contrast to classes aspect cannot be instantiated by the developer. Instead, aspects define on its own how and when they are instantiated. Aspects can be declared abstract and abstract aspect can be extended (similar to the extends relationship in Java). Abstract aspects are not instantiated and their pieces of advice do not influence the base program. Similar to the member sharing in Java, pointcut definitions are shared along the inheritance hierarchy.

The instantiation of aspects is defined in each aspect's header. Aspects are either singletons, that means there exists only one instance of, or there are instantiated on a per-object basis. That means an aspect is instantiated for each object, which participates in a certain call-join point. e.g.

```
aspect MyAspect perthis(this(A)) {
  //pointcut definition
  // advice definition
  // introduction definition
}
```

The aspect above is instantiated for each instance of A matching the `this(A)` pointcut. By default an aspect is instantiated as singleton that means omitting "`perthis(this(A))`" in the example above means that there is exactly one instance of the aspect in the system. An Aspect can also be defined as an percflow or percflow below aspect. The runtime creates in this case a new aspect instance for each callstack which passes the pointcut definition inside the percflow or percflowbelow statement.

```
aspect MyAspect percflow(somepointcut()) {
  //pointcut definition
  // advice definition
  // introduction definition
}
```

Aspects can implement interfaces or extends classes or other aspects. Aspect Inheritance is a powerful feature which is often used in AOP idioms and patterns. For a discussion of several of them see [4].

Abstract aspects can define abstract methods, or abstract pointcuts besides its usual members. Advice can not be overwritten. Please note:

If an abstract advice is subclassed twice (as in following sample),

```
abstract aspect AbstractAspect {

  pointcut somepointcut():call(public * *.somemethod(..));
  before():somepointcut(){
      //someAdviceCode
  }
}

aspect Aspect1 extends AbstractAspect{
}

aspect Aspect2 extends AbstractAspect{
}
```

two instances (Abstract1, and Abstract2) of the abstract aspect are created. Both contain the pointcut somepointcut and bind their advice to their pointcuts. This results in the fact that the call to somemethod gets adviced twice.

Like classes Aspects can be declared in its own file, together with other classes and aspects in the same file, or even like inner classes as inner aspects. On the other side, aspects can contain inner classes and anonymous inner classes just like “normal classes”.

1.7 A-1.5 HelloWorld in AspectJ

This section just presents a small AspectJ variation of the well-known Hello World example. The class `BaseProgram` represent the base program the aspect `MyAspect` is woven to. The base class just instantiates itself and calls its method `sayHelloWorld`. The aspect defines a pointcut on the call of this method and a corresponding piece of advice.

```
public class BaseProgram {
  public static void main(String[] args){
    BaseProgram base = new BaseProgram();
    base.sayHelloWorld();
  }
  public void sayHelloWorld (){
    System.out.println("Hello World");
  }
}


aspect MyAspect {
  pointcut pc(BaseProgram b): this(b) &&
           calls(void BaseProgram.sayHelloWorld);
  void around(BaseProgram b): pc(b) {
    System.out.println("An instance of "
       + b.getClass().getName() +"  invokes sayHelloWorld");
    proceed(b);
    System.out.println("invocation done...");
  }
}
```

When `sayHelloWorld` is invoked the around advice is executed instead, which first prints out some additional text which depends on the calling object. Then the originally method is executed using the special command `proceed()` which can be used inside around advice. After the original message is shown the advice finally prints out some additional text. The result of calling the main method is:

```
An instance of BaseProgram invokes sayHelloWorld
Hello World
invocation done...
```