

Configuration of Aspects

Arno Schmidmeier

AspectSoft

Lohweg 19

91217 Hersbruck

Germany

Email Arno@_nospam_aspectsoft.com

Abstract.

Most aspect-oriented examples and actual usages describe scenarios where aspects provide infrastructure services to a set of objects. This is currently the dominant use case of aspects. The aspects are often assembled in aspect libraries for an easier use. These library aspects must be highly customizable and configurable in order to be applicable to many projects and platforms. This paper describes two patterns for advanced configuration and customization of aspects.

Introduction

Both patterns have been extracted from commercial aspect-oriented projects in which the author was evolved. (see [20, 21, 15, 16, 17]). However there are other references of use of these patterns available, e.g. the Springframework [18] describes Injected Aspect several times in its documentation and sourcecode. For illustrating the patterns the aspect-oriented programming language AspectJ is used (see [13, 7, 2, 3] for introducing material on AspectJ). All the samples and code fragments are written in AspectJ and some are configured with the Springframework and the names are influenced deeply by AspectJ and Spring terminology. However, the patterns are useful outside AspectJ and Spring, too. All patterns are to the authors' knowledge applicable to other aspect-oriented systems such as Aspektwerkz [1], JBoss-AOP [10], or Nanning [14].

Structure of the paper and format of the patterns

This paper describes two patterns, which deal both with the same problem “configuration of aspects”, which is common especially in infrastructure aspects. So all patterns share the general important parts:

- Motivation and general technical background,
- Problem,
- Forces.

So it is only natural to apply a kind of document refactoring “extract chapter” and discuss these section outside of the individual discussion of the aspects. To simplify the problem domain and elaborate the differences between the patterns more easily, both patterns will share the same motivation sample of a trivial cache aspect implementation, which should be configured. Each of the patterns

- Constructor Lookup

- and Injected Aspect,

has its own solution, discussion and example implementation section. The paper finishes with a general conclusions and a literature list.

Note:

For the purpose of Europlop I will append a short introduction to AspectJ, this introduction will be removed in the final proceedings.

Motivation and technical background

Software Development is a big cost factor for modern enterprises. Reducing the cost is therefore a major target of software engineering for decades. AOSD helps here by improving the modularity and maintainability. Especially generic crosscutting functionality can be easily moved to one common place, the aspect. However generic crosscutting functionality is normally needed in different places inside a larger application with slightly different configurations (e.g. different parameters, different configurable strategies, etc.). It would violate the idea of modularisation, if somebody would copy an aspect multiple times, replace and customize some configuration settings in each copy, just to customize the each aspect to its actual requirement.

If the aspect would be an object the solution would be simple. The object designer would provide the object with a parameterised constructor or some access methods. The user of the object could use the interface of the object to customize it to its need. However such a straight forward OO-approach via constructors or setters is normally not applicable for aspects for a lot of reasons especially:

- The user of the aspect, the base application, is often not aware that it is advised by an aspect.
- The aspect advices often many places in the base application, so how do we know, where in the base application we do have to configure the aspect.
- Advanced AOP languages like AspectJ and Sally generate the aspect on demand, and they do not provide any natural hooks into the configuration process.
- Aspects can have often only a default constructor (e.g. AspectJ)
- Aspects can only be accessed via its accessormethods (e.g. aspectOf in AspectJ) after they have been “in action” e.g. advising an joinpoint. But executing an advice does normally require a correct and fully configured aspect

For example, we have following simple caching aspect:

```
public abstract aspect CachingAspect {  
  
    //////////////////////////////////////  
    // The pointcuts to extend  
    public pointcut CacheInvalidatingMethods();  
  
}
```

```

public abstract pointcut ExpensiveMethods();

////////////////////////////////////
// The configuration Properties
/**
 * The CacheProvider is an interface to a cache implementation.
 * Possible implementations could be based on simple solutions
 * like Hashtables or on complex products like
 * distributed object dictionaries
 */
protected CacheProvider cacheProvider;

/**
 * The CacheKeyCalculator is an interface to an algorithm
 * which calculates the hashkey from the joinpoint.
 */
protected CacheKeyCalculator cacheKeyCalculator;

before():CacheInvalidatingMethods(){
    cacheProvider.invalidateCache();
}

Object around():ExpensiveMethods(){
    try{
        Object key=cacheKeyCalculator.getKey(thisJoinPoint);
        if (key!=null){
            return cacheProvider.getCachedValue(key);
        }
    }catch(RuntimeException e){
    }
    Object toReturn=proceed();
    try{
        cacheProvider.cache(thisJoinPoint,toReturn);
    }catch(RuntimeException e){
    }
    return toReturn;
}
}

```

This simple aspect needs three configuration dimensions:

- Where the aspect must be applied,
- Which CacheKeyCalculator must be used,
- Which CacheProvider must be used.

Each concrete aspect must now provide a complete configuration of all of these dimensions. The first dimension is normally solved by the application of the pattern `ABSTRACT POINTCUT` which is discussed in [15]. The configuration of the other dimension must be provided by each concrete subsaspect of `CachingAspect`. This leads

- to redundant aspect configuring code,
- or to many different implementation approaches for the same or a similar configuration problem.

Both effects are not desirable. We need a reusable modular configuration approach, which can be shared by all concrete aspects of `CachingAspect`.

So we have following general problem:

Problem

How can aspects configure themselves?

Forces

During the design of reusable aspects, especially if we deal with a collection of configurable aspects we have to deal with following forces:

- Dependencies and constraints
The number of dependencies and constraint which are imposed from the aspect to the base application. If we increase this number we decrease the number of projects and platforms, which can reuse our aspects, and vice versa. Decreasing of this number increases the complexity of the aspect development and does often decrease the ease of configuration.
- Ease of the configuration:
The ease of configuration should not exceed an acceptable and reasonable amount. Especially beginners in writing aspect libraries overdose often the configurability. I remember a case where a smart developer wrote a highly configurable tracing library aspect. Unfortunately it took normally longer for aspect users to set up and debug the configuration than writing a sufficient but simple tracing aspect basic from the scratch. Configuration should be at least a magnitude easier than implementing the aspect. Following questions could help you in finding the right balance
 - How difficult is it to configure the aspect?
 - How likely is it to get the configuration wrong?
 - How difficult is it to write a simple non configurable solution?
 - How likely is it to get the simple non configurable solution wrong?
- Adoptability to the configuration approaches of the organisations or a given platform:
Each operating organisation has found its own way how to organize the configuration of its applications in production. Aspect configuration approaches should be able to adopt to the configuration approaches of the organisation, which will apply the aspects in their applications in their products. Increasing this force increases the complexity of the aspect development or decreases the ease of configuration.
- Consistency of Configuration:
It is highly desirable that all aspects of an aspect library are configured in a consistent style. Optimizing the consistency of the configuration reduces the amount of dependencies and constraints, increases the likeliness of reuse and eases of configuration (by reducing the complexity), however it increases the complexity for the aspect development due to the increased need of communication between the aspect developers.
- Complexity of the aspect development
Optimizing any of the forces above, increases the complexity of the aspect development. E.g. decreasing the dependencies increases the number of indirections, the amount of used patterns, abstraction layers, which are typical signs of complexity of the aspect development. By increasing this force, it is more unlikely that the aspect

gets successfully and fast implemented, tested and deployed. If this force exceeds a given number, then there is no business case anymore for the use of aspect libraries.

Constructor Lookup

Solution:

Wrap the configurable and customizable data and behaviour behind some well known interfaces. Provide a constructor in the aspect, which is responsible to retrieve implementations of these interfaces from well known addresses, with a well known lookup protocol. E.g. use Java Naming and Directory Interface (JNDI) on the Java EE platform, to retrieve a Java Transaction Architecture (JTA) resource.

Discussion and Context

This approach is straight forward and natural to implement. There is a high probability, that this configuration approach matches exactly the configuration approach of the organisation or of the platform, especially, if the access to a common service like JTA is configured with this pattern. However this approach has some prerequisites, which are often not meet:

- There must be a lookup protocol, which is widely available and used in the intended domains of the aspect.
- The interfaces must be from a nature, which can be stored and retrieved from the directory system.
- The configuration objects must be stored in a well known “standardized” location, or the aspect has to standardize such a location.

Otherwise other additional approaches, e.g. configuration file, or the pattern `TEMPLATE ADVICE` [15] have to be used to determine the lookup protocol and the lookup address. In these cases it is recommended for simplicity reasons to use the other additional approaches to configure the aspect directly.

This pattern is therefore nearly only used in following contexts:

1. The aspect needs some general available services of a platform, which can or should only be retrieved through a standard discovery protocol on this platform. (A typical sample is a JTA resource in an EJB container).
2. The aspect is written for a set of projects, which are deployed on the same platform, with a very strict configuration approach. (e.g. For a library, which implements some company standards and which will be only used in projects, which adhere to the standard configuration schema and directory organisation of this company)
3. The aspects are written with a specific platform in mind, which has a consistent directory based application approach.

Forces resolved

This pattern has following impacts to the forces:

- Dependencies and constraints
This pattern creates a specific strong dependency to a fixed directory service. It requires that the target platform has a given directory service with a given structure in operation

- ease of the configuration: Directory services of common platforms have normally a large and good set of tools for an easy configuration. Also the operators and developers are normally used to the configuration process via a directory service. So this force is fully resolved.
- Adoptability to the configuration approaches of the organisations or platforms:
This pattern can only be used, with the lookup service, which the developer had in mind at the time of writing the aspect. So this pattern can not resolve this force in any way.
- Consistency of Configuration: A consistent configuration of multiple aspects with this approach is easily possible. It is also easily possible to retrieve all configuration approaches through this pattern. This force can be therefore fully resolved with this pattern.
- Complexity of the aspect development: Accessing a directory service is relatively simple and easy. This force can therefore be considered as resolved.

Example:

Following code sample provides an abstract implementation from the CachingAspect of the motivating section.

```
public abstract aspect CachingAspectConstructorLookUp extends CachingAspect
{

    public CachingAspectConstructorLookUp() {
        try {
            InitialContext context=new InitialContext();
            cacheProvider=lookupCacheProvider(context);
            cacheKeyCalculator=lookupCacheKeyCalculator(context);
        } catch (NamingException e) {
            // ... Configuration Exceptionhandling
        }
    }

    private CacheKeyCalculator lookupCacheKeyCalculator(InitialContext
context) throws NamingException {
        return (CacheKeyCalculator)
            context.lookup(getKey2CacheKeyCalculator());
    }

    private String getKey2CacheKeyCalculator() {
        return "KEY_to_CacheKeyCalculator";
    }
}
```

```
protected CacheProvider lookupCacheProvider(InitialContext context)
throws NamingException {
    return (CacheProvider) context.lookup(getKey2CacheProvider());
}

protected String getKey2CacheProvider() {
    return "KEY_to_CacheProvider";
}
}
```

The pattern `ABSTRACTPOINTCUT` is used, to define where the aspect should apply. `CONSTRUCTORLOOKUP` is used to configure the `cacheProvider` and the `cacheKeyCalculator`. The lookup approach can be further customized on demand with the application of the pattern `TEMPLATEADVICE`.

Injected Aspect

Solution:

Use a dependency injection framework (DI framework), to configure the aspect. When the aspect is used for the first time, let the aspect invoke a dependency injection framework (e.g. Spring Beans Container, HiveMind) for its configuration. Quite often this invocation is triggered by a secondary aspect (the invoker), which triggers the invocation of the DI-framework.

Discussion and Context

This pattern is nowadays not very straight forward and natural to implement, this might change as DI and AOP are sneaking more and more into mainstream software development. This approach however fits nicely in the configuring scheme of an organisation, which already uses a DI framework. However this pattern has at least one of the following consequences:

- It relies on a specific existing DI-framework, whose infrastructure can be used,
- it introduces a dependency to its DI-framework,
- or it requires an additional abstraction layer, which encapsulates the dependency to a DI-framework. (“Any software problem can be solved, by adding another layer of abstraction”)

For the usages, where no DI-Framework is natively available, the aspect library might be shipped with a DI-framework.

And it does add at least another layer of indirection.

Implementation Discussion

Most DI-frameworks do more than just dependency injection.¹ They take often the responsibility for the creation of the object. Aspects can not be created naturally (e.g. by a constructor or a factory method []) on most AOP platforms e.g. Spring AOP, AspectJ, AspectWerks, JBoss AOP. So the classical OO-DI approaches of creation via a constructor and setter injection or constructor injection do normally not work.²

All widely adopted DI frameworks provide fortunately a way to configure objects, which they retrieve from static or non static methods. This is a prerequisite for wide adoption of DI-framework, because most OO-based systems uses patterns like singleton or factory methods to hide, encapsulate or control the creation of specific objects. Exactly the same approach can be used to configure located objects, e.g. objects retrieved with a deterministic algorithm from a pool, a directory service, a singleton, from some instance variables of an object graph, etc. So what we have to provide is a locator method which returns the instance of the aspect. Some

¹ Especially Spring 1.2.x [], HiveMind []

² The only exception is the combination of Spring AOP (Version 1.x, and Version 2) in combination with the Spring DI framework, where the AOP platform relies fully on the Spring DI framework. Any other AOP platform or language in combination with Spring suffers the same problem.

AOP platforms like AspectJ for example provide such a locator function. In AspectJ the method is called aspectOf.

But note: this locator function can often (e.g. in AspectJ) not be used before the advice is bound, and the advice is bound just before its first invocation. In other words, you simply can not use aspectOf as a locator function. So you have two extreme options:

You can provide either a “simple locator” function by yourself and invoke this method after the construction of the aspect. A library aspect like

```
public abstract aspect PostConstructorInitAspect {

    abstract public pointcut AspectInitialization();

    private static ThreadLocal aspect2Configure=new ThreadLocal();

    public static Object initAspect(){
        return aspect2Configure.get();
    }

    after():AspectInitialization(){
        System.out.println(thisJoinPoint);
        Object newAspect=thisJoinPoint.getTarget();
        String classname=newAspect.getClass().getCanonicalName();
        System.out.println(classname);
        synchronized(PostConstructorInitAspect.class){
            aspect2Configure.set(newAspect);
            Object obj=DCBeanFactory.getBean(classname);
            System.out.println("Aspect configured through Spring: "+obj);
        }
    }
}
```

can provide this functionality.

Or you write a library aspect, which access internal APIs from the DI-framework, which will configure an object, with configuration information for a given name. This approach can be implemented with Spring 2.0 with aspects like:

```
public aspect InternalConfigurationAspect extends BeanConfigurerSupport
{
    public InternalConfigurationAspect(){
        setBeanWiringInfoResolver(new
AnnotationBeanWiringInfoResolver());
        Resource res = new FileSystemResource("beans.xml");
        XmlBeanFactory factory = new XmlBeanFactory(res);
        setBeanFactory(factory);
    }

    after(Object beanInstance) returning : beanCreation(beanInstance) {
        configureBean(beanInstance);
    }
}
```

```

    }

    protected pointcut beanCreation(Object beanInstance) :
        initialization((@Configurable *).new(..) && this(beanInstance));
}

```

It is quite likely that future versions of IoC container or frameworks will provide such library aspects out of the box. Spring 2.0 provides already such a library aspect for the use with AspectJ.

Consequences

This pattern has following impacts to the forces:

- **Dependencies and constraints**
This pattern creates a dependency to a specific dependency injection framework or to an abstraction of an DI-framework. It may introduce a new DI-framework indirectly into a project, organisation or platform which does not yet use a DI-framework or use a different DI-framework.
- **ease of the configuration:**
DI configuration files have a tendency to get large and complex. Most DI-frameworks have currently no or only inconvenient configuration tools. Also quite a lot of developers and operators are currently not trained in administrating and using the DI-frameworks. The ease of configuration can be improved. However this might change as DI frameworks mature.
- **Adoptability to the configuration approaches of the organisations:**
With the help of a DI-abstraction layer it is straightforward to adopt to nearly all configuration approaches of all organisations. However this might require writing some adapter objects, if they are not shipped in the distribution of the DI-framework.
- **Consistency of Configuration:**
A consistent configuration of multiple aspects with this approach is easily possible. It is also easily possible to retrieve all configurable elements through this pattern. Also the standardized syntax of a DI-framework helps to keep a consistent configuration approach between multiple aspects of an aspect library.
- **Complexity of the aspect development:**
Using DI frameworks correctly and efficient is not yet in the head of every developer. As the long implementation discussion shows there is currently quite a lot of development complexity in this pattern. With future improvements with DI frameworks and a direct support from DI frameworks to aspects, this might change dramatically, e.g. applying this pattern with Spring 2.0, and AspectJ 5 means just adding one annotation and exposing accessor functions to the data which should be configured.

Example:

Following code sample provides an abstract implementation from the CachingAspect of the motivating section.

```

public abstract aspect AbstractCachingAspectUsingDI
    extends CachingAspect {
    public void setCacheKeyCalculator(
        CacheKeyCalculator cacheKeyCalculator) {
        this.cacheKeyCalculator = cacheKeyCalculator;
    }

    public void setCacheProvider(
        CacheProvider cacheProvider) {
        this.cacheProvider = cacheProvider;
    }
}

```

Each concrete aspect, annotated with an `@Configurable` annotation, can be configured in a spring bean configuration file.

```

@Configurable
public aspect ConcreteCachingAspectUsingDI extends
AbstractCachingAspectUsingDI {
    public pointcut ExpensiveMethods();
    //pcd to to expensive Methods ...
}

```

Closing remarks

Constructor lookup and injected aspect are very heavyweight and equally powerful patterns. Using both patterns at the same time is quite likely an overdose of aspect configuration. Most AOP libraries which want to use Constructor lookup will avoid therefore injected aspect and vice versa.

It seems that constructor lookup is the currently “easier to apply” pattern, especially for closed platforms, in a traditional environment.

Injected aspect is on the other hand the pattern of the future, which might be currently a technological overdose for some applications.

Acknowledgements

There are several people who helped me writing this paper. I want to say thanks to my shepherd Allan Kay for his very supportive and insightful and friendly comments, which helped to improve and finalize the paper. Finally, I want to say thanks to my wife Eva and my daughter Sophia for their patience, while the patterns have been mined and written.

References

- [1] AspectWerks, <http://aspectwerkz.codehaus.org/index.html>, June 2004
- [2] AspectJ, <http://www.eclipse.org/aspectj>, June 2004.
- [3] Colyer A., Clement A., Harley G.: *Eclipse AspectJ*, Addison Wesley, 2004.
- [4] Filman, R. E.; Elrad, T.; Clarke, S.; Aksit, M. (Eds.): *Aspect-Oriented Software Development*, Addison-Wesley, 2004.
- [5] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W. F.; Roberts, D.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
- [6] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

- [7] Gradecki J.D; Lesiecki N.: *Mastering AspectJ*, John Wiley & Sons, March 2003.
- [8] Hanenberg, S.; Oberschulte, C.; Unland, R.: *Refactoring of Aspect-Oriented Software*, 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts and Applications for a Networked World (Net.ObjectDays), Erfurt, Germany, September 22-25, 2003, pp. 19-35.
- [9] Hibernate, www.hibernate.org, June 2005.
- [10] JBoss-AOP, <http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop>, last access: June 2004.
- [11] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwing, J.: *Aspect-Oriented Programming*. In M. Aksit and S. Matsuoka (Eds.): *ECOOP '97 - Object-Oriented Programming: 11th European Conference*, LNCS 1241, Springer-Verlag, 1997, pp. 220-242.
- [12] Kiczales, G.: *The fun has just begun*, Keynote AOSD, Boston, 2003.
- [13] Laddad, R.: *AspectJ in Action*, Manning Publications, Greenwich, 2003.
- [14] Nanning, <http://nanning.snipsnap.org/space/start>, June 2004.
- [15] Schmidmeier, A.; Hanenberg, S.; Unland, R.: *AspectJ Idioms for Aspect Oriented Software Construction*, EuroPLOP'03 June, 2003.
- [16] Schmidmeier, A.: *Using AspectJ in Component-Based Architectures on the Server Side*, Invited talk at AOSD'02, Enschede, April, 2002.
- [17] Schmidmeier, A.: *Using AspectJ to Eliminate Tangling Code in EAI Activities*, practitioners report at AOSD'04, Boston, 2004.
- [18] Spring Framework, <http://www.springframework.org> June 2006.
- [19] Sun: Java Platform Enterprise Edition <http://java.sun.com/j2ee>, June 2005.
- [20] Schmidmeier, A.: *Aspect oriented Programming with Python*, Invited talk, ACCU, April 2005.
- [21] Schmidmeier, A.: *Let the code look like the design*, Invited talk, JAOO 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 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.

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)` describes 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.

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 |

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.

5 A-1.3 Introductions

Introductions permit to add new members to classes (which is similar to open classes or mixins) 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.

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 perflow or perflow below aspect. The runtime creates in this case a new aspect instance for each callstack which passes the pointcut definition inside the perflow or perflowbelow statement.

```
aspect MyAspect perflow(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 advised 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”.

7 Accessing Aspects

Aspects are normally accessed from advices. The AOP runtime system is responsible to invoke the appropriate advices at the joinpoints selected by the pointcuts. It is very common that aspects interact like any object with OO-libraries and contain state like any object. Other objects or aspects might want to access this state. The aspect could store itself therefore at a well known location, e.g. in a directory service or a singleton, from where other objects or aspects could retrieve and use it. A more convenient alternative is the use of the static accessor function aspectOf, which every non abstract Aspects class does provide.

The aspectOf function returns the aspect, if an aspect has been instantiated and throws an exception otherwise. The aspectOf function requires an additional parameter of type Object for aspect classes declared as perthis or pertarget to differentiate which aspect is referenced. Singleton aspects or aspect declared as perflow or perflowbelow can be accessed without any additional parameter. The AspectJ runtime engine provides for these aspect types the current context implicit to the aspectOf method. You can get therefore the current aspect instance for an aspect declared as

```
aspect MyAspect perflow(somepointcut()){
```

with following call.

```
MyAspect.aspectOf();
```

8 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...

```

However a plain AOP AspectJ-HelloWorld is also possible. Following code below is the shortest possible HelloWorld AOP solution:

```

public aspect HelloWorld {
    before():execution(public static void
        HelloWorld.main(String[])) {
        System.out.println("Hello World");
    }
    public static void main(String[] args) {
    }
}

```