AspectJ Idioms for Aspect-Oriented Software Construction

Stefan Hanenberg, Rainer Unland
University of Essen, Institute for Computer Science
Schützenbahn 70, D-45117 Essen, Germany
shanenbe@cs.uni-essen.de

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

Abstract. For concrete usage scenarios there are different options of how to use AspectJ's language features, and these options deeply impact the chances for further evolution of both base classes and aspects. Aspects can use aspect-oriented as well as object-oriented features. The combination of these mechanisms provides new powerful mechanisms. Idioms can help in guiding AspectJ users through this frontier of new language features. This paper proposes a number of regularly used idioms in AspectJ applications which are successfully practiced in real-world projects.

1 Introduction

Aspect-Oriented Programming (AOP, [10]) deals with code fragments which logically belong to one single module (a concern) but cannot be modularized because of limited composition mechanisms of the underlying programming language. Such code is called *tangled code* or *crosscutting code*, the underlying concern which is responsible for such tangling is called *crosscutting concern*. Aspect-oriented programming is about modularizing such crosscutting concerns into separated modules called *aspects*.

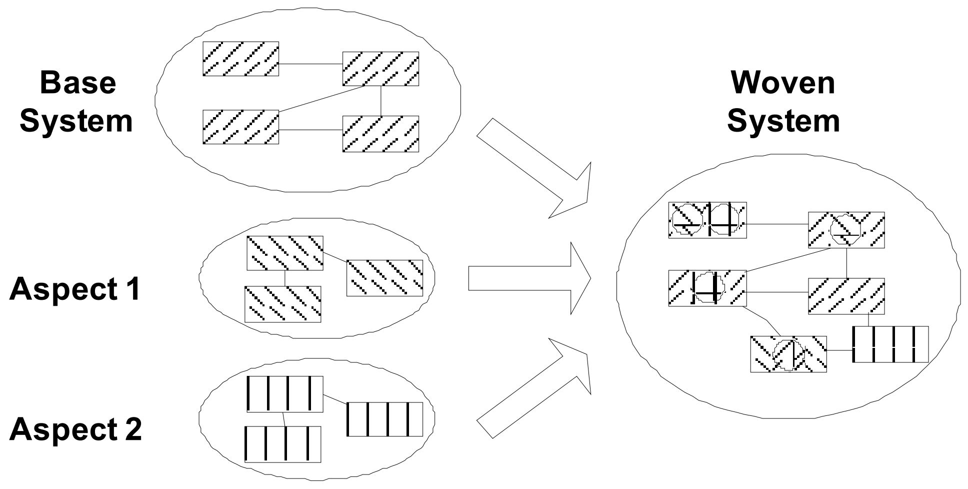


Figure 1. Weaving in Aspect-Oriented Programming

The underlying technique for preventing crosscutting code in aspect-oriented programming is the so-called *weaving mechanism* which is illustrated in figure 1. Two aspects each consisting of its own units are woven together with the original object-oriented base system into a final woven system. The weaver is responsible for combining each aspect with the object-oriented system and to create a final representation of the woven system.

For specifying how and where the additional aspects should be woven to the base system, aspect-orientation makes use of the *join point* concept. [8] introduces *join points* as "principled points in the execution of the program". Typical examples for join points are a certain method call from a certain caller object to a certain callee object or the execution of a constructor.

To facilitate the separate definition of the base system and aspects, AspectJ [11] (which is an aspect-oriented extension of Java) provides a number of new language features in addition to Java: *aspect*, *pointcut*, *advice* and *introduction*. The appendix at the end of this paper gives a short overview on the language features of AspectJ. A complete description of AspectJ can be found in the AspectJ programmer's guide available at [2].

The way how aspect-oriented features are applied has a direct impact on how reusable aspects are and how easy they can be applied to other object-oriented programs. Hence, the application of aspect-oriented language features must follow deliberate design decisions and cannot be used *ad-hoc* [8].

This paper discusses a number of regularly and successfully applied idioms for use with AspectJ. We present the idioms in the *Alexandrian style* [1] where we first describe the context of the idioms followed by three stars. Then, we describe the problem in one or two short sentences in bold face and afterwards discuss the problem in more detail. Then we present the solution of the problem in form of instructions (bold face). After this, we discuss the solution in more detail and present a typical implementation of the solution. After another three stars we discuss the relationship of each idiom to other idioms described here.

At the end of the paper we provide one example, which shows the application of the proposed idioms in a larger context.

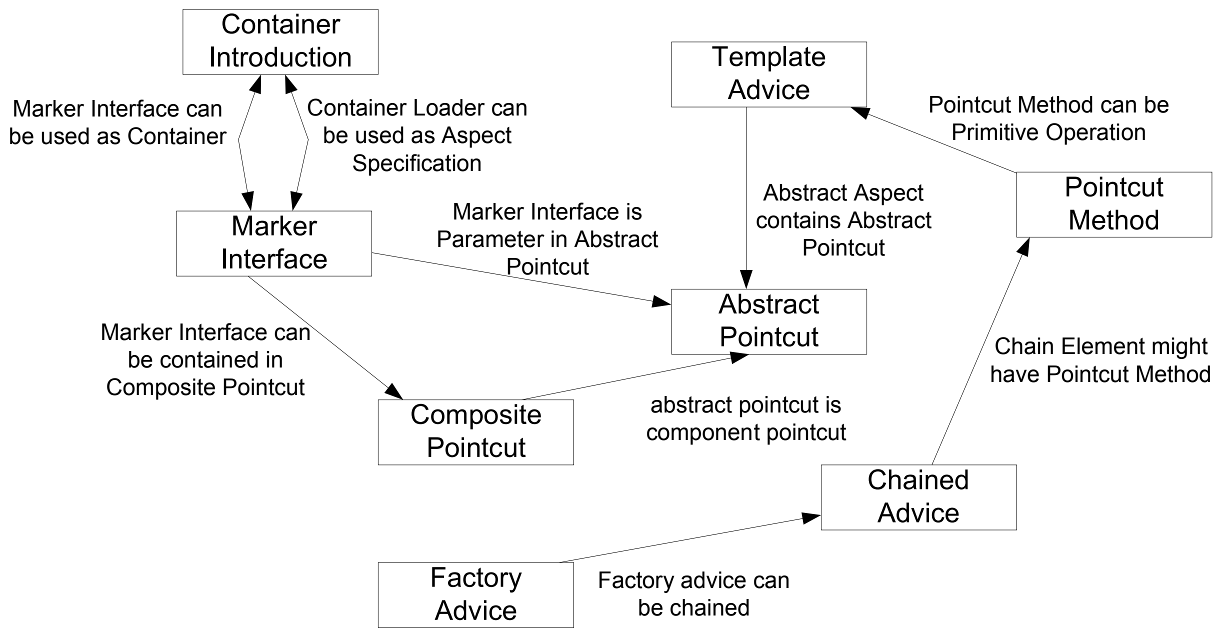

Container Introduction
Marker Interface can be used as Container
Container Loader can be used as Aspect Specification
Template Advice
Pointcut Method can be Primitive Operation
Abstract Aspect contains Abstract Pointcut
Pointcut Method
Marker Interface
Marker Interface is Parameter in Abstract Pointcut
Abstract Pointcut
Marker Interface can be contained in Composite Pointcut
Chain Element might have Pointcut Method
Composite Pointcut
abstract pointcut is component pointcut
Chained Advice
Factory Advice
Factory advice can be chained

Figure 2. AspectJ idioms and their relationship to each other

The idioms presented here and their relationship to each other are illustrated in Figure 2.

- Container Introduction permits to encapsulate a number of extrinsic features in a *container* which are later introduced to a target class. The container is filled by a number of *Container Loaders* which specify the extrinsic features.

- MARKER INTERFACE specifies crosscutting features depending on an interface, which is later connected to an application. The crosscutting feature is specified by *Aspect Specifications*.
- ABSTRACT POINTCUT specifies code that crosscuts a number of modules without specifying the corresponding join points. These join points have to be specified inside the *Concrete Aspect*.
- COMPOSITE POINTCUT decomposes a pointcut into a number of logically independent *Component Pointcuts*.
- TEMPLATE ADVICE permits to specify crosscuttings where each occurrence contains some variabilities. These variabilities are defined by *Primitive Operations* invoked inside a template advice.
- POINTCUT METHOD shifts the problem of pointcut specification from the join point language to the base language. A piece of advice's candidate pointcut determines a number join points where potentially the piece of advice should be executed. The *Pointcut Method* decides definitively whether or not to executed the piece of advice.
- CHAINED ADVICE permits to specify a number of pieces of advice at the same *Anchor Pointcut* which interact without knowing from each other.
- ADVICED CREATION METHOD permits to weave a number of aspects which participate independent from each other in the creational process of selected objects.

CONTAINER INTRODUCTION

In applications written in object-oriented programming languages each class usually consists of a number of members. Some belong to the core concern of the class, some belong side concerns..

* * *

In object-oriented languages we wish to encapsulate a number of different, additional functionalities in its own modules (e.g. class). How can we add these additional functionalities reusable to base classes and ensure that the base class does not need to have knowledge, about the implementation of the side functionalities?

In object-oriented languages usually all members of a class should address only the primary purpose, i.e. its base concern, the concern for which the class was built for. That means it is desirable to compose different classes into a new class which represents a combination of all those different concerns. Such a composition can be achieved using inheritance. However, languages like Java or Smalltalk do not support multiple inheritance which means that it is not easily possible to compose different classes into a new one. Although other concepts such as *mixins* [3] permit such a composition the previously mentioned languages do not directly support these concepts. But even assuming the existence of multiple inheritance or mixins does not lead to a desired solution: all classes where a number of features should be added to would contain a direct *extends* relationship to those classes containing the code for all different features. That means adding new features to a class requires invasive changes to it by editing its source code.

A language feature provided by AspectJ is *introduction* which is similar to *open classes* [5]. Introductions are defined inside an aspect and permit to add a number of members, interfaces or superclasses to target classes. However, the direct application of introductions has a number of disadvantages: introductions are not reusable. That means to apply a once defined introduction to a new target class requires a destructive modification of the aspect definition containing the introduction.

The problem is, that different features should be combined and composed to a number of different classes independent of each other. That means for an additional feature A and another additional feature B, that it should be possible to compose both to the same target class without performing destructive modifications inside the code defining these features. Those features should be reusable in a sense that it should be possible to add those features to new classes without the necessity to perform invasive changes on the code. That means application developers should be able to group a number of features and add them later to classes.

Therefore:
Introduce additional features to a *Container* and introduce the *Container* to different target classes independent of the feature implementation. That means the implementation of introductions and their application to target classes is done in two different aspects. The *Container Loader* introduces the feature-specific properties to the *Container*. The *Container Connector* introduces the *Container* to a number of target classes.

The only reasonable possibility to perform CONTAINER INTRODUCTION in AspectJ is to specify the introductions in a *Container Loader* aspect which introduces all members to an interface playing the role of the *Container*. The *Container* is defined only for the purpose of being used as a container, that means, there are no classes which define an *implements* relationship to that interface. The *Container Connector* is an aspect which introduces the *Container* to a number of target classes (or interfaces). Technically that means that the aspect declares a new *implements* relationship between that target class and the *Container*[1]. It depends on the application how the *Container Connector* looks like. Typically, there is more than one aspect that connects the container. Often, a *Container Connector* is implemented for each class the new features should be added to. In other situations (cf. [9]) a single *Container Connector* connects the *Container* to a number of target classes. Features can be combined to be later added to target classes by introducing *Container* to *Container*. That means the target class of a *Container Connector* itself is a (different) *Container*.

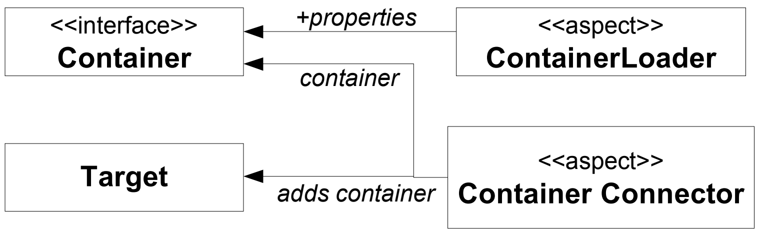


Figure 3. *Container*, *Container Loader*, *Container Connector* and Target Class in a CONTAINER INTRODUCTION

The main benefit of CONTAINER INTRODUCTION is the transparent introduction of a number of features. That means the implementation of the introduced features can change without the need to modify the *Container Connector*. However, the price for such a transparency is, that in case members of the *Container* and the target class are conflicting an error occurs in the *Container Connector* (cf. [9]).

The following example illustrates how a method `foo` is added to a class `TargetClass` using a CONTAINER INTRODUCTION.

```
public interface Container {}
public aspect ContainerLoader {
  public void Container.foo() {
    System.out.println("foo");
  }
}
public aspect ContainerLoader {
  declare parents:
    TargetClass implements Container;
}
```

* * *

The CONTAINER INTRODUCTION idiom is used very often in AspectJ since most aspects come with a number of extrinsic properties. Often, the *Container* also plays the role of a *Marker Interface* and the *Container Loader* corresponds to the *Aspect Specification* in MARKER INTERFACE.

There are large similarities between CONTAINER INTRODUCTION and the language features *mixins* or *multiple inheritance* (cf. [12]). Hence, this idiom is often used in AspectJ in situations where the application of mixins or multiple inheritance is appropriate but cannot be used because of a lack of support for these features in Java.

[1] Introductions permit to introduce implementation to an interface. That means, the code to be introduced is added to every class implementing this interface.

MARKER INTERFACE

Crosscutting code is often related to methods of classes which do not share a common naming characteristic. In those cases characteristic elements (such as number and types of parameters, return types, access specifiers, etc.) are alone not sufficient to separate these methods from the rest.

* * *

In some situations AspectJ's pointcut language is not sufficient to determine those target classes where an additional piece of advice should be executed. How can abstract aspects be specified where everything in the pointcut can be defined except the classes where such crosscutting occurs?

An aspect can often specify the pointcut except the classes where the pointcut should be valid. A typical sample is found, when an application is built on a modular or layered architecture. Often, interactions crossing these borders are implemented as *proxies* [6], where only the proxies contain the relevant crosscutting code (e.g. caching or synchronization). If an aspect should be woven either to the proxies or to the real class there is one problem: the proxies contain the same signatures as the original elements. In these scenarios it is desired, that everything in an aspect is specified except the classes where such crosscutting occurs. This would permit to reuse the pointcut specification and the underlying aspect in different classes and different applications.

Therefore:
Inside an *Aspect Specification* refer to a *Marker Interface* every time the functional crosscutting needs application specific information on the classes participating in the crosscutting. That interface is later connected by an application specific *Marker Setter* to application classes into which the aspect should be woven to.

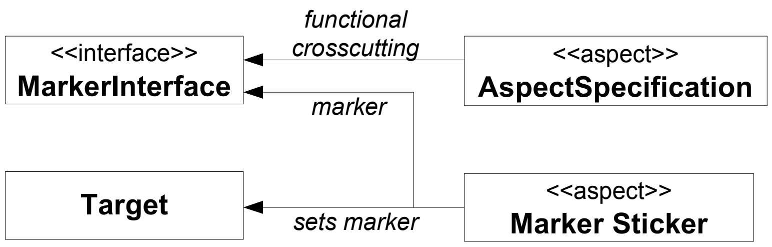


Figure 4. *Marker Interface*, *Aspect Specification* and *Marker Setter*

The aspect including the specification of pointcuts and pieces of advice is defined inside the *Aspect Specification*. Its pointcuts refer to a number of *Marker Interfaces* which are built only for the purpose of being sticked to a number of target classes. The application developer knowing to what target classes the aspect should be woven with defines a *Marker Sticker* aspect that connects *Marker Interfaces* to the target classes. Hence, MARKER INTERFACE defines a uniform way to enumerate classes which need adoption without naming them in a central place. This makes the *Aspect Specification* reusable because they do not contain application specific information. In AspectJ the *Marker Sticker* just performs an introduction of the interface to the target class. It depends on the application and the designer's decisions how many *Marker Stickers* and *Marker Interfaces* exist for a *Aspect Specification*.

Usually the aspect specification does not influence the application as long as the *Marker Interface* is not attached to the application. That means, all concrete pointcuts in the aspect specification depend on the *Marker Interface.* This also means usually, that all occurring join points are restricted to single classes or objects.

The consequences of using MARKER INTERFACE is that no internal knowledge is required in the *Aspect Specification* except the name of the *Marker Interface* and its signatures. However, a potential danger in using this idiom is that the *Aspect Specification* contains a pointcut specification which needs to match the target classes. Since the developer sticking the *Marker Interface* to a target has to guarantee that this contract is fulfilled the developer needs precise information about when a target classes is valid or not. In practice it is usually sufficient to have a non-formal description, i.e. there is usually not need for the developer to understand the whole aspect implementation.

The following example illustrates how the *Marker Interface* `MarkerInterface` is attached to a class `TargetClass`. Each call of method `foo` in a marked class is intercepted by the `AspectSpecification`.

```
public interface MarkerInterface {}
public abstract aspect AspectSpecification {
  pointcut fooCall():
    call(void *.foo()) && target(MarkerInterface);
  before(): fooCall() {
    System.out.println("Before foo was called");
  }
}
public aspect MarkerSticker {
  declare parents:
    TargetClass implements MarkerInterface;
}
```

* * *

MARKER INTERFACE is often used in conjunction with CONTAINER INTRODUCTION where the *Marker Interface* corresponds to the *Container* and the *Aspect Specification* corresponds to the *Container Loader*. Often the application of ABSTRACT POINTCUT in conjunction with COMPOSITE POINTCUT is an alternative to MARKER INTERFACE. The advantage of MARKER INTERFACE in contrast to them is that the need for internal knowledge on the *Aspect Specification* is quite limited and it is easy to connect a number of target classes. The disadvantage is that the crosscutting effect specified inside the *Aspect Specification* is limited to objects of a certain class.

ABSTRACT POINTCUT

To build reusable aspects it is not always certain what parts of the aspect can be specified entirely and what parts might vary. The most common situation is, that the behavior of an aspect (that means its pieces of advice) can be specified entirely, but it is not known when this behavior take place, that means the aspect's join points are not known at aspect definition time.

* * *

An aspect's behavior should be reused in different applications but it is unknown at aspect definition time when the aspect's behavior should take place, that means its pointcuts vary. How can such an aspect be specified to be adapted later and incrementally to concrete applications?

This situation often occurs when the aspect is rather generic and should be reused in a number of different applications or just at a number of different join points. That means the problem is not only that some parts of the pointcut are not known. Instead, it is assumed that the aspect can be used in a number of different join points which cannot be specified at aspect definition time: the aspect can be applied to a large number of pointcuts but the corresponding join point information such as method names or parameter types vary widely.

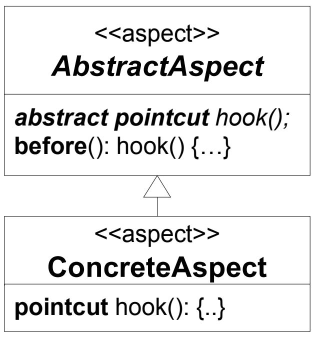


Figure 5. ***Abstract Pointcut***

Therefore:
Define the aspect behavior inside an *Abstract Aspect* where the pieces of advice refer to a *Hook Pointcut*. Whenever the aspect can be added to an application extend the aspect by a *Concrete Aspect* and define the *Hook Pointcut* as needed.

The *Abstract Aspect* contains the behavior without specifying where this behavior occurs. Also, the *Abstract Aspect* contains the abstract definition for the pointcut. The *Concrete Aspect* extends the *Abstract Aspect* and specifies the corresponding join points where the behavior should take place by defining or overriding the *Hook Pointcut*.

ABSTRACT POINTCUT permits to apply the behavior adaptation to situations that have not been foreseen at the time of the aspect definition. It should be kept in mind, that for each concrete aspect at least one new aspect instance is created even if the aspect is defined to be a singleton aspect. Variables defined in the *Abstract Aspect* are not shared between different *Concrete Aspects* except the static ones. In a straight forward implementation of an abstract pointcut in AspectJ, the parent aspect has to be abstract and the concrete aspect has to extend the concrete aspect. In [7], the application of the abstract pointcut in AspectJ is discussed in more detail.

The following code example illustrates the use of ABSTRACT POINTCUT where the concrete aspect `ConcreteAspect` defines that before each call of `foo` in class `TargetClass` a message should be printed.

```
public aspect AbstractAspect {
  abstract pointcut fooCall():
  before(): fooCall() {
    System.out.println("Before foo was called");
  }
}
public aspect ConcreteAspect extends AbstractAspect {
  pointcut fooCall():
    call(void TargetClass.foo());
}
```

* * *

ABSTRACT POINTCUT is similar to MARKER INTERFACE. Both permit to defer to binding of an aspect to the application without the need to perform destructive modifications on the aspect.

ABSTRACT POINTCUTS permits a higher level of reusability of aspects which are not only restricted to certain interfaces (like MARKER INTERFACE) since the whole pointcut definition is moved to the developer performing the connection of the aspect to the application. That means that the developer needs to know exactly for what purposes the aspect to be connected is specified for and at what join points the execution of the aspect-specific code makes sense. Also, because of the diversity of possible join points definitions, the developer performing the connection needs a highly specialized knowledge of the join point language in AspectJ: he needs to know the pointcut designators and the valid combination of those designators.

The freedom of connecting an aspect to any arbitrary join point has some further dangers: usually it is an error if a single aspect is connected for the same join point twice. Since, for each *Abstract Aspect* in ABSTRACT POINTCUT there are usually several *Concrete Aspects* the developer performing the connection has to know what join points inside an application are not already adapted by the aspect. In other words, if the developer wants to connect *Abstract Aspect* to a number of join points he has to be sure that there is not already a *Concrete Aspect* connected to one of those join points.

Because of this the use of ABSTRACT POINTCUTS is much more error-prone than e.g. MARKER INTERFACE which already prescribes the way of how the aspect can be connected (and hence reduces the complexity of pointcut specification) and which usually reduces the problem of multiple execution of the same aspect code at the same join point.

The relationship between ABSTRACT POINTCUT is on the one hand that ABSTRACT POINTCUT is often combined with POINTCUT METHOD, that means each piece of advice inside *Abstract Aspect* invokes a POINTCUT METHOD which offers some opportunities for customization of the advice behaviour.

On the other hand if all potential join points are known at aspect specification time the application of POINTCUT METHOD is more appropriate than ABSTRACT POINTCUT because its connection is less error-prone.

COMPOSITE POINTCUT

The more complex an aspect's behavior the more complex is usually its pointcut definition. Generic aspects tend to have a large pointcut definition because of the large number of different join points where additional behavior takes place.

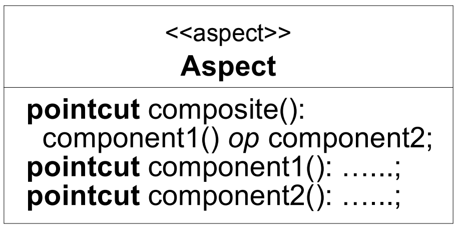
* * *

A complex pointcut definition reduces the comprehension of the aspect and reduces its ability to be reused. Whenever the pointcut definition is too complex to be understood, how can the pointcut definition be made more comprehensible and more adaptable?

If pointcuts refer to a number of join points there are often a large number of pointcut designator (like `call`, `target`, `this`, etc.) used. This reduces the comprehensibility of the pointcuts. Furthermore, if the pointcut defined inside an abstract aspect is not final, i.e. the aspect designer assumes that the pointcut could be overridden by some sub-aspects, the reusability of this pointcut is limited, because AspectJ does not permit to override just parts of a pointcut. That means, if a large pointcut definition needs to be adapted inside a sub-aspect there is only the possibility to redefine the whole pointcut.

Therefore:
Decompose the pointcut into a *Composite Pointcut* which refers to a number of *Component Pointcuts*. Each of the *Component Pointcut* represents a logically independent pointcut.

The logically independent *Component Pointcut* can be modified without knowing the complete (composite) pointcut and can be used in other aspects. The *Composite Pointcut* cannot guarantee the consistency of the pointcuts, so the developer must be aware of how to define the component pointcuts correctly. In AspectJ COMPOSITE POINTCUT can be implemented by defining a pointcut that consists of a combination of pointcuts and does not use any pointcut designators on its own. Parts of the composite pointcut consists often of pointcuts from other aspects. Pieces of advice refer only to the *Composite Pointcut* and not to its *Component Pointcuts*.



```
pointcut composite():
  component1() op component2;
pointcut component1(): ......;
pointcut component2(): ......;
```

Figure 6. *Composite Pointcut* and corresponding *Component Pointcuts*

The following example show the usage of COMPOSITE POINTCUT where each *Component Pointcut* (`fooCaller` and `barCaller`) represents a call to different methods.

```
public aspect Aspect {
  pointcut fooOrBarCaller(): fooCaller() || barCaller();
  pointcut fooCaller(): call(void *.foo());
  pointcut barCaller():call(void *.bar());
  before():fooOrBarCaller () {...}
}
```

* * *

Often COMPOSITE POINTCUT occurs in conjuction with ABSTRACT POINTCUT where one of the *Component Pointcuts* is abstract. In the same way MARKER INTERFACE often makes use of *Component Pointcuts* when the corresponding pointcut definition is too complex.

TEMPLATE ADVICE

Sometimes the behavior of a piece of advice at some join points contains some variabilities. The behavior slightly differs in those cases from application to application or even in different join points inside the same applications. That means, essential parts of an aspect's behavior is the same in different join points, whereby other parts vary from join point to join point.

* * *

How can aspects be specified if the aspect-specific behavior should differ slightly in different join points? That means, how can essential parts of a piece of advice be reused in different join points?

Whenever the code to be executed at certain join points is partly known and fixed, but contains (depending on the concrete join point) some variability, the designer has to decide how to consider such variability. In case the join points are well-known, it is possible to implement a number of different aspects for each of those different kinds of join points. The disadvantage of this approach is twofold. First, all those aspects contain redundant pointcut definitions because usually all those join points have some commonalities. Those commonalities are usually partly common signatures of methods or the location of join points inside a partly known class hierarchy. Second, those aspects contain some redundancies inside the pieces of advice. These redundancies are the fixed parts of the pieces of advice common to all aspects. The redundant pointcut specification can be reduced using a COMPOSITE POINTCUT within an abstract aspect where some COMPONENT POINTCUTS are abstract. However, the problem is still that the piece of advice contains redundant code.

Therefore:
Use the stable part of the piece of advice as a template and put the variable part (the *Primitive Operation*) into a method and combine both in an *Abstract Aspect*. The *Primitive Operation* can be overridden on demand in a *Concrete Aspect*.

In AspectJ pieces of advice cannot be refined in subaspects. That means the decision of which part of the code is fix is ultimative and cannot be revised incrementally. Instead, if there is a need to modify pieces of advice the aspect needs to be refactored. Hence, directly designing a template advice reduces the necessity for destructive modifications. The *Abstract Aspect* contains declarations of a number of *Primitive Operations* which are defined in sub-aspects. Furthermore, the abstract aspect contains the pieces of advice (called *Template Advice*) which contains the invocations of the primitive operations. Usually the pointcut referred by the *Template Advice* is an abstract and the *Concrete Aspect* defines or overrides the primitive operations of the abstract aspect and implements in that way the aspect-oriented adaptation of the target classes.

It seems questionable if the template advice is really a specific AspectJ idiom since it is very similar to the GOF-TEMPLATE METHOD [6]. However, we regard it as an specific idiom, because the consequences of using a template advice are quite different than than those of using a TEMPLATE METHOD . First, in AspectJ only abstract aspects can be extended by further aspects. That means, when the corresponding aspect is written it must be clear whether or not the aspects tends to contain a template advice or not. Using pure object-oriented language features in a language supporting late binding this question does not have to be answered. For example in Java or Smalltalk almost every method can be overridden by a subclass. That means every method inside the superclass which contains invocations of an overridden method can be considered as potential TEMPLATE METHOD. That means methods might become TEMPLATE METHOD because of an incremental modification of the class structure. Hence, the preplanning problem of design patterns as mentioned in [4] is not that significant for a

TEMPLATE METHOD. On the other hand, because of the limited possibilities of incremental aspect refinement in AspectJ this problem is more present in a template advice. Hence, the consequences of using a template advice are much more restrictive.

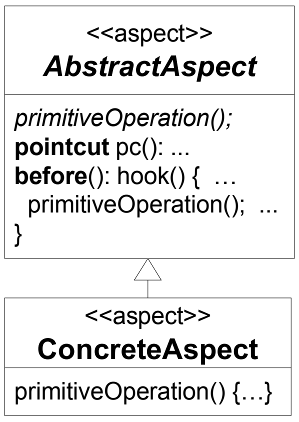


Figure 7. Template Advice

If there are more than one concrete aspect which refer to at least one common join point the developer need to determine which advice should be executed. This can be either realized by further idioms, or by an explicit use of dominate relationships between aspects. Another consequence of TEMPLATE ADVICE is that only limited knowledge of an aspect's internals required (in case the referring pointcut is not abstract): the adaptation of the aspect behavior just depends on the concrete method definition. Hence, the developer performing the aspect adaptation only needs little knowledge about the concrete pointcut or the advice internals. However, a detailed description of the contract belonging to the abstract method is needed. But the application of TEMPLATE ADVICE also means a lost access to introspective facilities: since the reflective facilities of AspectJ are just available inside an advice there is no possibility to refer inside the method to the execution context. This must be considered during the design. In case the execution context might be needed, it has to be passed as a parameter.

* * *

TEMPLATE ADVICE often occurs together with ABSTRACT POINTCUT where the *Concrete Aspect* specifies the *Hook Pointcut*. A combination of TEMPLATE ADVICE and ABSTRACT POINTCUT is often used in conjunction with COMPOSITE POINTCUT where a *Component Pointcut* is abstract. Such a combination permits to reduce the number of pointcuts where the advice can be executed.

Also, TEMPLATE ADVICE is often used in conjunction with POINTCUT METHOD and CHAINED ADVICE where in both cases the *Concrete Aspect* refines the pointcut definition. Hence, different applications of TEMPLATE ADVICE usually differ in their handling of the corresponding pointcut definitions.

POINTCUT METHOD

There are situations where a certain advice is needed whose execution depends on runtime specific elements which cannot or only with large effort expressed by the underlying pointcut language.

* * *

How can for a piece of advice be executed in dependence of runtime-specific properties?

The pointcut language of AspectJ is quite expressive. Dynamic pointcuts designators like `args(..)` permit to specify join points which are evaluated during runtime and permit in that way to specify a large variety of crosscuttings. Typical examples where dynamic pointcuts are used are the simulating dynamic dispatching on top of Java (cf. e.g. [12]). However, sometimes the decision of whether or not a corresponding advice should be executed is not that easy to specify inside a pointcut definition. Such a situation is usually given if the advice execution depends on a more complex computation or includes a invocation history of the participating objects.

The usage of *if*-pointcuts can reduce this problem. However, *if*-pointcuts are somehow ugly since they permit only to call static members functions of aspects or classes. Internal state of the aspect can not easily be accessed using *if*-pointcuts. Additionally, the reflexive information from the special variable `thisJoinPoint` is not available. Furthermore, the usage of *if*-pointcuts usually reduces the reusability of the enclosing aspect, because they are usually very specific to a small set of join points. Usually, the usage of the pointcut language seems to be inappropriate when the decision whether or not a corresponding piece of advice should be executed can be better expressed by methods than the pointcut language.

Therefore:
Define a *Candidate Pointcut* which describes all join points where potentially a piece of advice might be executed. Let the *Conditional Advice* call a method (POINTCUT METHOD) which determines whether or not the advice should be executed.

The *Candidate Pointcut* which determines all potential join points where additional behavior might take place includes of course more join points than needed to perform the aspect specific behavior. The POINTCUT METHOD is invoked from inside the advice to determine whether or not the advice should be executed. Typically the return type of a pointcut method is boolean. The *Conditional Advice* contains the behavior which might be executed at the specified join points. The additional behavior is conditional executed depending on the result of the POINTCUT METHOD.

Implementations of pointcut methods vary in a number of ways. First, usually a POINTCUT METHOD's return type is boolean. That means a POINTCUT METHOD only determines whether or not the additional behavior specified inside an *Conditional Advice* should be executed. On the other hand, a POINTCUT METHODS can also include just any computation whereby the conditional execution of the advice depends on it's result (and any other context information). That means the decision whether or not the piece of advice should be executed not only depends on the pointcut method itself.

Another important issue is how the computation of the pointcut method depends on the execution context of the application. Usually context information is directly passed by the piece of advice to the POINTCUT METHOD. That means the referring pointcut either passes some parameters to the advice or the advice extracts context information using the introspection capabilities of AspectJ like `thisJoinPoint` or `thisStaticJoinPoint`. Another possibility is, that the aspect itself has a state that is set by the application's execution context.

The POINTCUT METHOD can decide because of this state whether the piece of advice should be executed or not.

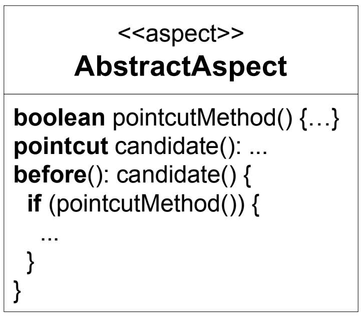


Figure 8. POINTCUT METHOD with *Candidate Pointcut* and *Conditional Advice*

An advantage of using a POINTCUT METHOD is its adaptability by aspects. In case the *Pointcut Method*[2] is a *Primitive Operation* in a TEMPLATE ADVICE (usually POINTCUT METHOD are used in that way) it is possible to specify further pieces of advice which refine the *Pointcut Method* outside the aspect hierarchy. That means, the condition whether or not a piece of advice should be executed can be modified incrementally. In case the pointcut is hard-coded by using the pointcut language such an extension is not that easy. It assumes a corresponding underlying architecture or *rules of thumb* like discussed in [7].

The nice property of POINTCUT METHOD based on TEMPLATE ADVICE is that the user which specifies the *Pointcut Method* does not need to understand the implementation of the whole pointcut. He just needs an acknowledgement by the aspect developer that at least all join points he is interested in are specified by the pointcut. However, to determine whether or not the piece of advice should be executed, the *Pointcut Method* needs some inputs. This might be for example property files, or (which is more usual) parameters which are passed from the pointcut to the piece of advice and then to the *Pointcut Method.* In case the *Conditional Advice* is an after or around advice, it is necessary to specify any default behavior. Around advice usually call `proceed`, while after advice usually pass the incoming return value. When specifying the pointcuts it is not necessary to understand all internals of the piece of advice. Usually, it is enough to have a description in natural language what kinds of join points can be handled by the advice and what kind of impact the piece of advice has on the join point.

* * *

As mentioned above, POINTCUT METHOD is usually used in conjunction with TEMPLATE ADVICE. POINTCUT METHOD idiom is similar to COMPOSITE POINTCUT (see also [5]). Both divide the pointcut into a stable and variable part (usually COMPOSITE POINTCUT it used in conjunction with a inheritance relationship between aspects). The difference between them is, that for adapting a *Composite Pointcut*[3] the application of an inheritance relationship between aspects is necessary. This also implies that a COMPOSITE POINTCUT has some preplanned variabilities (which are usually the *Component Pointcuts*). A POINTCUT METHOD does not directly depend on an inheritance relationship. The refinement might be either achieved via inheritance or by an advice. In the first case a POINTCUT METHOD plays the role of an *Primitive Operation* inside a TEMPLATE ADVICE. In the latter case, a POINTCUT METHOD is often refined by a CHAINED ADVICE.

[2] Since Pointcut Method is also a role in the POINTCUT METHOD idiom we use italic to refer to the role.
[3] Like above, Composite Pointcut is a name of an idiom and a role within that idiom. Hence, we use italic when we refer to the role.

CHAINED ADVICE

Often, at certain join points a number of different actions have to take place. That means there are a number of aspects which have a number of join points in common. However, since all actions are specified in different aspects they are logical not closely related to each other.

* * *

Whenever pieces of advice coming from different aspects are related to the same join points, the question is how each of them could be expressed in its own module. How can independent pieces of advice for the same join points be expressed in a modular style?

Typically the actions to be performed are writing a message to a logging framework, creating of a proxy object and informing the persistence framework about the existence of a new object.

Object-orientation already provides the modular extension of classes via inheritance. However this does not really solve the adaptation problem: the adaptation is achieved by inheritance and that implies a new class has to be created which overrides and adapts a known one. Furthermore, it must be guaranteed that the request for creating new objects must be redirected to the new class in certain situations. If (for the original classes) no creational patterns (like the ones in [6]) where used such a task tends to be error-prone and the resulting design is usually unacceptable. In such cases, where an application's behavior at (at least) one join point depends on a number of concerns those concerns are usually not orthogonal, but interact in some way. That means, the new behavior should be modularized in separate aspects, but the relationship between such non-orthogonal concerns must be considered.

Therefore:
Define a *Abstract Chain* aspect which contains the definition off all join points inside its *Anchor Pointcut* where additional behavior should take place and define a number of *Chain Element* aspects which extends the *Abstract Chain* and define additional pieces of *Chained Advice* for the *Anchor Pointcut*.

The *Anchor Pointcut* is used by every advice within the chain. We used the term *Anchor Pointcut*, because each *Chain Element* is anchored at each join points part of this pointcut definition. Each chain defined by each piece of advice may have a predefined order. Usually each advice contains a mechanism to redirect the execution to a different advice.

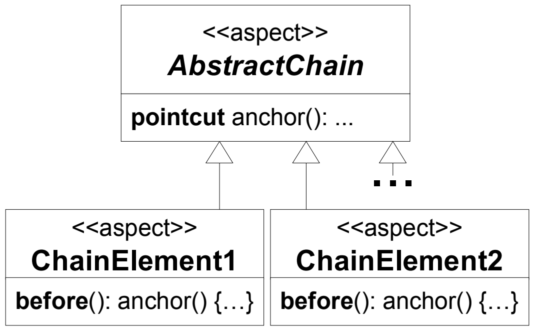


Figure 9. CHAINED ADVICE **with *Anchor Pointcut* inside the *Abstract Chain* and *Chain Elements*, each defining its own *Chained Advice* referring the *Anchor Pointcut*.**

In contrast to the previous mentioned idioms, a CHAINED ADVICE comes with a number of different implementations. On the one hand it is not necessary that the *Anchor Pointcut* is inherited from a super-aspect. Instead, we found either the usage of static pointcuts or even more complex aspect hierarchies than illustrated in figure 9. We found implementations where the CHAINED ADVICE was realized by an ordinary `proceed`-call, in other cases we found more complex pointcut definitions (that means each chain element offers a join point used by the following chain element). The latter one is implemented whenever the *Chain Elements* should not be defined entirely separate from each other. This is usually the case, when the order of the execution is important for the *Chain Elements*. Furthermore, such an implementation has the advantage that the order is implicitly given by the pointcut definition: the first elements to be executed is the one that only refers to the *Anchor Pointcut*, the second element is the one which refers to a join point of the first one, etc. Such an implicit ordering has the advantage that no destructive modifications inside the aspect specification has to be performed (i.e. no additional *dominates*-relationships have to be inserted). On the other hand, such an implementation is quite complex and it is seldom the case that chain elements should know each other.

Also, in many cases the execution of each *Chained Advice*[4] is mutually exclusive, than means at most one *Chained Advice* is executed. But there are situations where more than one chain element is executed. What kinds of *Chained Advice* should be used depends on the concrete situation. The way how the mutually exclusive pieces of advice are realized differs in different applications. On the hand POINTCUT METHODS were used, in other cases ordinary advice in combination with COMPOSITE POINTCUTS [5] were used. Both implementations have their pro and cons. The advantage of the first approach is that aspects do not need to have any knowledge about each other, i.e. their implementations do not depend on each other. But this also means that the advice execution order has to be controlled in some way. The latter approach assumes an explicit dependency of each piece of advice.

The consequences of using a CHAINED ADVICE are that each *Chain Element* represents a certain behavior coming from different concerns within its own module. These *Chain Elements* can be combined independent of each other, that means it is possible to add more elements to the chain without destructively modifying the existing chain. The definition of a mechanisms for passing parameters between the chain elements lies as in the responsibility from one *Chain Element* to another as well as in the definition of the *Abstract Chain*. Furthermore, in case the CHAINED ADVICE is used on top of a POINTCUT METHOD using around advice, a default behavior has to be specified inside the *Chain Element*.

* * *

Chained advice make often use of POINTCUT METHOD to determine whether or not a chain element should be executed. Furthermore, CHAINED ADVICE often make use of COMPOSITE POINTCUTS to reduce redundant pointcut definitions.

[4] Since Chained Advice is also a role in the CHAINED ADVICE idiom we use italic to refer to the role.

ADVICED FACTORY METHOD

Whenever the instantiation of certain objects depends on specific aspects which might vary from application to application or the execution context of an application a mechanism is needed to determine which object should be created without performing destructive modifications on those classes participating in the creational process.

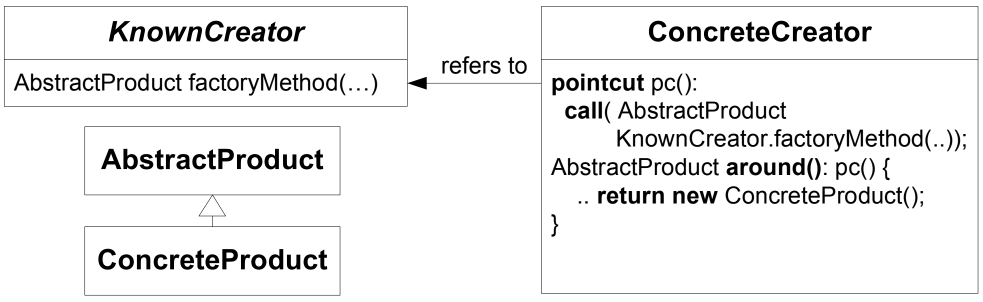
* * *

How can different aspects in different installations of aspect-oriented software participate in the creational process of an object?

Whenever the creation of objects depends on certain aspects (and there might be more than one aspect) and such object creation differs in different applications or different execution contexts it is usually not appropriate only to intercept the object creation using a pointcut to the constructor and then redirecting the creation using an around advice. The problem in such a context is usually the restriction that around advice need to return the same type than its join points.

Therefore:
Create a *Factory Method* for each object creation which returns an *Abstract Product*. Create a *Concrete Creator* aspect which defines an pointcut on the *Factory Method*. Define a *Factory Advice* that performs the object instantiation.

The relationship between an Adviced Factory Method and the usual application of advice can be seen like the relationship between the Factory Method [6] and Template Method [6]: although both are similar in their relationship of hook and template their differ mainly in the way their intention.



```
pointcut pc():
  call( AbstractProduct
        KnownCreator.factoryMethod(..));
AbstractProduct around(): pc() {
   .. return new ConcreteProduct();
}
```

Figure 10. ADVICED CREATION METHOD

The consequences of using an Adviced Factory Method are a deferred object instantiation, that means the object instantiation is no longer hard coded inside the object structure, but moved to the aspect definition. That means the instantiating aspect must be woven to the application to guarantee its correctness. Since a factory method returns an object, a default behavior need to be specified inside the *Factory Method* which is refined by the *Factory Advice*. Usually, the advice overrides the whole behavior specified there (i.e. the advice does not call `proceed()`). But there are situations where the *Factory Method* contains some meaningful code and the aspect code is just executed in "special situations". The application of Adviced Factory Method achieves a higher level of independence between the modules performing the instantiation: the Adviced Factory Method permits to exchange the object creation process without performing destructive modifications within the object structure.

* * *

Adviced Factory Method is often used as Chained Advice in cases where the object to be instantiated depends on some execution context.

2 Example

The example is separated and simplified from the Sirius EOS System, a full blown service level management system. It uses software components to represent physical and logical service resources like a logical subnet, a router or a NT-server.

These software components must implement an observer contract. This observer contract may be used by graphical controls and triggers, which start a partial recalculation of some service state after each state modification of the observed network.

These software components are typically developed long after the framework has been architected and implemented. It illustrates another solution of the well known problem to create new classes which are unknown at the time when the code responsible for the creation is written. This example avoids the violation of the *need to know* rule, where the framework must not be aware of components and still works with two layers of dependency. Existing object oriented component based frameworks do require three layers of dependencies. The lowest layer is the framework, which does not depend from anything. The next layer the components can only depend on themselves and on the services and contracts provided by the framework. The last layer the component assembly layer can depend on itself, the components and the framework. This layer can be realized in code, which deploys the final configuration class for a dense set of patterns. But more often it is realized with configuration files in combination with reflexive code in the framework. Sometimes a combination of both of this approaches is used. OO-creational patterns do allow an efficient implementation of these three layered frameworks however they can not eliminate the third layer. Avoiding the third layer offers great benefits for a very strict component oriented configuration management and delegates quite a lot of quality assurance tasks to the compiler by avoiding reflective code and deployment descriptors.

Because the framework must not violate the *need to know* principle, the framework cannot name directly the objects which must be created. Furthermore, we do not know what further aspects are interested in the object creation. Hence the problem corresponds to the problem description in ADVICED FACTORY METHOD . We define an ADVICED FACTORY METHOD in our framework, which has two parameters, each defining the object which is about to be created (in our example we identified the participating entities by a string representing the logical name and a string representing the entities location). Clients, which need a new instance of a certain entity inside the framework call the static method `createEntity` in `EntityCreator`. Since there are no default entities in the system, the default behavior of the class is returning `null`.

```
public class EntityCreator {
  public static Entity
  createEntity(String name, String location) {
    return null;
  }
}
```

The situation for the *Concrete Creator* is, that each component installed into the system should be able to specify how it should be created. So, we have a number of *Concrete Creators*, one defined for each entity class in the system. That means, each *Concrete Creator* refers to the same set of join points and decides on the passed parameters from the ADVICED FACTORY METHOD whether it should be instantiated or not. This matches the problem description in CHAINED ADVICE: there are a number of aspects all having the same pointcut definition. Furthermore, each *Concrete Creator* depends on runtime-specific information (the parameters passed to `createEntity`). That means, we have a similar problem as decribed in the POINTCUT METHOD. Hence, the *Concrete Creators* are implemented as a combination of CHAINED ADVICE and POINTCUT METHOD.. The CHAINED ADVICE is specified by defining the pointcut which refers to the ADVICED FACTORY METHOD: the *Anchor Pointcut* is defined inside an aspect `AbstractEntityCreator` which needs to be specified by each *Concrete Creator*:

```
public abstract aspect AbstractEntityCreator {
  pointcut create (String p1,String p2):
    execution(
      static Entity
        EntityCreator.createEntity(String,String )
    ) && args(p1,p2);
  ...
}
```

Since the creation of entities depends first on runtime-specific elements we define a concrete piece of advice inside this aspect which invokes the *Pointcut Method* `accept` which determines whether or not the advice should be executed. The default behavior of the piece of advice is to call `proceed()`.

```
public abstract aspect AbstractEntityCreator {
  pointcut create(String p1,String p2)...
  abstract boolean accept(String p1,String p2);
  Entity around(String p1,String p2): create (p1,p2) {
        if (accept(p1,p2))
            ...
        else return proceed(p1,p2);
  }
  ...
}
```

Furthermore, we have to define what should happen in case the piece of advice should be executed. Since this code is specific to the *Concrete Creator* extending `AbstractEntityCreator` that means, it varies from aspect to aspect how and what concrete class should be instantiated, we apply the TEMPLATE ADVICE idiom: the above specified piece of advice calls an abstract method `createMethod` which needs to be defined in every concrete aspect:

```
public abstract aspect AbstractEntityCreator {
  ...
  abstract Entity createMethod(String p1, String p2);
  Entity around(String p1,String p2): create (p1,p2) {
        if (...)
            return createMethod(p1,p2);
        else ...;
  }
  ...
}
```

Hence, every concrete entity installed in the system needs a corresponding *Concrete Creator* which extends `AbstractEntityCreator` and defines the methods `createmethod` and `accept`. Following code sample contains how the resulting code including a *Concrete Creator* for the entity `Router` may look like.

```
public aspect RouterCreator extends AbstractEntityCreator{
  protected boolean accept(String param1,String param2){
    return ("Router".equals(param1));
  }
  protected Entity createMethod(String p1,String param2){
    return new Router(param2);
  }
  declare parents: Router implements
          ChangeNotificationAspect.NotificationHelper;
}
```

Another point in the example is, that an appropriate implementation of the notification service has to be provided. Here, an appropriate implementation of the Observer design pattern [6] is sufficient. That means, all entities to be instantiated should provide an interface that different observers can be attached and detached. Furthermore, every time a subject changes its listeners should be informed. The methods for attaching and detaching observers represent extrinsic features of the classes they are attached to. Furthermore, since a number of new entities can be added to the framework, these features should be encapsulated and later applied to different target classes.

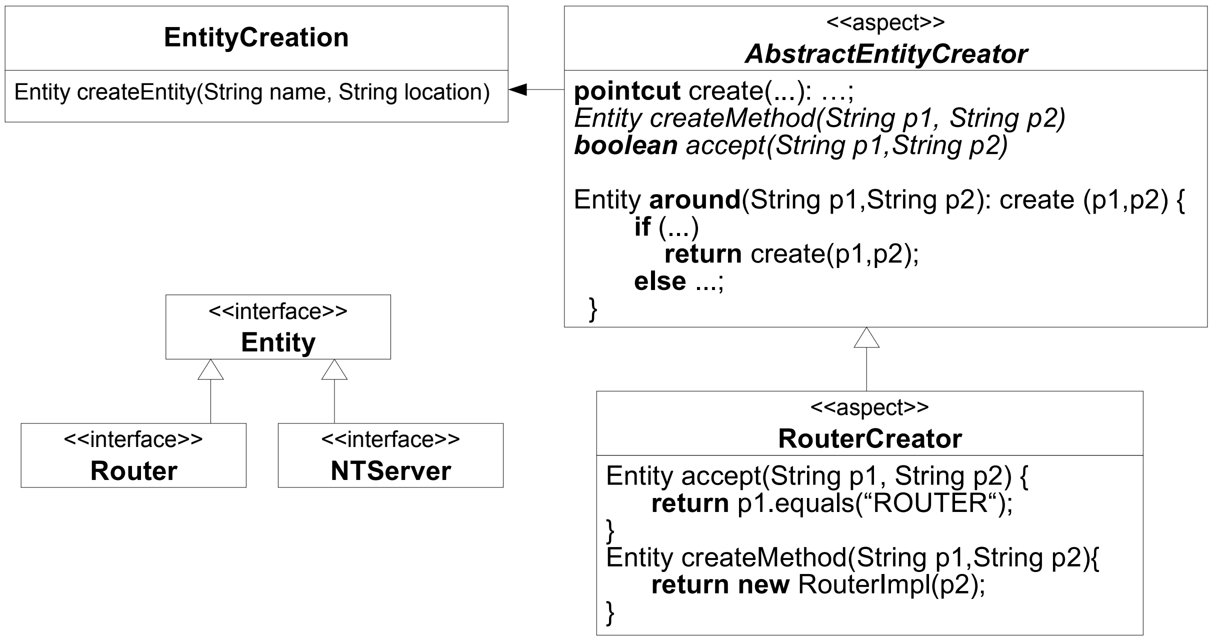

EntityCreation
Entity createEntity(String name, String location)
<<aspect>>
AbstractEntityCreator
pointcut create(...): ...;
Entity createMethod(String p1, String p2)
boolean accept(String p1,String p2)
Entity around(String p1,String p2): create (p1,p2) {
if (...)
return create(p1,p2);
else ...;
}
<<interface>>
Entity
<<interface>>
Router
<<interface>>
NTServer
<<aspect>>
RouterCreator
Entity accept(String p1, String p2) {
return p1.equals("ROUTER");
}
Entity createMethod(String p1,String p2){
return new RouterImpl(p2);
}

Figure 11. Implementation of ADVICED CREATIONAL METHOD based on POINTCUT METHOD and TEMPLATE ADVICE

The context and the problem perfectly matches the CONTAINER INTRODUCTION. Hence, we introduce the methods `addlistener`, `removeListener`, `notifyListeners` and the field `list` which stores all listeners to a container and later connect this container to the target classes. In this implementation `ChangeNotificationAspect` corresponds to the *Container Loader* and the `NotificationHelper` corresponds to the Container. Since a common type is needed for attaching, detaching and notifying listeners, we also define a corresponding `ChangeNotificationListener` interface.

The specification of when the listeners should be notified can be almost entirely specified: every time right after a method in a `ChangeNotificationProvider` is executed. Furthermore, for reasons of efficiency *inner calls*, that means the invocation of a methods from a `ChangeNotificationProvider` to itself should be neglected and even more important in order to avoid an endless loop, the callback from the observing object should be neglected[5]. The here described property is a description of crosscutting whereby the corresponding pointcut can be specified entirely except that it cannot refer to a concrete target class.

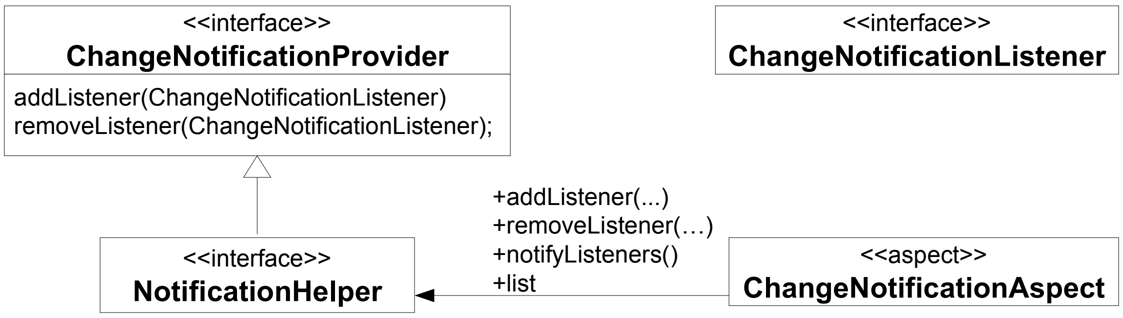


Figure 12. Applied Container Introduction

Hence, the problem corresponds to the problem from the MARKER INTERFACE idiom. So, we specify a pointcut which refers to a MARKER INTERFACE. The MARKER INTERFACE is the interface which is added to a target class to provide this class with the ability to notify its listeners. That means we use the above defined Container `NotificationHelper` also as a MARKER INTERFACE. Furthermore, since the crosscutting features belong to the same concern as described in CONTAINER INTRODUCTION, we use the `ChangeNotificationAspect` as the *Aspect Specification*. The corresponding pointcut definition look like this:

```
pointcut methodExecution(NotificationHelper helper):
  execution(public * NotificationHelper+.*(..)) &&
  this(helper) && !within(ChangeNotificationAspect) &&
    !cflow(execution(*   NotificationHelper.notifyListeners
())
  && this(helper));
```

The pointcut refers to all join points where a method in a subclass of the MARKER INTERFACE `NotificationHelper` is executed (except callbacks from the notification call expressed by cflow-construct). Additionally, methods defined in the aspect `ChangeNotificationAspect` are excluded. The pointcut parameter is used by the advice to invoke the corresponding `notifyListeners()` method.

[5] Often read only calls are removed from the pointcut, too. This is often done with quite complex description of reading methods. These descriptions are often added to the pointcut by applying the pattern COMPOSITE POINTCUT. We do not consider the read only calls for increased simplicity of the example.

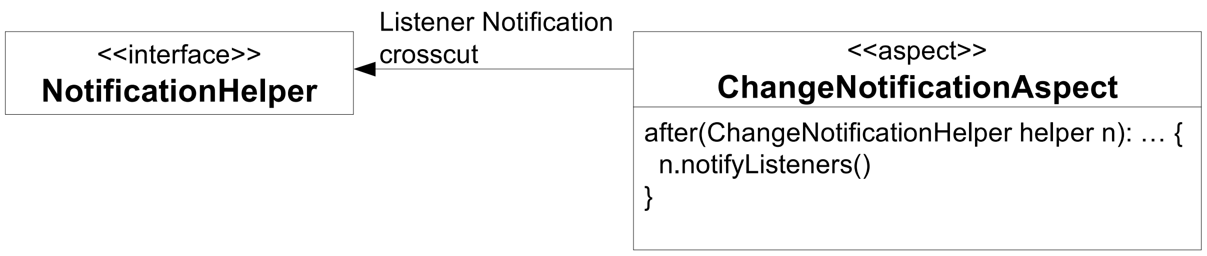


Figure 13. Marker Interface application

Until now we did not determine how to connect the *Container* (which is also a MARKER INTERFACE). In our framework each component installed to the framework needs to connect itself to the CONTAINER. Since each new component needs to be connected to a container and each component already comes with its *Concrete Creator* we define the CONNECTOR inside the *Concrete Creator*. For example the code line in the sample `RouterCreator` above shows how inside the *Concrete Creator* the connection between the target class `Router` and the Container `NotificationHelper` is defined.

```
declare parents: Router implements NotificationHelper;
```

The example above showed one important part of the above mentioned idioms: usually a number of the proposed idioms are used at the same time in the same location (class). Furthermore it showed that container introductions are exhaustively used during the application development in AspectJ.

Acknowledgement

We would like to say thanks to our shepherd Mr. Markus Voelter, for his valuable comments and for his efforts, which helped us to improve the paper.

References

[1] Alexander, C.; Ishikawa, S.; Silverstein, M.; Jakobson, M; Fiksdahl-King, I.; Angel, S.: *A Pattern Language – Towns, Buildings*, Construction. Oxford Univ. Press, 1977.

[2] AspectJ-Homepage, `http://www.eclipse.org/aspectj/`

[3] Bracha, G.; Cook, W.: *Mixin-based Inheritance.* In: Norman Meyrowitz (Ed.). OOPSLA / ECOOP'90 Conference Proceedings, ACM SIGPLAN Notices 25, 10, pp. 303-311.

[4] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming: Methods Tools and Applications*, Addison-Wesley, 2000.

[5] Clifton, C.; Leavens, G.; Chambers, C.; Millstein, T.: *MultiJava: Modular open classes and symmetric multiple dispatch for Java*, In Proc. of OOPSLA 2000, pp. 130–146

[6] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Design*, Addison-Wesley, 1995.

[7] Hanenberg, S.; Unland, R.: *Using And Reusing Aspects in AspectJ.* Workshop on Advanced Separation of Concerns, OOPSLA, 2001.

[8] 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.

[9] Hanenberg, S.; Unland, R.: *Specifying Aspect-Oriented Design Constraints in AspectJ*, Workshop on Tools for Aspect-Oriented Software Development at OOPSLA, Seattle, November 4, 2002.

[10] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwing, J. *Aspect-Oriented Programming*. Proceedings of ECOOP, 1997.

[11] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W.: *An Overview of AspectJ.* Proceedings of ECOOP 2001.

[12] Schmidmeier, A.; Hanenberg, S.; Unland, R.: *Implementing Known Concepts in AspectJ*, 3rd Workshop on Aspect-Oriented Software Development of the SIG Object-Oriented Software Development of the German Informatics Society, Essen, Germany, March 4-5, 2003.

APPENDIX

A-1 Aspect-Oriented Programming In 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.

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, accessing a variable, handling an exception, initializing an object or class.

Pointcuts

A *pointcut* selects a collection of join points. To specify pointcuts AspectJ provides a number of pointcut designator like `call`, `args`, `this`, `target`, `within` and `initialization`. 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+);
```

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

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 pointcuts it is meant to be executed, i.e. every piece of advice refers to one or more pointcuts. 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. 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.

A-1.3 Introductions

Introductions permit to add new members to classes (which is similar to open classes [5]) 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.

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 e.g. participates in a certain call-join point.

```
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.

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...
```