Design Dimensions of Aspect-Oriented Systems

Dissertation

vorgelegt dem Fachbereich Wirtschaftswissenschaften,
der Universität Duisburg-Essen
(Campus Essen)

von Stefan Hanenberg, geboren in Oberhausen

zur Erlangung des Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Gutachter:
Prof. Dr. Rainer Unland
Prof. Dr. Klaus Pohl

Oktober, 2005

Tag der mündlichen Prüfung: 27. Januar 2006

für Monika Hanenberg (1945-2002)

Ombra mai fu
Di Vegetabile,
Care ed amabile
Soave piu.

ABSTRACT

Aspect-oriented software development is a promising approach that addresses the problem of modularizing crosscutting concerns – concerns whose implementation cannot be modularized due to the set of abstractions provided by the underlying programming language as well as due to the set of decomposition criteria applied to the underlying problem. Thereto, aspect-oriented systems provide additional abstractions in order to increase the modularity of software systems and consequently to increase the system's readability, understandability, maintainability and reusability.

Nowadays, there are already a number of so-called aspect-oriented systems available that supply a set of new constructs to address the given problem. However, it is not clear what the criteria for a system are in order to be called aspect-oriented and what commonalities and differences exist among different aspect-oriented systems. This also implies that for a given crosscutting concern it cannot be determined on an abstract level what system is able to modularize such a concern in an appropriate way.

This thesis describes the characteristics of aspect-oriented systems by so-called design dimensions – orthogonal, conceptual views on the core ingredients of aspect-oriented systems that determine the underlying implementation. It is shown that by means of such abstract design dimensions it is possible to estimate the appropriateness of a system with respect to modularize a given crosscutting concern.

ACKNOWLEDGEMENT

This thesis would have never been written without the continuous support of many colleagues and friends.

I am grateful to Professor Dr. Rainer Unland for his supervision, for the environment he provided, and especially for his support in those situations, where I was no longer sure about how to go on with the thesis.

Dominik Stein spent with me countless hours, days and month of discussion. I owe him much for the enumerable and valuable comments he gave that helped me improving (and finishing) the thesis. It is a great joy to work with him.

I would like to thank Robert Hirschfeld for many, many discussions on aspect-orientation and for his patience when I tried to understand the implementation details of AspectS. The work with him influenced many of my views on software construction.

The close collaboration with Arno Schmidmeier helped me a lot when trying to figure out what the core ingredients of aspect-orientation are. He never hesitated to share his detailed knowledge of aspect-orientation and to share his experience from industrial projects.

Jeff Gray's comments significantly improved my thesis. I feel honored that he spent so much time reading my thesis, correcting faults, and giving advice.

Furthermore, I would like to thank many people for their time and support, for sharing their ideas, for shepherding, and for giving advice: Mohammed Al-Mansari, Frank Büscher, Jim Coplien, Pascal Costanza, Kris De Volder, Rémi Douence, Oliver Horn, Günter Kniesel, James Noble, Christian Oberschulte, Andrea Schauerhuber, Maximilian Störzer, Markus Völter, Uwe Zdun, and Renat Zubairov.

Finally, I would like to thank Melanie and Ben for their love and patience. Without you, this thesis could have never been written.

CONTENT

1

INTRODUCTION

1.1 Decomposition, Composition, and Abstraction in Software Construction

Nowadays, software systems become larger and larger and the functionality provided by single software systems becomes more generous. Consequently, approaches are needed in order to make software systems understandable, manageable, and maintainable.

The term **decomposition** (cf. e.g. [Parn72], [Cour85]) describes the result (often as well as the process) of dividing large problems into smaller parts for reducing the complexity. The benefit of reducing the system's complexity is caused by the ability of being manageable by humans. The purpose is to reduce the *psychological complexity* [BCK98, p. 39] inherent in dealing with large software systems in order to make the system *intellectually manageable* (cf. [SmSm77], p. 105). The result is that it is easier to handle a subsystem instead of the whole system because one has not to think about all aspects of the system when the system is created. This argument is mainly directed towards the creation of new systems. Furthermore, subsystems become more comprehensible because one only partially needs to understand the solution instead of the whole one. Hence, this argument is more directed to the maintenance of existing systems. The term **composition** simply states the way of how the decomposed subsystems are being combined.

So far, the necessity for dividing a large problem into a smaller problem is plausible. However, the main question is how to decompose a system into a number of subsystems: It is necessary to determine what mental rules or guidelines for decomposing a system are to be applied to a given problem.

How a complex system is decomposed into smaller subsystems depends on the underlying **decomposition criteria** as described by D. Parnas [Parn72]. He argues that the underlying criterion of the modularization is directly responsible for the quality of the resulting software. This quality implies on the one hand the suitability of dividing the problem into sub-problems and the corresponding reduction of the complexity. On the other hand, it refers to each module's stability with respect to future changes: Additional requirements coming later to the software might lead to changes in numerous modules. However, the question is what criteria should be applied to decompose a system and what collection of criteria might be used in combination.

The underlying philosophy of separating a large problem into simpler parts in the context of software construction is usually referred to as a **paradigm** [Floy79]. This (philosophical) term was first popularized in [Kuhn70]. However, the term is often used in different ways in the area of software engineering. Here, this thesis agrees with Jim Coplien who defines a (software) paradigm as something that "*encodes rules, tools, and conventions to partition the world into pieces that can be understood individually*" [Copl98, p. 107]. A paradigm provides a set of criteria that (when applied to a system) permit to decompose complex systems into a set of sub-systems where each one has a reduced complexity.

A complex system that is decomposed according to an underlying paradigm is usually organized as a hierarchy, whereby the system "*is composed of interrelated subsystems that have in their turn their own systems, and so on* […]" [Cour85, p. 596]. Consequently, the elements of each subsystem depend on the criteria applied to the enclosing system. Hence, when the subsystem is decomposed any further only the parts of the problem space left by the enclosing decomposition criteria influence the resulting sub-subsystems. So, an applied decomposition criterion indirectly determines parts of the properties of each resulting subsystem: The aspects that are already left out by a system's decomposition criterion are no longer available in the module's subsystems. According to [TOHS99], the term **dominant decomposition criterion** is being used to describe this implicit relationship between a system and its subsystems: A decomposition criterion is dominant since it implicitly determines the elements that do no longer need to be considered in the subsystems. Consequently, the subsystems decomposition criteria are dominated by the super systems' criteria.

Although paradigms offer collections of criteria for decomposing systems, it is still up to the developer to select an appropriate one: The process of choosing the right criteria from the paradigm for a given problem and applying it to the problem is still a creative process. The main problem in this context is to determine good decompositions: Decomposition criteria are required that are reasonable in the sense that the resulting decomposition is understandable, manageable, and maintainable.

Paradigms like **object-orientation** (cf. e.g. [Wegn87] for an explanation of the core ingredients of object-orientation) provide a number of **abstractions** like object, field, method, class, subclassing, subtyping, aggregation or delegation in order to handle a system's complexity. On programming language level, corresponding constructs are provided in order to map the conceptual model created from the object-oriented paradigm to corresponding pieces of code.

However, although a given paradigm provides a set of abstractions the problem of the dominant decomposition criterion still exists. Once a certain element is identified as a class, the way of how its ingredients are to be decomposed is restricted. Consequently, if a decomposition criterion is applied quite early and in a rather impractical way, the underlying paradigm cannot help to organize the software in a better way.

1.2 Aspect-Oriented Software Development

1.2.1 Separation of Concerns

The claim for the need of an appropriate representation of a complex software system is also described by the term **separation of concerns** by E. Dijkstra (cf. [Dijk76], p. 210)[1]:

> *To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. The other aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused. This is what I mean by focusing one's attention upon a certain aspect; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extend that they are irrelevant for the current topic. Such a separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of.*
>
> *I usually refer to it as "a separation of concerns"* [...].

Separation of concerns describes the process of dividing large problems into smaller parts for reducing the complexity. In that way, the phrase is comparable to the previously mentioned term decomposition.

However, Dijkstra also emphasizes a new point of view on decomposition: Instead of searching for certain (and possibly artificial) decomposition criteria, the phrase separation of concerns is considered to be somehow natural since it is regarded as a characteristic of intelligent thinking. Consequently, a problem is decomposed is rather a natural way: A system is decomposed into a number of elements whereby the decomposition corresponds to the understanding of the system from the developer's point of view. The decomposition according to separation of concerns is not the result of an artificial construction and selection of decomposition criteria. Instead, the decomposition comes from a natural understanding of the problem.

From the developer's point of view, the phrase separation of concerns urges to think about the concerns that influence the software to be created and to keep them in their own modules. Typical examples (that are often mentioned in the aspect-oriented literature) are **business logic** (cf. e.g. [CDHJ]), **persistency** (cf. e.g. [RaCh03]), **security** (cf. e.g. [HZU05]) or **transaction handling** (cf. e.g. [ClBa05]). The idea still relies on a dominant decomposition criterion that typically corresponds to the underlying business objects and business logic. The other (non-dominating) decomposition criteria come from non-functional requirements.

However, it turns out that this way of decomposing a problem into separate modules does not satisfy the idea that all modules should reflect the concerns they represent: It turns out to be even impossible to modularize code coming from some of the non-dominating decomposition criteria. Such concerns are called **crosscutting concerns** and are discussed in the following section.

[1] In fact, the phrase separation of concerns is being used in countless papers in the aspect-oriented literature as the key motivation for the aspect-oriented approach (cf. e.g. [FECA04]).

1.2.2 Crosscutting Concerns and Crosscutting Code

The term **crosscutting** was first introduced in [KLM+97]. It describes the nature of elements resulting from the underlying decomposition [KLM+97, p. 227]:

In general, whenever two properties being programmed must compose differently and yet are to be coordinated, we say that they crosscut each other.

Although not mentioned explicitly in [KLM+97] the term **crosscutting concern** is being used to describe the same problematic (cf. for example [ClBa05] among many others). A crosscutting concern is a concern (or its implementation) that cannot be well modularized. The lack of modularization is not necessarily inherent to this concern but comes from decomposition criteria applied to those modules the concern is related to. Consequently, crosscutting concerns result in code that is distributed over a number of modules and typically induces highly redundant code. This thesis uses the term **crosscutting code** to describe the code implementing the crosscutting concern, cf. e.g. [HaUn02a].

A typical problem that results from crosscutting code is the problem of the **co-evolution of design and code** (see [Wuyt01]): If for example in an object-oriented application a class (that is created because of an underlying functional decomposition criterion) evolves, it is potentially necessary to consider also all crosscutting concerns that touch this class. If inheritance is the underlying mechanism for implementing this evolution, this problem is also known as **inheritance anomaly** (see [MaYo93] for a detailed discussion).

In order to give an intuitive understanding of the terms crosscutting concern and crosscutting code, the following subsections illustrate two examples of crosscutting concerns and their corresponding code representations resulting from typical design pattern implementations. According to [HaKi02], implementations of **design patterns** [GHJV95] are typical examples of crosscutting code. The reason for this is quite simple: Design patterns are elements of **reusable design**. In case the design elements occur on the code level in the same or in a similar way such elements cause code redundancies. In addition to the illustration of crosscutting code, the problems caused by such crosscutting characteristics are being discussed in the following.

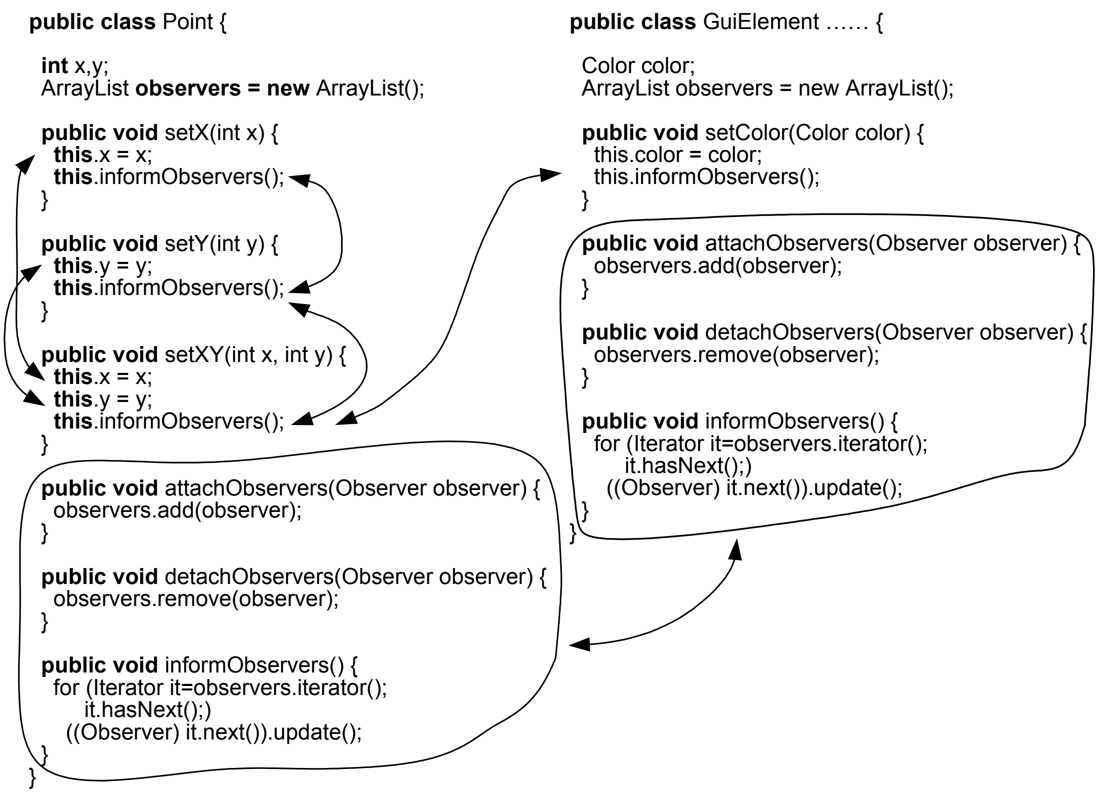
1.2.2.1 Example 1: Observer Implementations

Implementations of the **observer design pattern** [GHJV95] are commonly accepted to be examples of crosscutting concerns and they are probably the most often mentioned examples in the aspect-oriented literature (see for example [GyBr03, HaUn02a, SHU02, VeHe03] among many others). The intention of the observer pattern is to separate the object that changes its state (called **subject**) from the objects that depend on this object's state (called **observers**) and that need to be updated whenever a state change occurs. The observer pattern is often applied to the design and implementation of graphical user interfaces[2].

In [GHJV95] a typical implementation of the observer pattern is proposed that occurs often in almost the same form. The subject maintains a collection of observers that need to be informed whenever the subject changes. Furthermore, the subject has a

[2] See also the **publish-subscriber pattern** as described in [BMRS96].

method `attachObserver` and `detachObserver` (each receiving an observer as a parameter) for adding and removing observers to and from the subject's observer list. In addition, subjects have a method that informs its observers about state changes (`informObservers`). Every time a subject' state changes, the corresponding method `informObservers` is being invoked that in turn triggers the notification of each observer by calling the corresponding method `update`.



```
public class Point {

  int x,y;
  ArrayList observers = new ArrayList();

  public void setX(int x) {
    this.x = x;
    this.informObservers();
  }

  public void setY(int y) {
    this.y = y;
    this.informObservers();
  }

  public void setXY(int x, int y) {
    this.x = x;
    this.y = y;
    this.informObservers();
  }
  public void attachObservers(Observer observer) {
    observers.add(observer);
  }

  public void detachObservers(Observer observer) {
    observers.remove(observer);
  }

  public void informObservers() {
    for (Iterator it=observers.iterator();
        it.hasNext();)
      ((Observer) it.next()).update();
  }
}
```

```
public class GuiElement ...... {

  Color color;
  ArrayList observers = new ArrayList();

  public void setColor(Color color) {
    this.color = color;
    this.informObservers();
  }
  public void attachObservers(Observer observer) {
    observers.add(observer);
  }

  public void detachObservers(Observer observer) {
    observers.remove(observer);
  }

  public void informObservers() {
    for (Iterator it=observers.iterator();
        it.hasNext();)
      ((Observer) it.next()).update();
  }
}
```

Figure 1-1. Observer implementation from the crosscutting's point of view.

Figure 1-1 illustrates the corresponding implementations of subjects. Both classes `Point` as well as `GuiElement` have the corresponding fields and methods, which are very close to the implementation proposed in [GHJV95]. Obviously, the observer implementation contains some crosscutting characteristics that are due to different circumstances.

First, the field for managing observers and the methods for attaching, detaching, and notifying observers occur in different classes in an identical way. The reason for this comes mainly from the fact that Java provides only single (implementation) inheritance. In case the inheritance relationship is needed for other purposes (like a superclass `AbstractGuiElement` for `GuiElement` that encapsulates domain-specific knowledge for graphical user elements), it is necessary to specify the field and the corresponding methods in a redundant way. This means that the observer-implementation crosscuts the business logic of the classes `Point` and `GuiElement`. Both classes are created due to the underlying functional decomposition criterion and the observer implementation needs to be added to these classes.

Second, the informObservers messages that are sent after each field assignment also represent crosscutting code. Each method call obviously occurs in the same way more than once within both classes. In comparison to the previously discussed fields and methods, such crosscutting calls have a finer level of granularity. They do not depend only on the decomposition criterion that modularizes a certain element from the problem space to a class. They also depend on the behavior being performed by state changing methods, i.e. on the decomposition criterion that is responsible for separating the class's behavior into methods.

The problem with this code is that in case it is necessary to change the observer implementation (for example, for storing observers in a different data structure), changes need to be performed in all classes implementing the subjects. For example, if developers decide to use a WeakHashMap[3] for storing observers, a number of class definitions need to be modified in an **invasive** way because there is no single module encapsulating the corresponding field declaration.

In case a subject is extended via inheritance and new fields are being introduced, it is necessary to implement corresponding method calls whenever the new state changes. Otherwise, the underlying subject-observer protocol would be inconsistent. In case methods that change the subject's state are overridden, it is also necessary to pay attention to the corresponding observer notifications.

The same problem occurs if it seems desirable to change the notification in a way that observers should receive a reference to the subject that has been changed and also a string describing the name of the field to that a new stated has been assigned. In such a situation, all state-changing methods as well as all informObservers methods need to be changed.

1.2.2.2 Example 2: Visitor Implementations

Frequently recurring implementations of the **visitor pattern** [GHJV95] are also known examples of crosscutting concerns (cf. e.g. [HaKi02, HaUn03a]). The intention of the visitor pattern is to encapsulate polymorphic behavior outside the class hierarchy and is used to implement operations on complex structures. This encapsulation implies two different elements: First, the functionality that needs to be provided for each element of the structure, and second, a traversal strategy that determines how the elements are to be visited.

Figure 1-2 illustrates an extraction of a typical implementation according to the exemplary one proposed in [GHJV95] for a given class hierarchy consisting of the classes A, B, C, D, and E[4]. Each class in the hierarchy implements the interface VisitedElement that declares the methods accept and getChildren. All classes in the hierarchy need to implement the method accept, which represents a **double dispatch method**: It invokes the method visit on the visitor and passes this in order to invoke the *right* method based on the static type of this. Consequently, this method cannot be inherited and a redundant definition is required.

[3] The use of the class WeakHashMap is motivated by the underlying garbage collection in Java. Objects stored in a WeakHashMap are not prevented from being garbage collected (see [Sun04c]). Observer implementations like proposed in [HaKi02] make use of WeakHashMap.

[4] An alternative Java implementation of the visitor pattern can be found for example in [PaJa98].

The method `getChildren` returns a collection of child objects, i.e. the method's intention is to provide means to navigate through the hierarchy. Consequently, all classes, which have additional children, need to provide an appropriate implementation.

The interface `Visitor` defines a method `visit` for each class to be visited in the hierarchy with a corresponding parameter. This method is being invoked by the double dispatch method. Each concrete visitor implements its version of `visit`. Typically, a concrete visitor performs some operations on the object being delivered, requests the object's children, and invokes the `accept` method on each child. Figure 1-2 illustrates an exemplary piece of code for the method `visit(A a)` in `ConcreteVisitor`.

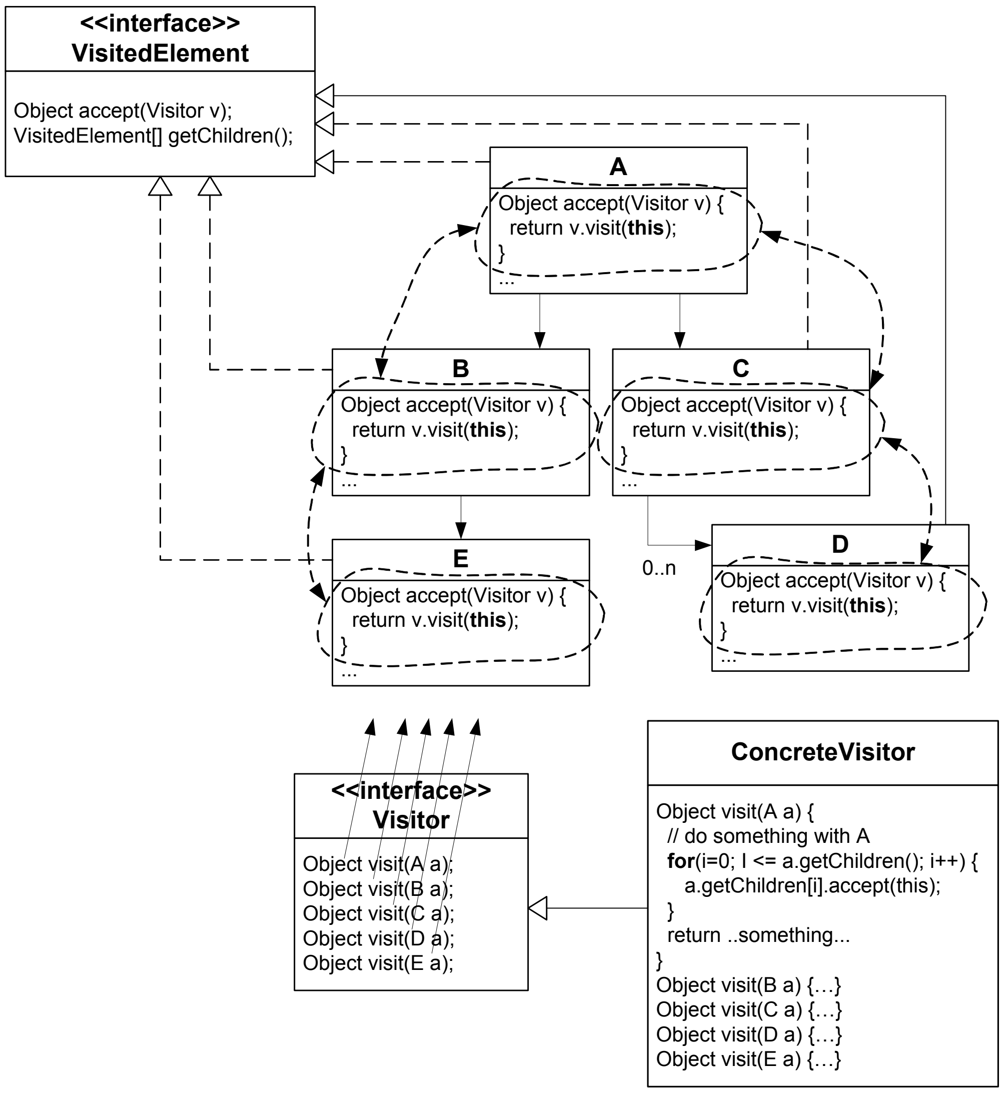


Figure 1-2. Crosscutting visitor implementation.

The visitor implementation is also a good example for crosscutting code although the underlying rules that describe the crosscutting are slightly more complex than in the observer example. The class hierarchy represents a given decomposition the visitor refers to. Consequently, the class hierarchy is the result of the applications of a dominating decomposition criterion.

First, in case the class hierarchy is a pure representation of the domain model all relationships between such classes describe classes that should be traversable via the visitor. Consequently, such relationships implicitly define what classes should implement the interface `VisitedElement`: Just by analyzing the type relationships in the class hierarchy starting from type `A` it can be determined that classes `B`, `C`, `D`, and `E` need to implement `VisitedElement`.

Second, all concrete classes implementing directly or indirectly `VisitedElement` need to implement the method `accept` in order fulfill the needed double dispatching characteristic. Syntactically, the method appears in all classes in the same way. However, the static type of `this` differs in all classes (which is the reason why the method cannot be inherited).

Third, the signatures of the interface `Visitor` are implicitly determined by the subtypes of `VisitedElement` (which are in turn implicitly defined via the relationships within the class hierarchy). In the example, it can be determined from the inheritance hierarchy that the interface `Visitor` requires five `visit` methods with the parameter types `A` to `E` (represented by the arrows from the interface `Visitor` in Figure 1-2).

Forth, in case the ordering of children is not important for the visitor[5] the different `getChildren` methods can be directly computed from the domain model. In such a case the method `getChildren` simply returns all available children without guaranteeing any ordering. For example, in Figure 1-2 (assuming that ordering is not important) the method `getChildren` in class `A` returns an instance of `B` and an instance of `C` (in arbitrary order) because those are the only available children.

The crosscutting also results in a number of problems when the visitor implementations need to be changed. In case a new return type is required for the visit method, all `accept` and `visit` methods need to be changed. This is caused by the fact that the return type is defined in the double dispatch methods as well as in the `visit` methods. While the `visit` methods are modularized in the visitor interface (and the corresponding visitor implementations), the double dispatch methods are distributed over the whole class hierarchy.

In case new classes are added to the class hierarchy, it is necessary to add corresponding double dispatch methods to them and to add corresponding `visit` methods to the visitor.

1.2.2.3 Summary

In general, the terms crosscutting concerns and the resulting crosscutting code refer to the fact that a given concern depends on a number of given modules that result from a given decomposition. In the observer example as well as in the visitor example the dominating decomposition criteria lead to a number of classes the observer or the visitor pattern is applied to (in fact, this variety of classes is a prerequisite for the

[5] In general, this is not the case. For example, in parse-trees which are typical applications of the visitor pattern the ordering is obviously important. However, [Lieb96] illustrates a number of examples where the ordering of children is not important.

application of both patterns). However, there are 1) different kinds of crosscutting and 2) the term crosscutting code closely depends on the underlying techniques being used.

First, crosscutting code like the redundant methods for attaching and detaching observers as well as the redundant double-dispatch methods in different classes seem to be quite simple kinds of crosscutting because it simply requires to copy existing code to some classes. On the other hand, as discussed in the visitor example, the underlying dependencies of the crosscutting might be quite complex.

Second, both examples are based on the programming language Java, which has a number of specific characteristics like **single implementation inheritance** and **single dispatching** (see for example [DLS+01] for a detailed discussion on method dispatching in Java and corresponding alternatives). In case Java would provide multiple inheritance and multi dispatching, the crosscutting nature would be different: Neither in the observer example nor in the visitor examples there would be the need for redundant (and syntactically equal) method definitions.

Consequently, the terms crosscutting concerns and crosscutting code cannot be considered objective terms in the sense that a certain concern or its code are *inherent crosscutting*, i.e. independent of the underlying techniques being used. In fact crosscutting concerns exist only for a given dominant decomposition and crosscutting code depends on the underlying language features. Nevertheless, both terms turn out to be useful in describing the previously illustrates code anomalies. Hence, this thesis still uses the phrases *crosscutting* and *inherent crosscutting* and implicitly means crosscutting that results from a given decomposition.

1.2.3 Aspect-Oriented Programming

The term aspect-oriented programming[6] addresses the problem of crosscutting code. Originally, the term was introduced in [KLM+97] and was defined as a *property that must be implemented but which cannot be cleanly encapsulated in a generalized procedure.*

Nowadays, the term aspect-oriented programming is understood as a set of techniques and mechanisms that permit to address the problem of crosscutting concerns (see for example [FECA04]).

Similar to object-oriented programming, there are languages or systems called **aspect-oriented programming languages** or **systems**[7] that provide a number of abstractions that permit to address the problems resulting from crosscutting concerns. The intention of aspect-oriented systems is to provide language constructs that improve the modularity of the underlying software. In that way aspect-oriented systems try to decrease (or in the best case even to solve) the problem of crosscutting code.

---

[6] Nowadays the term **aspect-oriented software development** [FECA04] is more frequently used which applies the ideas underlying aspect-oriented programming to all parts of the software development process.

[7] The use of the term *system* is mainly motivated by approaches like CLOS [Stee90], which are object systems for LISP but which do not extent the underlying programming language via new syntacitical and semantical language constructs. Instead, the new constructs are written in the underlying language themselves. Hence, such new constructs do not really represent new language features.

1.3 Problem Statement

Although the term aspect-oriented software development is being frequently used and although there are already conferences (see for example [Kic02, Aks03, Lieb04]) on that topic as well as additional literature (see for example [Labb03, FECA04, ClBa05]), there is until now no satisfying description of the key characteristics of aspect-oriented systems available. For example, it turns out that often mentioned characteristics like **quantification** and **obliviousness** [FiFr00, Film01] are not appropriate to describe aspect-oriented systems (this problem will be discussed in Chapter 5 in more detail).

Although there are a number of systems that are commonly called aspect-oriented (some of these systems are introduced in the following chapters), it turns out that they differ widely. It turns out that different systems provide quite different constructs for modularizing crosscutting concerns. Consequently, some systems are more appropriate to modularize a given crosscutting concern than other systems.

From a developer's perspective such a situation is not satisfying because it is not possible to determine on an abstract level what characteristics a system needs to provide in order to solve a given problem. A similar problematic situation arises if developers want to build their own aspect-oriented system. They need to study features of existing systems that address the modularization of crosscutting concerns. Since there are no system-independent characteristics of aspect-oriented systems available, developers need to study a large number of systems in detail and analyze on their own the impact of a certain feature in order to modularize a crosscutting concern.

The consequences of this lack of characteristics are quite annoying. It is not possible to determine from a system's description whether it is valid to call it aspect-oriented. Furthermore, it is not possible to argue about the benefits or drawbacks of aspect-orientation in general. In order to argue about the benefits and drawbacks of a specific system it is necessary to find examples that illustrate that the system is or is not able to modularize a crosscutting concern. If developers build up a new system and are interested in its *aspect-orientedness* they also needs to find such examples to compare one system with existing ones.

Another related problem similar to the ones above is that the term aspect-orientation is understood in a language independent way. This corresponds to the term object-orientation (cf. e.g. [Wegn87]), which also describes a way of developing software and which is not directly related to certain semantics of programming language constructs (i.e. languages like Java, C++ or Smalltalk are called object-oriented languages although their semantics differ). Since aspect-oriented systems exist for different languages that substantially differ with respect to their semantics, it is desirable to have a set of key characteristics that do not depend on a certain language.

Consequently, when providing a set of characteristics for aspect-oriented systems it is desirable to describe them in a language-neutral way.

1.4 Goal

This thesis provides a number of **design dimensions** that describe key characteristics of aspect-oriented systems. Each dimension describes a certain characteristic and the different divisions within the dimension describe how systems may vary along such characteristic.

The design dimensions are to be introduced in a non-formal way. The underlying intention is to rely not on language specific properties like for example whether an aspect-oriented system is build upon an object-oriented or a procedural programming language. Furthermore, language-specific details like whether an underlying object-oriented language provides multi-dispatching should not be considered. Consequently, the design dimensions do not directly rely on language-specific semantics but are extracted in a language neutral way.

Another goal of the design dimensions is to describe the different (possibly orthogonal) factors that influence the design of certain constructs and mechanisms in aspect-oriented systems. Consequently, it is the intention to extract *essential* characteristics of aspect-oriented systems – characteristics that have a large impact on how the systems can be applied and what kind of crosscutting concerns can be handled. Hence, the design dimensions must permit to reflect on the different conceptual models underlying different aspect-oriented systems.

1.5 Structure

This thesis provides a set of abstractions – called **design dimensions** – underlying the design of aspect-oriented systems.

In order to argue for the need of such abstractions and to illustrate the variety of constructs in aspect-oriented systems, the first chapters (Chapter 2, 3 and 4) introduce different aspect-oriented systems – *AspectJ*, *Hyper/J*, *AspectS*, *Sally*, and *Morphing Aspects*. The introduction of these systems serves two purposes. On the one hand, it illustrates the variety of different conceptual constructs an aspect-oriented system may possibly provide – and argues in that way for the need of abstractions that represent different conceptual approaches in aspect-orientation. On the other hand, the systems are used in the main chapter (Chapter 5 - *Design Dimensions of Aspect-Oriented Systems*) to illustrate the different kinds of dimensions – to explain each design dimension as well as to show that such dimensions are not just theoretical constructs, but can be found in different systems. Each system is being introduced with its own terminology, i.e. aspect-oriented terms like for example *join point* or *weaving* are directly taken from the underlying systems and documentations without extracting the differences and commonalities between such terms. Consequently, from these sections a number of these system-specific terms are later on (in Chapter 5) discussed in more detail and are compared to each other. While the aspect-oriented systems introduced in Chapter 2 are implemented by different research teams, the systems introduced in 3 and 4 have been built as part of this thesis.

Afterwards, the design dimensions are introduced. Thereto, the problem addressed by this thesis is explained in more detail based on the system-specific terminologies as introduced in the previous chapters. Furthermore, the chapter already discusses in its beginning some related work that tries to describe the common characteristics of aspect-oriented systems. It shows that such approaches are not sufficient in order to give a conceptual understanding of aspect-oriented systems, to distinguish aspect-oriented from non-aspect-oriented approaches and to distinguish between different conceptual approaches among different systems. The design dimensions are extracted by a corresponding observation of different aspect-oriented systems in literature or they are motivated by the corresponding different kinds of aspect-oriented systems as introduced in the previous chapters.

Then, the applicability of the design dimensions is shown by mapping existing systems to them. Thereto, the previously introduced systems are described in terms of the design dimensions.

Next, the design dimensions are used in order compare systems and select systems for a given concern. Thereto, certain concerns (variations of the **observer design pattern**) from the aspect-oriented literature are used as examples and different implementation strategies are proposed. Then, such implementation strategies are described in terms of the design dimensions and the resulting descriptions are compared to the previously described systems. Based on this comparison, the applicability of a certain system in order to modularize the given crosscutting concern is estimated. Consequently, this chapter shows that the design dimensions represent qualitative criteria than represent *essential* characteristics – characteristics that permit to estimate the appropriateness of a system described in terms of the design dimensions in order to modularize a given crosscutting concern.

After discussing related work, the thesis is summarized and concluded. The following gives a more detailed description of each chapter.

Chapter 2 (*Examples of Aspect-Oriented Systems*) introduces three aspect-oriented systems – **AspectJ**, **Hyper/J** and **AspectS** – and discusses them in detail. These systems are being used later on to illustrate problems of current aspect-oriented systems in order to modularize crosscutting concerns, and also to extract commonalities and variabilities of aspect-oriented systems. Due to the design dimensions that are later on proposed, the chapter focuses on how an aspect can be added to a system, i.e. what language constructs each system provides in order to describe the places in the code where an aspect contributes to a system. Thereto, in AspectJ the *pointcut language*, *advices*, and *introductions* are introduced, in Hyper/J *concern mappings*, *hypermodules* and *composition rules* are introduced. In AspectS the framework for *join point descriptors*, *advice instantiation*, and *installation of aspects* is introduced. Furthermore, the way each system actually achieves the adding of aspects to the application (known as *weaving* in the aspect-oriented literature) is described.

Chapter 3 (*Sally – Specifying Generic Aspects*) illustrates the need for **generic aspect-oriented systems** by illustrating examples of crosscutting concerns that cannot be modularized using conventional systems. The chapter introduces a system called **Sally** which has been developed as part of this thesis. This system provides new language features that permit to modularize crosscutting concerns that previously introduced systems fail to modularize. Obviously, Sally differs noteworthy from the previously described systems. Consequently, Sally is a representative for a number of different design alternatives available for aspect-oriented systems, which is being used later on in Chapter 5 to illustrate the underlying design dimension – the **parameterization of join point adaptations** (see section 5.5.3). The chapter also discusses related work, i.e. other systems with similar facilities.

Chapter 4 (*Morphing Aspects*) introduces **continuous weaving** – an approach that addresses a performance problem of current aspect-oriented systems related to the specific way of how aspect-oriented systems achieve the integration of aspects (i.e. *weaving*) into the system. The approach described in this chapter can be considered as a new kind of aspect-oriented system with a specific way of specifying aspects. Similar to the previous chapter Morphing Aspects represent a new approach of building aspect-oriented software and it utilizes a certain technique, which is later on identified as a new design dimension of aspect-oriented systems in respect to **weaving** (see section 5.6.3). This chapter also discusses related work, i.e. approaches that can be considered related to the way of how weaving is performed in Morphing Aspects.

Chapter 5 (*Design Dimensions of Aspect-Oriented Systems*) represents the main part of this thesis. It introduces the design dimensions of aspect-oriented systems. Thereto, the chapter discusses in detail why current language-independent approaches are not sufficient to describe the characteristics of aspect-oriented systems. Then the chapter introduces a general model for aspect-oriented systems and describes for each element of the model corresponding design dimensions. In more detail, the chapter provides design dimensions for **join point models**, **join point selection**, **join point adaptation** and **weaving**.

In Chapter 6 (*Implementations of Design Dimensions*) the design dimensions are applied to describe the systems as introduced in Chapters 2, 3 and 4. Furthermore, the design dimensions are applied to related approaches in order to study their **aspect-orientedness**. Consequently, this chapter checks the applicability of the design dimensions in order to *describe* aspect-oriented systems.

Chapter 7 (*Design Dimensions-Based Comparison and Selection*) studies the applicability of the design dimensions in order to *compare* different systems as well as to *select* a system based on the descriptions of a crosscutting concern in terms of the design dimensions. Thereto, a number of different observer-implementations are used as crosscutting concerns. Each implementation is described in terms of the design dimension and then compared with the descriptions in Chapter 6. According to the comparison, a system's capability to modularize the crosscutting concern is estimated. Consequently, for each concern a number of systems are estimated that permit to handle the concern.

Chapter 8 (*Related Work*) discusses related work. Thereto, only such works that classify aspect-oriented systems or that provide conceptual models of (aspect-oriented) systems are discussed as related work. Although the systems in Chapter 3 and 4 provide new aspect-oriented systems, the approaches related to such systems are discussed in the corresponding chapters. Consequently, the related work discussed in Chapter 8 represents the work directly related to the design dimensions of aspect-oriented systems.

Chapter 9 (*Conclusion*) summarizes, discusses, and concludes this thesis and argues for possible extensions and further work.

2

EXAMPLES OF ASPECT-ORIENTED SYSTEMS

2.1 Introduction

This section shortly introduces known systems which are commonly accepted as being aspect-oriented. More precisely, this section introduces the systems **AspectJ** [KHH+01], **Hyper/J** [OsTa01], and **AspectS** [Hirs02] as exemplary aspect-oriented systems. Each system has either been introduced at international conferences that consider the topic of aspect-oriented software development, or has been cited as a typical example of an aspect-oriented system in the literature published at such conferences.

The intention of this chapter is to give an overview of the variety of constructs that are to be considered aspect-oriented features. Such features are being used in subsequent chapters for extracting commonalities and differences among aspect-oriented system. This chapter does not intent to be a complete description of all features provided by the introduced techniques (for further details this chapter refers to the corresponding technical references available for each system).

While this thesis has been written, a number of aspect-oriented systems appeared or evolved. Examples for such systems are **PROSE** [PGA02, PGA03] and **JAC** [PSDF01], **ObjectTeams** [Herr02], or **Ceasar** [MeOs03]. However, since the intention of this chapter is to motivate and explain the design dimensions, only systems that are necessary for this task are introduced.

2.2 AspectJ

AspectJ [AspJ03, KHH+01, Labb03] is a language developed at the Xerox Palo Alto Research Center. The implementation of AspectJ directly refers to the original work on aspect-oriented programming [KLM+97] and describes itself as a general-purpose aspect language, which is commonly accepted in the literature. In contrast to most other aspect-oriented systems that are mainly research prototypes, AspectJ is already being used in industry. Furthermore, AspectJ has been studied for modularizing a number of different concerns like *logging* [HiHu04], specifications of *assertions* [KPRS00], *persistency* [RaCh03] or typical *design pattern implementations* [HaKi02]. Furthermore, the impact of using AspectJ on the software design has been studied and a number of *AspectJ idioms* or *design patterns* are already known (cf. e.g. [HaCo03, HSU03]).

AspectJ is an extension of the programming language Java and provides the following new constructs in addition to the known ones provided by Java:

- **Introduction**: An introduction[8] permits to add new supertypes to existing types or to add new methods or fields to classes or interfaces.
- **Pointcut**: A pointcut refers to elements in the execution of a program (called join points) where an aspect contributes.
- **Advice**: An advice (referring to a certain pointcut) specifies how the aspect contributes to the corresponding join points.
- **Aspect**: Aspects are class-like constructs that are used as a container for the previous mentioned language features.

2.2.1 Introduction

Introductions are designed to be pure *type increasing operations* [HaUn03a] that do not change the behavior of an application[9]. They permit to define fields and methods within a single aspect which are to be added to a number of target types (this thesis refers to these kinds of introductions as **member introductions** in the following). The visibility of introduced members corresponds to the visibility rules of Java. Furthermore, AspectJ permits to specify private introductions which are visible only by the introducing aspect (cf. [AspJ03]). Introductions permit to define new `extends` or `implements` relationships for several target types (this thesis refers to this kind of introduction as **parent introduction** in the following).

Member introductions are either achieved by introducing a member to a class or an interface and then introducing this interface to a number of target types by means of a parent introduction, or by directly adding the members to a target type. Target types are specified by so-called **type patterns**[10].

In order to specify type patterns, AspectJ permits to specify a type via its name, or via its inheritance relationship to other types.

Specifying a type via its name is achieved either by specifying the whole type name as a sequence of characters or by specifying a type name partially and using the so-called wildcard `*`. A wildcard is a placeholder for an arbitrary number of characters (including zero characters). For example, in order to specify the target type `MyType` it is possible

[8] Newer versions of AspectJ use the term **intertype declaration** instead of introduction (see [AspJ03]). However, because of its frequently use this thesis still remains using the term **introduction**.

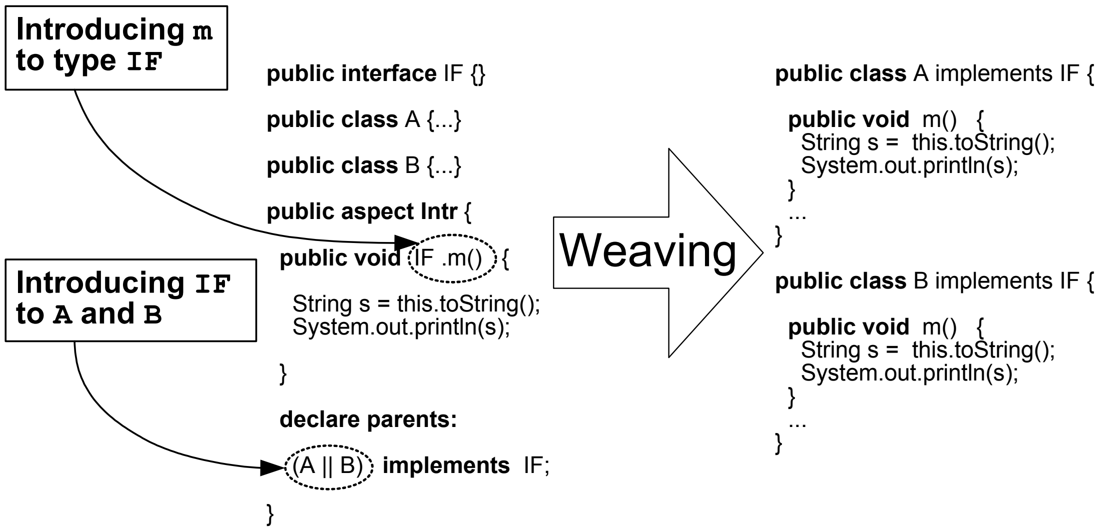
[9] However, in the current implementation there are some cases where introductions by accident *do* change the behavior of the application (cf. [Stör03] for a detailed discussion). However, the intended design principle is to provide a pure type increasing operation.

[10] In previous versions of AspectJ members could directly introduced to a number of target types which lead to some confusion concerning the type of `this` inside the method to be introduced (cf. e.g. [HaUn03a]). However, even in older versions of AspectJ introducing members to interfaces which are then introduced to a number of target types turned out to be a useful idiom (called the **Container Introduction**, c.f. [HSU03, HaCo03]).

to specify the whole sequence of characters the name consists of (i.e. the type pattern corresponds to the string MyType). Also, it is possible to specify only the starting characters and using the wildcard like MyT* (a wildcard without any additional characters simply refers to all existing types in the application).

Type relationships are expressed via the operators + and &&. The unary operator + applied after a type name refers to all types which are subtypes of the specified one. For example, the type pattern MyType+ refers to all subtypes of MyType. The binary operator && describes all types that correspond to both specified types. For example, the type pattern MyType+ && MarkerInterface+ describes all types which are subtypes of MyType as well as subtypes of MarkerInterface.

Furthermore, it is possible to enumerate type patterns using the binary operator ||. For example, the type pattern MyType || AnotherType refers to the type MyType as well as to AnotherType.



```
public interface IF {}
public class A {...}
public class B {...}
public aspect Intr {
  public void IF .m() {
    String s = this.toString();
    System.out.println(s);
  }
  declare parents:
  (A || B) implements IF;
}
```

```
public class A implements IF {
  public void m() {
    String s = this.toString();
    System.out.println(s);
  }
  ...
}
public class B implements IF {
  public void m() {
    String s = this.toString();
    System.out.println(s);
  }
  ...
}
```

Figure 2-3. Introducing method m to classes A and B in AspectJ.

Since newer versions of AspectJ only one single target type, specified via its name, can be applied to member introductions while parent introductions permit to specify an arbitrary type pattern.

An example definition of an introduction is illustrated in Figure 2-3. A method m is being introduced to two target classes A and B by performing a member introduction to the interface IF and then introducing IF to A and B within the same aspect. After compiling the application the method m becomes part of the class definitions of classes A and B: The aspect Intr becomes **woven** to the application.

2.2.2 Pointcut

Pointcuts specify those elements of an application's behavior which are adaptable by an aspect. Thereto, [KHH+01] introduces the **join point concept**. A join point is a *principled point in the execution of a program*. According to this a pointcut specifies a set of join points. Join points in AspectJ are for example the execution of a method (or constructor), the invocation of a method (or constructor), the initialization of objects, or field accesses and assignments.

Pointcuts are recursively defined: A pointcut is a **named pointcut**, a combination of pointcuts or a (parameterized) **primitive pointcut**. Named pointcuts are declared by using the special keyword `pointcut` followed by an identifier, a (possibly empty) list of parameters, and a pointcut definition. A pointcut can be combined using the binary operators `&&`, `||`, or the unary operator `!` (negation). Primitive pointcuts are predefined pointcuts in AspectJ which describe the kind of join point being addressed or additional characteristics of the corresponding join points. Primitive pointcuts like `call`, `execution`, `set` and `get` describe the kind of join point being addressed (and refer to method call, method execution, field assignment and field access join points). Primitive pointcuts `this`, `target` or `args` describe the object where a certain join point occurs in, the target object of a method call, the arguments of a call or an execution join point. Furthermore, primitive pointcuts like `within` describe the class containing the corresponding join point, `withincode` describes the method containing the corresponding join point, and `cflow` describes the control flow a join point occurs in. A complete list of all primitive pointcuts is available in [AspJ03].

The developer needs to specify a number of parameters for each primitive pointcut. In general, such parameters are signatures, type patterns, or bounded objects.

Pointcuts like `call` and `execution` (as well as `withincode`) require a parameter, which specifies the signature of the method being invoked or executed. Such signature consists of the return type, the declaring type of the corresponding method, the method's name, and a list of the method's parameter types. The types are specified by type patterns as explained in the previous section. Method names are specified similar to how type names are specified. They can either be specified by declaring the whole method name, or by declaring the method name partially in combination with the unary operator `*`, or by enumerating method name patterns using the binary operator `||`. Parameter type lists are specified either by enumerating type patterns (each type pattern separated by the operator `,`) where each type pattern represents the type in the parameter list at the specified position. Furthermore, AspectJ permits to specify the type list partially in order to abstract over the rest of the list using the operator `..` . For example, the signature (`void MyType+.set*(..)`) specifies all signatures where the corresponding method is defined in a subtype of `MyType`, where the return type is `void`, the method name starts with the characters `set`, and the method has an arbitrary number of parameters[11].

Pointcuts describing objects require a parameter that describes either the corresponding runtime type or a **bounded** identifier. An identifier is bounded, if it is declared (with a corresponding type) in the pointcut's header. The primitive pointcut `args` requires a list consisting of type patterns or bounded identifiers. It also permits the usage of the operator `..` for abstracting over parts of the type list.

[11] This kind of signature is based on a common naming convention like setter-methods which are defined by a number of frameworks like for example JavaBeans [Sun04a]. Such naming conventions are also often being used in persistency frameworks. Hence, pointcut definitions for modularizing persistency concerns often make use of such conventions (see for example [RaCh03]).

The primitive pointcuts `cflow` and `if` differ slightly from the previous ones. The `cflow` pointcut requires a pointcut as its parameter. The `if` pointcut requires a boolean expression as a parameter where each object of the expression is bounded within the pointcut.

```
aspect MyAspect {
  pointcut pc(MyType t, AType arg):
          cflow(execution(* MyObject.*(..))) && call(void MyType+.set*(..))
          && this(t) && args(arg, ..);
  ...
}
```

Figure 2-4. Example pointcut `pc` with bounded parameters `t` and `arg`.

Figure 2-4 illustrates a pointcut definition in AspectJ. The pointcut refers to all method call join points where:

- The called method is defined in a subtype of `MyType`, and
- The called method's name begins with the characters `set`, and
- The called method's return type is `void`, and
- The called method has at least one parameter, and
- The first actual parameter of the method call has the type `AType`, and
- The calling object has the type `MyType`, and
- An arbitrary method defined in class `MyObject` is on the call stack.

The join point's calling method is bound to the identifier `t` and the first actual parameter is bound to the identifier `arg`. The type constraints of `t` and `arg` are defined in the pointcut's header.

2.2.3 Advice

An advice is a module similar to a method which is implicitly invoked whenever a join point described by a corresponding pointcut is reached. It is not possible to explicitly invoke an advice via method calls or similar constructs.

An advice definition includes the keywords `before`, `after`, or `around` that describes when the advice is to be executed relatively to the corresponding join point: A **before advice** is executed before the join point is executed, an **after advice** is executed afterwards, and an **around advice** is executed instead of the original join point. Furthermore, an advice has a list of typed parameters and a pointcut the advice refers to. An advice (in correspondence to method definitions) has a body which shares the same namespace as the surrounding aspect and that permits to refer to the parameters declared in the advice's header.

AspectJ permits to refer from an advice's execution to the corresponding join point from within its body. The keyword `thisJoinPoint` delivers a reference to an object representing the join point being reached at the advice. Furthermore, an around advice permits to execute the join point being adapted via the keyword `proceed` followed by a number of parameters.

Figure 2-5 illustrates an around advice referring to the previously defined pointcut `pc`. The advice increases the variable `counter` and proceeds with the join point by passing the advice's parameters.

```
aspect MyAspect {
  int counter = 0;
  pointcut pc(MyType t, AType arg): ...

  void around(MyType t, AType arg): pc(t, a) {
     counter++;
     proceed(t, arg);
  }
}
```

Figure 2-5. Example advice increasing the variable `counter` and proceeding at the corresponding join point.

2.2.4 Aspect

An aspect is a class-like construct that represents a container for introductions, advices and pointcuts. Unlike classes, aspects are not explicitly instantiated by the developer. In correspondence to classes, aspects can define class and object variables as well as class and object methods. Aspects can be declared abstract whereby abstract aspects can include abstract methods as well as abstract pointcuts. Abstract aspects do not influence any join point. Like classes, aspects can extend other aspects or implement interfaces and override or overload methods.

```
public aspect AbstractAspect {

  abstract pointcut fooCall();

  before(): fooCall() {
    System.out.println("Before foo was called");
  }
}

public aspect ConcreteAspect extends AbstractAspect {
  // define pointcut fooCall
  pointcut fooCall(): call(void TargetClass.foo());
}
```

Figure 2-6. Concrete aspect `ConcreteAspect` defining abstract pointcut `fooCall`.

Aspects permit to override pointcuts by defining a pointcut with the same signature (pointcut name and parameters). The semantics of pointcut overriding is similar to the semantics of method overriding in Java: A concrete aspect containing a pointcut definition refers only to the overriding pointcut. Consequently, all advices within the aspect's hierarchy (referring to the overridden pointcut) refer to all join points as being specified in the overriding pointcut.

Figure 2-6 illustrates the definition of an abstract and a concrete aspect according to the **Abstract Aspect idiom** (cf. [HSU03]). The aspect `AbstractAspect` declares a pointcut `fooCall()` and also a before advice referring to this pointcut. The concrete aspect `ConcreteAspect` defines the pointcut. Hence, for the concrete aspect the advice defined in the superaspect refers to the pointcut defined in `ConcreteAspect`, and the advice adapts all method call join points referring to method calls to a method named `foo()` in an expression having the static type `TargetClass`.

2.2.5 Weaving Aspects

According to the original paper on aspect-oriented programming [KLM+97] AspectJ provides the **weaving** of aspects. Weaving describes the process that integrates aspects into an application. The weaving process is executed at compilation. When an aspect is compiled with a number of Java types the compiler computes those join points where potentially aspect-specific code needs to be invoked. At such points, additional statements are generated that check (if necessary) if actually aspect-specific code needs to be invoked and that perform the corresponding invocations (cf. [HiHu04]). Aspects are translated into ordinary Java classes in order to be executable in all Java compatible virtual machines.

While in previous versions of AspectJ weaving was performed on the source code level, newer versions of AspectJ operate on the bytecode level. Consequently, aspects can be added to ordinary Java classes and it is not necessary to have any sources available for such classes. The resulting woven code requires some additional libraries that contain class definitions like for example the join point representation provided within every advice.

2.3 Hyper/J

Hyper/J comes from the background of **subject-oriented programming** [HaOs93]. The foundation of this approach is the observation that some properties associated to an object are *subjective*. I.e. different individuals have different mental models of real-world objects. In that way subject-oriented programming is related to the aristotelian distinction between **essential** and **accidental properties** of subjects like discussed in [RaCa03] in relation to object-oriented programming: Properties associated to things are either essential, which are properties telling about what the thing is, or accidental, which are properties coming from a subjective view on the subject.

The composition techniques provided by object-oriented programming on the other hand do not provide any possibility to specify such different perspectives. Instead, there is one dominating perspective to which all other perspectives are directly related. Hence, the modularization is not based upon multiple decomposition criteria, but mainly on one dominant one. Tarr et al. [TOHS99] call this the **tyranny of the dominant decomposition** and argue that such an approach stands in contradiction to the **principle of separation of concerns** (see also Chapter 1.1).

Originally, Hyper/J was considered to be a tool that permits to specify a number of different perspectives (or dimensions and concerns according to the Hyper/J terminology) on a piece of software in correspondence to the underlying ideas of subject-oriented programming. Although the foundations of Hyper/J do not directly refer to aspect-oriented programming, more recent works of the same authors consider

Hyper/J (and subject-oriented programming in general) to be also an aspect-oriented programming approach (cf. for examples [OsTa01]). Since furthermore Hyper/J is also being considered to be an aspect-oriented programming system by a large number of different authors (see for example [FECA04] among many others), it is valid to consider it in this context and to call it an aspect-oriented system.

Hyper/J is a tool that permits to assign classes and methods specified in the programming language Java to dimensions and concerns depicted by corresponding identifiers, and to compose such dimensions and concerns using some predefined **composition rules** [OKK+96]. Thereto, Hyper/J does not extend the programming language Java itself by providing new language constructs integrated into Java (which differs from the AspectJ approach) but permits to specify the composition outside the usual type definitions in separate files. Such files are used to compose the resulting application. There are three different kinds of specification files in Hyper/J:

- **Hyperspace**: The hyperspace file contains all classes that are considered when performing the composition.
- **Concern Mapping**: The concern mapping assigns elements like classes, methods, and fields to so-called dimensions.
- **Hypermodule:** The hypermodule specifies the composition rules that are to be applied to the classes within the hyperspace.

The hyperspace just corresponds to the list of all classes that are to be composed. It can be compared to the list of all classes that are considered during a compilation process or the list of all classes within the classpath during the Java compilation process. Furthermore, since this file can be omitted (cf. [TO00]) this thesis does not consider this file to be an essential element of Hyper/J and concentrates in the following only on the concern mappings and the hypermodules.

2.3.1 Concern Mapping

A concern mapping specifies a number of identifiers that represent concerns and assigns Java entities to them. Such Java entities are packages, classes, interfaces, methods (which are called **operations** in the Hyper/J terminology) and fields.

Hyper/J defines within the concern mapping for each of those kinds of entities a corresponding keyword (`package`, `class`, `interface`, `operation` and `field`). The developer assigns an entity to a concern by declaring the corresponding keyword and afterwards specifying a list of characters that identifies the corresponding entity.

In order to identify packages, the package's name is to be declared. In order to identify classes or interfaces, they are to be specified either by declaring them in a full qualified way (i.e. including the corresponding package they are defined in) or just by specifying their name (without specifying the package). In case only the name is specified, all classes whose name corresponds to the specified one are addressed (regardless of the package they are defined in). Operations are identified by the type they are defined in (whereby the types are identified as previously described) followed by the operation's name[12]. The operation's name corresponds to a method's name. Parameter types are not considered in order to refer to methods. Due to method overloading as defined in Java, it is possible that within a class there are a number of methods with the same name that differ only with respect to the parameter types. In that case, an operation name refers to all of such methods. The identification of fields corresponds to the identification of methods – they are identified by a character sequence that describes their declaring type and an identifier that describes the field's name.

```
package myApplication : Feature.Kernel
operation check : Feature.Check
operation display : Feature.Display
operation show : Feature.Display
```

Figure 2-7. Hyper/J concern mapping specification.

Each addressed element is assigned to a concern identifier. A concern identifier is structured into two elements: One identifier describes the **dimension**, and one identifier describes the **concern** within that dimension. Both identifiers are separated by the token "`.`".

Figure 2-7 illustrates a concern mapping in Hyper/J. The first line assigns the package `myApplication` to the concern `core` in the dimension `Feature`. Consequently, all classes defined in the package become part of the corresponding concern. The other lines assign methods to concerns: All methods having the name `check` are assigned to the concern `Check` in the dimensions `Feature` and all methods having the name `display` or the name `show` are assigned to the concern `Display` in the dimension `Feature`.

2.3.2 Hypermodules

Hypermodules specify how sets of concerns are to be composed in order to generate a new application. A hypermodule specification consists of a name that identifies the hypermodule, a number of concerns that are to be considered for the composition, and a number of relationships (which correspond to the subject-oriented composition rules, cf. [TO00, OsTa01]) that are to be established between such concerns. The concerns to be considered for the composition are called **hyperslices** in the Hyper/J terminology. However, for reasons of simplicity this thesis uses the term *concern* for describing the concerns defined in a concern mapping and used within a hypermodule.

The concerns being used within a hypermodule are explicitly enumerated: All concerns identifiers (in addition to the dimension identifiers) need to be specified within a corresponding hyperslice section of the hypermodule. Figure 2-8 illustrates the hyperslice declaration of a hypermodule named `ComposedApplication` that simply relies on the concern mappings defined in the previous section.

The main part of the hypermodule consists of the specification of relationships between the complete concerns as well as the specification of relationships between

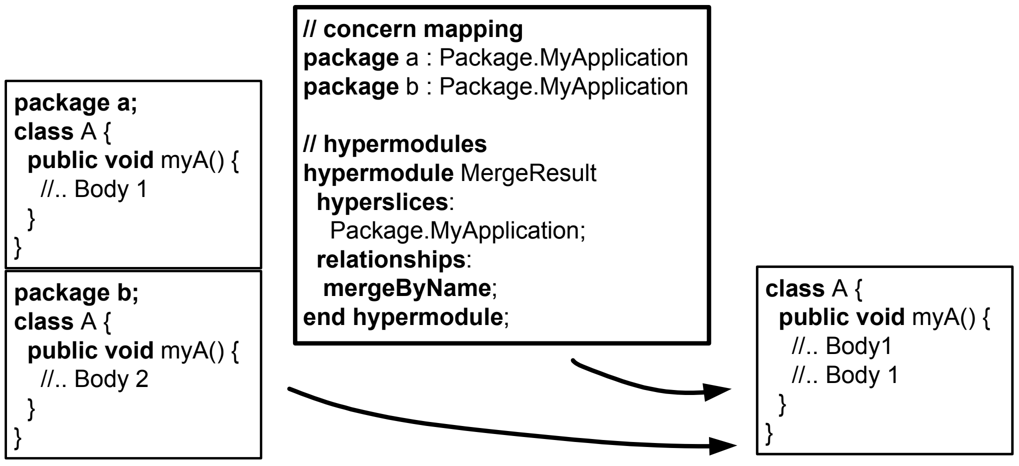
[12] The typename can be omitted so that all operations with the corresponding name are identified.

selected elements from the concerns. In general, Hyper/J provides composition rules named **general integration rules** like **merging** and **overriding** that are applied to all concerns, selected types as well as selected operations. Furthermore, each of such general integration rules can be adapted by additional relationships like **equate**, **match**, **compose** or **bracket**.

```
hypermodule ComposedApplication
    hyperslices:
      Feature.Kernel,
      Feature.Check,
      Feature.Display
    ...
```

Figure 2-8. Hyperslide specification within a hypermodule `ComposedApplication`.

The **merge** composition rule permits to compose a number of entities. For example, the application of a merge relationship to a number of classes results in the creation of a single class. That class combines members of all classes in a single one. Combining classes also means that operations are merged by adding corresponding invocations that refer to the original method definitions. Figure 2-9 illustrates the application of a merge relationship applied to two classes defined in different packages each having the same name `A` and each defining a method having the name `myA`. In such a case, Hyper/J composes a single class `A` that contains the bodies of both methods[13].



```
package a;
class A {
  public void myA() {
    //.. Body 1
  }
}
```

```
package b;
class A {
  public void myA() {
    //.. Body 2
  }
}
```

```
// concern mapping
package a : Package.MyApplication
package b : Package.MyApplication

// hypermodules
hypermodule MergeResult
  hyperslices:
    Package.MyApplication;
  relationships:
    mergeByName;
end hypermodule;
```

```
class A {
  public void myA() {
    //.. Body1
    //.. Body 1
  }
}
```

Figure 2-9. Merging two classes using the composition rule `mergeByName`.

[13] This is a slightly simplied view, because Hyper/J creates additional corresponding methods in the genenerated class containing the original method definitions (with modified signatures) and corresponding invocations to such methods. However, the illustrated code has the same semantics as the code actually generated by Hyper/J.

By applying the **override** composition rule, classes from one concern override classes from a different concern. Overriding means that all methods of classes that appear first in the import are being replaced by the content of methods whose classes appear later on in the import.

The **equate** relationship permits to consider two entities identified by different identifiers to be equal for the composition rules. Syntactically, the equate relationship consists of the keyword `equate` followed by the kind of entity (`package`, `class`, `operation`) that is to be considered equal. Afterwards, the elements to be considered equal are enumerated. Such elements are enumerated in terms of their corresponding concern mapping. For example, in the classes `A` and `B` which previously have been assigned to the concern `MyApplication` in the dimension `Package`, the identifiers `Package.MyApplication.A` and `Package.MyApplication.B` describe the corresponding classes. If the equate relationship is applied, the equal elements are composed according to the corresponding composition rule. Figure 2-10 illustrates the application of the equate relationship and its impact on the resulting application composed by using the override composition rule.

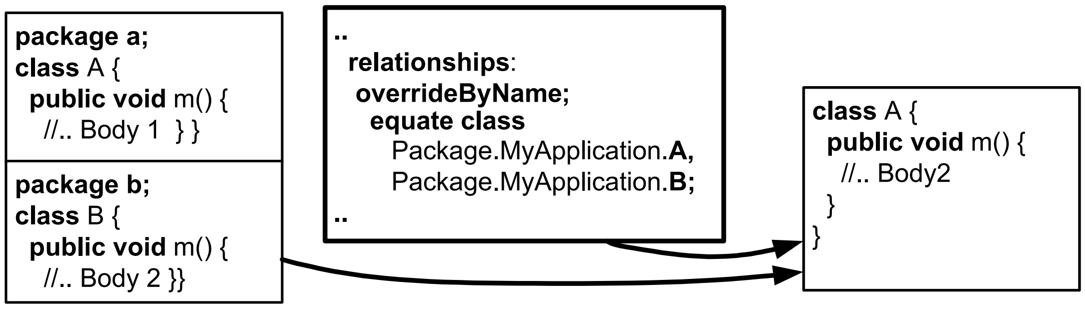

```
package a;
class A {
  public void m() {
   //.. Body 1  } }

package b;
class B {
  public void m() {
   //.. Body 2 }}
```
```
..
 relationships:
  overrideByName;
   equate class
      Package.MyApplication.A,
      Package.MyApplication.B;
..
```
```
class A {
  public void m() {
   //.. Body2
  }
}
```

Figure 2-10. Overriding class `A` with `B` by applying the `equate` relationship within an override composition rule.

The **match** relationship permits to declare in an explicit way that one entity matches other ones. Syntactically the match relationship consists of the keyword `match` followed by the kind of entity and the entity's identifier. After another keyword `with`, a sequence of characters follows that describes those entities the specified one matches with. Thereto, special tokens like a wildcard `*` and a negation `!` can be used to describe the target entities. For example, the following match specification corresponds to the above equate relationship (if only classes `A` and `B` are available):

```
match class Package.MyApplication.B with "*"
```

In case this relationships is specified instead of the equate relationship in Figure 2-10 it specifies that the class `B` that is part of the concern `Package.MyApplication` matches all other classes. Consequently, class `B` overrides all other classes and the composed application corresponds to the one in Figure 2-10.

The **compose** relationship permits to create a new unit which is a composition of a number of units, but which still keeps the original ones. In the case of type composition, the compose relationship adds a new inheritance hierarchy to those types participating in this relationship. Syntactically, the compose relationship consist of the keywords `compose` in addition to the kind of entity being composed, followed by the entity's name. Then, the keywords `with additionally` follow, and finally comes an identifier for the entity that is to be composed in addition to the first one that is prefixed by the hypermodule's name. Figure 2-11 illustrates an application of the compose relationship and the resulting output.

Bracket relationships slightly differ from the other relationships. Instead of selecting classes, operations, or fields, these relationships can be applied only to operations.

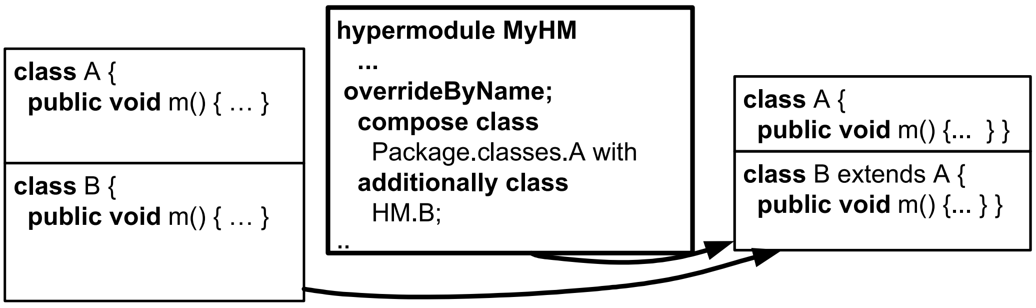


Figure 2-11. Adding a new superclass A to class B via `compose` relationships.

The **bracket** relationship permits to add additional code before or after some method calls within a certain class. Thereto, the target methods are to be specified via two character sequences that specify the target class names and the target method names. These sequences correspond to the sequences as being used within the match relationship (including e.g. wildcards). In addition to the target methods, it is possible to specify the invoking class or the invoking method by specifying their name in a corresponding concern. Then, the methods that are to be invoked before (and/or after) are specified. The before or after methods can be parameterized with some special variables: The variable `$ClassName` describes the name of the called type as a string, `$OperationName` describes the method's name being invoked as a string, `$This` described the target object being called, and `$valueArray` describes the method's actual parameters as an array. The before and after methods consequently require appropriate parameters.

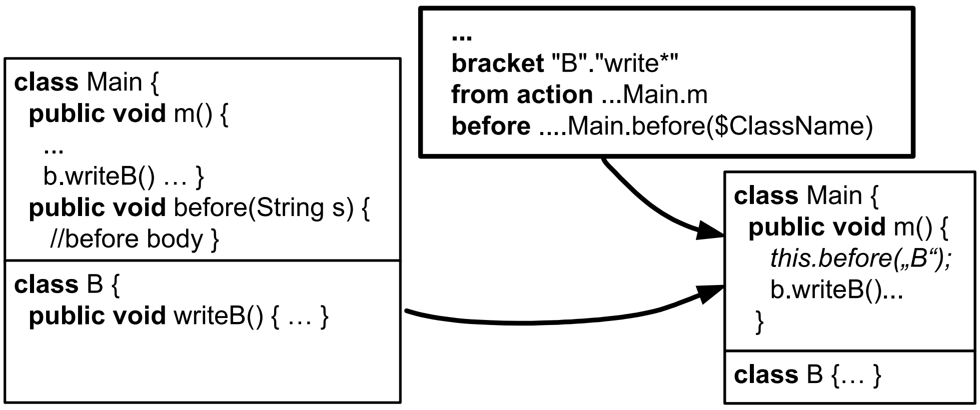


Figure 2-12. Specifying additional method calls before and after a certain call.

Figure 2-12 illustrates an application of the bracket relationship. Within the hypermodule all methods whose names start with the characters `write` are specified as target methods for the relationship. Furthermore, only those method calls that come from the method `m` in class `Main` are to be considered. Furthermore, the rule specifies that the method `before()` as specified in class `Main` is to be invoked before the original method call. The before call is parameterized with the variable `$className` (which is the string `B` in this example). Conceptually, Hyper/J generates the corresponding method call into the target method[14] as illustrated in Figure 2-12.

2.3.3 Hypermodule Composition

Although the term weaving is not explicit mentioned in the Hyper/J specification [TO00], Hyper/J also provides a composition of concerns that can be compared to weaving in AspectJ.

First, it permits to define a number of classes which are being used for the composition. Next, such classes are transformed in a way as specified within the corresponding hypermodules. Consequently, the integration of aspects as defined in AspectJ via weaving corresponds to the integration of concerns in Hyper/J.

Technically, Hyper/J reads bytecode from all classes being addressed by its concern mappings and hypermodules. Then, it composes a representation of these bytecode artifacts, which is simply a textual representation of all elements that are possibly composed. Then, Hyper/J determines from this representation the bytecode elements that are addressed by hypermodules and that need to be composed. Finally, new bytecode is generated based on the textual representation of the bytecode elements as well as the bytecode itself.

2.4 AspectS

AspectS [Hirs03, Hirs02, Hirs02b] is an implementation of a general-purpose aspect language in the Smalltalk dialect Squeak [IKM+97] based upon the Smalltalk specific possibility to provide **method wrappers** as proposed in [BFRJ98]. The features provided by AspectS are mainly motivated by AspectJ: There are different kinds of advice (before/after and around) and there is a notion of join points and pointcuts. In contrast to AspectJ, which is a language extension of Java, AspectS is an **object-oriented framework** (in the sense as introduced in [JoFo88]): All applied composition techniques are already part of the base language. Due to this, AspectS permits to weave aspects at runtime, because Smalltalk objects are used for composing aspects with the application.

In order to understand the underlying aspect composition in AspectJ it is necessary to understand the metaobject facilities of Smalltalk. First, the framework of AspectS is introduced throughout the following two sections. Afterwards, the metaobject facility of

[14] This is a slightly simplified view on the generation. In fact, Hyper/J extracts a number of different method bodies, composes separate methods and generates appropriate calls to such generated methods. However, the details of the composition are irrelevant for this thesis.

Smalltalk with respect to its method representation is shortly discussed. Finally, the relationship between the framework and such metaobject facility is explained.

2.4.1 The Framework AspectS

The language features of AspectS like advice or introduction are usual object-oriented constructs in Smalltalk. Aspects, pointcuts and advices are represented as objects. In contrast to AspectJ, AspectS requires an explicit instantiation of aspects by the developer. Furthermore, the framework makes use of some naming conventions in order to create pointcuts and advice.

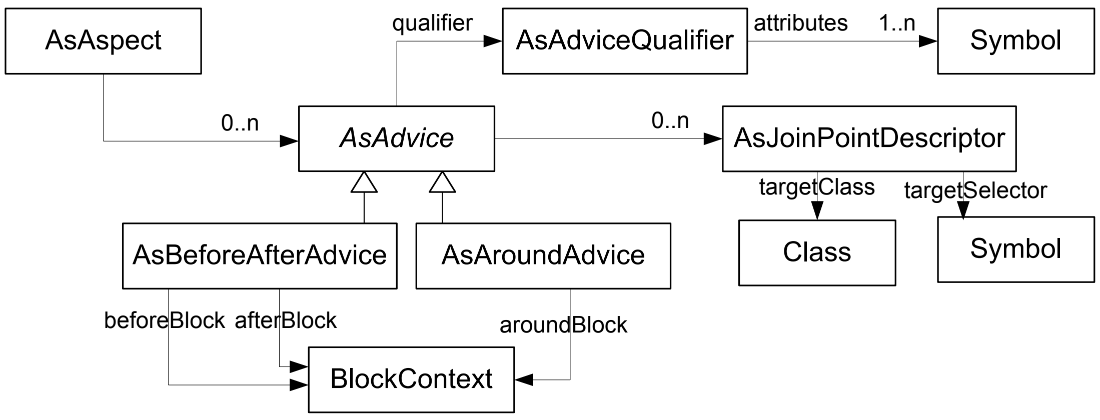


Figure 2-13. Structure of aspects (with advice) in AspectS.

Figure 2-13 illustrates a class diagram of those classes relevant for applying AspectS:

- The (abstract) class `AsAspect` is the root class for all aspects. It defines the mechanism how advices are created once an aspect is instantiated, and also defines additional fields and methods for managing aspects.
- An aspect has a number of associated advices. Each advice is either an instance of `AsBeforeAfterAdvice` or an instance of `AsAroundAdvice`. The first kind of advice permits to add additional behavior before or after a method is being executed. The latter one potentially permits to replace a target method.
- The behavior that is being executed when a certain method execution is being performed is stored in a **block**[15]. The corresponding class `BlockContext` is a system-defined class.
- The methods that are being wrapped are described by so-called join point descriptors (instances of `JoinPointDescriptor`)[16], which consist of a

15 In Smalltalk **lexical closures** are denoted under the term block. In other words, blocks are *the Smalltalk equivalent of closures* [GrJo89]. Blocks are objects which encapsulate behavior and which permit to access the local context in which they are instantiated. Furthermore, blocks permits to receive explicit parameters in order to perform the corresponding behavior (cf. [GoRo89]).

16 This is a slightly simplified view on the relationship between an advice and its join point descriptors. In fact, every advice has a reference to a block whose evaluation delivers a collection of join point descriptors.

reference to a class object and a reference to a selector (which corresponds to a method name in Smalltalk). The class object corresponds to the system defined metaobject. The selector corresponds to a `Symbol` instance defined for all methods in the system (and which represents a method's name).

- Each advice has an additional **advice qualifier** that specifies a number of additional constraints within its attributes whereby each attribute is a symbol. Exemplary attributes are `#receiverClassSpecific` or `#cfFirstClass`. The first one defines that the corresponding advice is to be invoked whenever an arbitrary instance of that class or any subclass receives a corresponding message. The latter causes the execution of the advice whenever an instance of the class occurs for the first time on the call stack. Altogether, AspectS provides ten different kinds of advice qualifiers (cf. [Hirs02]) that will be explained in more detail in the next section.

In order to create a new aspect, the developer needs to create a subclass of `AsAspect`. According to a naming convention in AspectS that relies on the reflective capabilities of Squeak each advice requires to be specified in a method whose selector starts with the character sequence "`advice`". Each of such methods needs to return an initialized instance of `AsAdvice`.

In order to weave an aspect the developer explicitly needs to instantiate an aspect and invoke its method `install` (which is defined in the abstract class `AsAspect`). Only after installing an aspect, the base application is being affected by the aspect. When an aspect receives an `install` message, it invokes all advice methods in order to retrieve a set of advice instances. From these instances, the corresponding target methods are being computed (due to their join point descriptors). Such target methods are being used for the corresponding method wrappers. In order to unweave an aspect developers need to invoke its method `uninstall` (which is also defined in `AsAspect`).

```
AsAspect subclass: #MyAspect
 adviceATest
  ^ AsBeforeAfterAdvice
     qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
     pointcut:
       [OrderedCollection
         with: (AsJoinPointDescriptor targetClass: A targetSelector: #m)]
     afterBlock:
       [:receiver :arguments :aspect :client :return |
         Transcript show: 'invoked method m in A or subclass'.
       ]
```

Figure 2-14. Simple aspect definition in AspectS.

The two different advice classes provide a number of class methods that permit a compact instantiation and initialization of corresponding advice objects.

Figure 2-14 illustrates a simple aspect definition. The aspect `MyAspect` contains a method `adviceATest`, which returns an instance of `AsBeforeAfterAdvice`. This instance is initialized with

- An advice qualifier which states that the advice is receiver class specific,

- A pointcut which defines the method named `m` in class `A` to be the target method of this aspect, and
- The advice that simply prints out the text "`invoked method m in A or subclass`".

When the aspect is instantiated and installed it prints the corresponding message every time an object of class `A` or one of its subclasses receives a message `m`.

```
AsAspect subclass: #MyAspect
 adviceATest
  ^ AsAroundAdvice
     qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
     pointcut:
      [OrderedCollection
        with: (AsJoinPointDescriptor targetClass: A targetSelector: #m)]
     aroundBlock:
      [:receiver :args :aspect :client :method |
        " something before……"
        method valueWithReceiver: receiver arguments: args
        " something after……"
      ]
```

Figure 2-15. Around advice with proceed.

Each block representing the behavior being executed at the corresponding advice receives a fixed number of parameters being defined by AspectS. An after block (as illustrated in the example above) has 5 parameters:

- The first one represents the object that is about to execute the corresponding target method,
- The second once represents the actual parameters of the method call,
- The third one represents the instance of the aspect containing the advice definition[17],
- The fourth one represents the sender of the message, and
- The last one is the object the corresponding method returns.

Before blocks have one parameter less since there is no return object. Around blocks have as their fifth parameter an object representing the method (i.e. an instance of `CompiledMethod`) that is about to be invoked. In order to invoke the original method within an around advice, the compiled method can be executed with the corresponding parameters (actual parameters as well as the owning object). Consequently, the delivered compiled method of around blocks plays a similar role as the keyword `proceed` in AspectJ within around advices.

Figure 2-15 illustrates the application of an around advice that also invokes the wrapped method. The advice sends the message `valueWithReceiver:`

[17] In case the programming convention of AspectS to define the corresponding block within an aspect's method is being followed, this parameter is not needed since `self` within the block refers already to the corresponding aspect instance.

arguments: to the method object and passes the original receiver as well as the original arguments to it. So whenever the original method m is being invoked, the corresponding advice is invoked and executes some code, then executes the original method, and finally executes some further code afterwards.

AspectS also defines introductions similar to AspectJ. Introductions in AspectS conceptually permit to add new methods to classes and are implemented as AsAdvice subclasses. Consequently, introductions also have join point descriptors. However, the descriptors play a different role than in before, after, and around advices: Instead of selecting existing methods in the system that are to be adapted by the corresponding aspect, they describe the name of the method that should be introduced. Consequently, once the corresponding aspect defining the introduction is being installed, a new method object is created: The method is added to the class defined in the join point descriptor with the corresponding selector that is defined there. Then, the new method is being wrapped with the introduction block. The introduction's block parameters correspond to the parameters of a before block.

```
AsAspect subclass: #MyAspect
 adviceIntroduce
  ^ AsIntroductionAdvice
   qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
   pointcut: [OrderedCollection
      with: (AsJoinPointDescriptor targetClass: A targetSelector: #newM)
   ]
   IntroBlock: [:receiver :arguments :aspect :client |
     "the behavior being executed"
   ].
```

Figure 2-16. Introducing method newM to class A.

Figure 2-16 illustrates the application of an introduction in AspectS. The aspect MyAspect defines within its method adviceIntroduce an introduction with the class A as its target. The behavior being executed at the introduced method is specified within the block passed to IntroBlock. The selector for this introduced method is newM. Consequently, whenever an instance of A (or any subclass) receives a message newM the introduction block is being executed.

Although conceptually the introduction of methods is similar to method introductions in AspectJ, there is still a difference in respect to the self-reference within the method body. Since the introduced method's body is being defined within an aspect (corresponding to the proposed design guidelines of AspectS), the self-reference within the introduction block refers to the aspect instance and not to the object owning the method. Hence, self-calls (to the owning object) need to be sent by using the first parameter of the introduction block. Another consequence of using blocks for the method body is that super calls are not that easily possible and require the use of the reflective features of Smalltalk.

Due to the lack of types in Smalltalk and the resulting lack of method overloading, introductions cannot potentially change the behavior of objects by introducing a method that (accidentally) overloads an already defined method in the target class (which is the case in AspectJ).

2.4.2 Advice Qualifiers

AspectS provides a number of advice qualifiers that are referred to by predefined symbols. There are qualifiers that permit to select join points based on

- The sending or receiving class (`#receiverClassSpecific`, `#senderClassSpecific`),
- The sending or receiving instances (`#receiverInstanceSpecific`, `#senderInstanceSpecific`),
- The first occurrence of the receiving class or instance on the call stack (`#cfFirstClass`, `#cfFirstInstance`),
- The non-first occurrence of the receiving class or instance on the call stack (`#cfAllButFirstClass`, `#cfAllButFirstInstance`),
- The first occurrence of a super message on the call stack (`#cfFirstSuper`) or the non-first occurrence of a super message on the call stack (`#cfAllButFirstSuper`).

In order to determine whether a certain advice is to be executed, the advice qualifiers `#senderClassSpecific`, `#senderInstanceSpecific` and `#receiver InstanceSpecific` require additional data about the permitted objects and classes. Thereto, aspect instances maintain additional references to sender and receiver instances as well as to sender and receiver classes (see Figure 2-17).

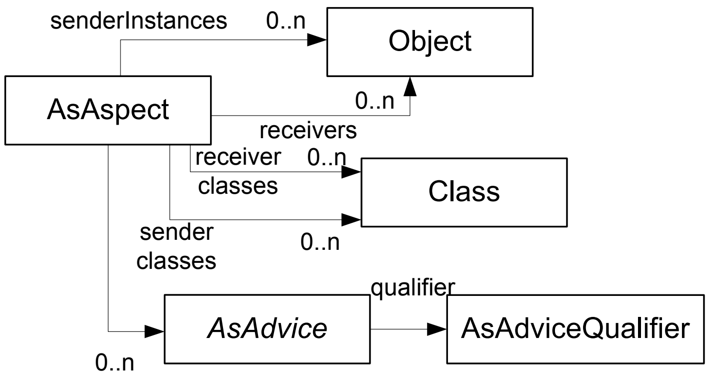


Figure 2-17. Relationships for instance and class specific advices.

Developers that specify class or instance specific qualifiers need to add the corresponding objects and classes to the aspect, which provides corresponding methods for this purpose.

2.4.3 Method Representation in Smalltalk

In contrast to for example Java the Smalltalk object system has much richer meta-object capabilities. Classes are objects themselves, (compiled) methods have an object representation, etc..

In Smalltalk every class is an instance of class `Class` that indirectly extends class `Behavior` (see Figure 2-18). This class has a reference to an instance of `MethodDictionary`, which manages all methods belonging to a certain class (see

[GoRo89, FoJo86] for a more detailed discussion). A method is an instance of class `CompiledMethod` that provides fields, which contain among others a byte code representation of the represented method. Furthermore, a compiled object has access to its source code or permits to reconstruct source code from its bytecode.

Whenever a client passes a message to a target object, Smalltalk's virtual machine performs the corresponding method lookup and computes the target method to be invoked. Then, the bytecode of this method is being evaluated with the corresponding receiver object and the arguments. Developers can also explicitly evaluate a compiled method by invoking its method `valueWithReceiver` by passing the target object and an array representing the message's parameters.

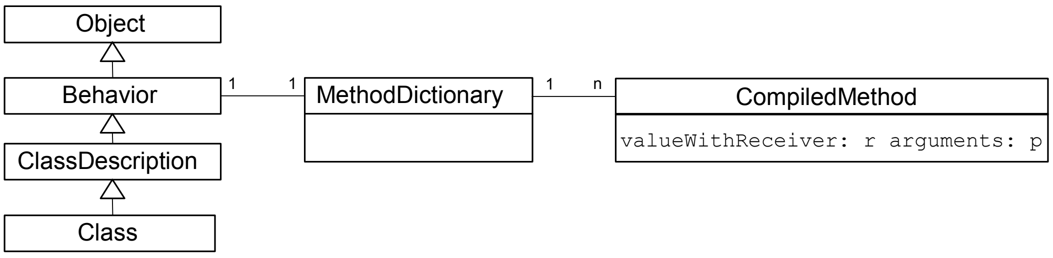


Figure 2-18. Smalltalk class hierarchy of `Class`.

The metaobject representation of classes and methods permits a flexible adaptation of class and object behavior. For example, it is possible to execute some additional code in case a message sent to an object is not understood. Such mechanism is used for example in [GSR96] for implementing **roles** (cf. e.g. [KrØs96]).

In order to adapt the behavior of methods in Smalltalk there are a number of different alternatives that are commonly known under the term **method wrapping** (cf. [BFRJ98]). Among others there are three often-applied ones:

- Changing the method's source code and recompilation of the corresponding method[18],
- Changing the byte code by changing the state of the corresponding compiled method object, or
- Removing the original method from the class's method dictionary and add a new method to it (that potentially also refers to the old one by invoking the `valueWithReceiver` method).

Hence, the ability to change the behavior of methods is directly available in the Smalltalk system and it is not necessary to provide external tools in order to change the behavior of applications.

[18] Such an approach is possible since the compiler is typically part of the runtime environment in Smalltalk systems. Consequently, an application can run a compilation process.

2.4.4 Method Wrapping in AspectS

The implementation of AspectS directly relies on method wrappers based on the replacement of objects in the corresponding method dictionary.

The framework contains an abstract class `AsMethodWrapper` subclassing `CompiledMethod` (see Figure 2-19). For each different kind of advice (including the introduction advice) there is a corresponding wrapper class (`AsBeforeAfterWrapper`, `AsAroundWrapper`, `AsIntroductionWrapper`). Instances of these wrapper classes refer to the block objects defining the advice specific behavior (as explained in section 2.4.1) as well as to the compiled method they are wrapping[19]. In same way each wrapper has a reference to its advice qualifier.

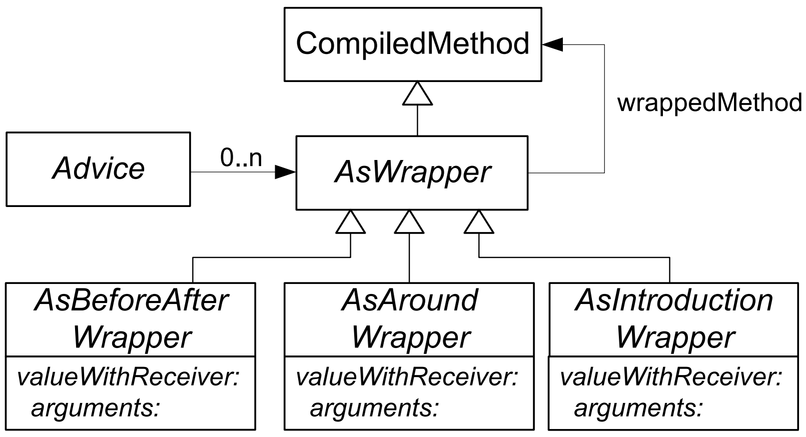


Figure 2-19. Wrapper implementation in AspectS.

The bytecode of each wrapper simply forwards to the method `valueWithReceiver:arguments:`, which is defined by each wrapper. This method includes the semantics of the different kinds of wrappers. In case of a before/after wrapper, it checks whether the advice qualifiers are fulfilled. In case they are fulfilled it executes the before block, the original method, and the after block.

If an aspect receives an `install` message, it creates corresponding wrapper instances for all method objects described by the join point descriptor. Then, all of such methods are being replaced by the corresponding wrapper instances. Consequently, whenever an object receives a message whereby the corresponding method is wrapped, the wrapper object is being evaluated. Uninstalling an aspect replaces for all existing method wrapper the corresponding entry in the method dictionary with the corresponding original method objects.

[19] This is rather a conceptual explaination. Technically, `AsMethodWrapper` defines a class field containing a dictionary that maintains wrapper instances and an additional state for them.

2.5 Chapter Summary

This chapter introduced three aspect-oriented systems. Two of them – AspectJ and Hyper/J – are based on the programming language Java. AspectS is based on the Smalltalk dialect Squeak. Each system's features have been introduced and for each system it is explained how the composition of aspects is being implemented.

Although all systems call themselves aspect-oriented (because they are commonly accepted called aspect-oriented and are cited in the aspect-oriented literature), there are still a number of differences among them. First, the terminology being used in Hyper/J differs in a significant way from AspectJ and AspectS. Furthermore, it is not obvious what the similarities between the composition relationships defined in Hyper/J and the language features of AspectJ or the framework of AspectS are. Another issues is that although AspectJ and AspectS use the term pointcut, the underlying concept is slightly confusing: In both approaches the way how pointcuts specify *the join points* to be selected (although is it not clear whether both approaches have the same conceptual understanding of join points) differs noteworthy.

In general, it is not obvious what the common characteristics of the proposed aspect-oriented systems are and it is not clear what the benefits and liabilities of the proposed systems are with respect to modularizing a given crosscutting concern.

The systems that are introduced in the following two chapters (Chapter 3 *Sally – Specifying Generic Aspects,* Chapter 4 -*Morphing Aspects*) directly refer to the here proposed systems. Chapter 3 identifies the problem that there are recurring crosscutting concerns that cannot be modularized with the here proposed constructs and introduces a new system that relies on different design decisions than the here proposed ones. Chapter 4 refers to a performance problem caused by the here proposed systems and their means to weave in aspects and proposes a new system (also based on different design decisions) that overcomes such problems.

The main chapter of this thesis (Chapter 5 - *Design Dimensions of Aspect-Oriented Systems*) relies on the here introduced systems and uses them for illustrating different design dimensions underlying aspect-oriented systems. It turns out, that there are a number of essential differences among the three proposed ones and that they differ in respect to modularizing given crosscutting concerns (which will be shown in Chapter 7 – *Design Dimensions-Based Comparison and Selection*).

3

SALLY – SPECIFYING GENERIC ASPECTS

3.1 Introduction

Aspect-oriented software development deals with the modularization of concerns that cannot be encapsulated by traditional composition techniques. Without modularization such concerns would be spread over numerous modules. Hence, such concerns are called *crosscutting concerns* and the code belonging to them *crosscutting code* (see also section 1.2.2).

Aspect-oriented programming systems like AspectJ [KHH+01] or Hyper/J [TO00] that extend object-oriented programming languages provide new composition techniques in addition to the existing ones. A core tool of aspect-oriented languages is the *weaver*: It takes *self-contained* concerns and weaves them into applications. This allows programmers to treat such (crosscutting) code in separate modules and to accomplish the crosscutting effect only afterwards.

AspectJ and Hyper/J support the concept of *introduction* to define new members for existing types outside the original class or interface definition. Since both are based on a strongly typed language (the programming language Java), the question whether a class has a certain member has to be answered at compile time. Both, AspectJ and Hyper/J claim to solve the problem of crosscutting concerns. Thus, they claim to solve the problem that crosscutting code coming from a single concern has to be implemented in different modules. However, there are often occurring examples of crosscutting code that cannot be modularized using such implementations of introductions. Hence, a more advanced implementation is needed.

This chapter addresses the problem of introductions in current aspect-oriented systems based on strongly typed languages, and introduces with Sally, a new language approach providing generic (or parametric) aspects, that addresses these problems.

The following subsections are structured as follows. Section 3.2 introduces the general concept of introductions and discusses its implementation in AspectJ and Hyper/J. Section 3.3 illustrates some typical examples of crosscutting code that cannot be modularized by AspectJ or Hyper/J. Section 3.4 introduces the language Sally that addresses the previously described problems and describes the underlying mechanism for parametric aspects. Section 3.5 applies parametric introductions in order to overcome the mentioned problems. Section 3.6 compares parametric introductions to other relevant concepts. Finally, section 3.7 concludes this chapter.

3.2 *Introductions* in Aspect-Oriented Systems

The term *introduction* was originally introduced in AspectJ. It implements a mechanism for adding fields, methods, and interfaces to classes. This is similar to *open classes* as described in [Cann82, CLCM00]. Introductions were motivated by the observation that different concerns have a direct impact on the type structure of object-oriented applications. As a consequence modularization is compromised since some elements in the type structure, like certain fields and methods, come from different concerns. Aspect-oriented techniques permit to remove these elements from the type definition and provide a mechanism to introduce them at weave time. An introduction is a strictly type increasing operation on types since it adds new features to types but does not permit to remove anything.

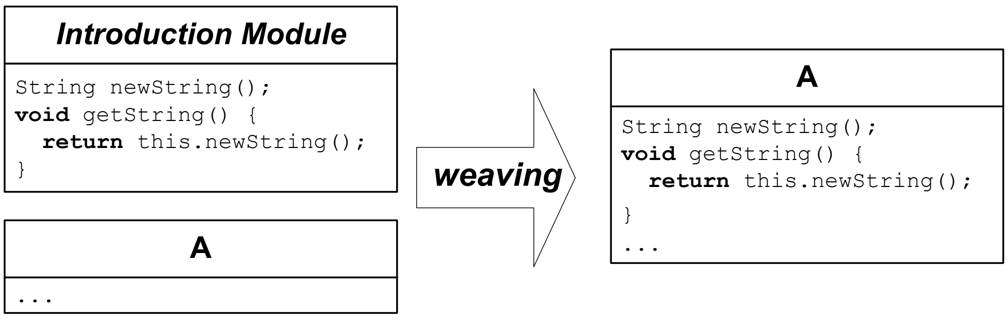


Figure 3-1. Aspect-oriented introductions.

Figure 3-1 illustrates an aspect-oriented introduction. A special introduction module defines new members that are to be introduced into a target class `A`. The weaver takes the introduction module and the target class and weaves them together. Thus, all elements of the introduction module become members of the target class `A`.

A key question of an introduction mechanism is how to handle the self-reference `this` in the introduction module. Its handling has a direct impact on the type correctness of the code that is supposed to be introduced. If `this` is bound at *weave-time*, that means when the introduced members become part of a target class, it is hardly possible to determine the correctness of an *abstract introduction*. Abstract introductions are introductions that can be reused in different contexts. That means the target classes are unknown at the time the introduction is defined[20]. If inside an abstract introduction `this` is bound at weave-time and the introduced code sends some messages to `this`, it is not possible to determine whether the (unknown) target type provides an appropriate method.

[20] In aspect-oriented programming such abstract introductions are often used to define aspect libraries that are provided by a third party and adapted by the application programmer.

In some approaches, `this` is not bound to any type at first until the weaving process adds the introduction modules to its targets. Other approaches bind `this` already before weave-time. The binding of `this` at weave-time permits a flexible combination of introductions since an introduced method may use further introduced members. In case `this` is already bound before weave time, the corresponding type can be used for type checking of the introduced code.

3.2.1 Introductions in AspectJ

In AspectJ introductions are declared in the class-like construct *aspect*. It syntactically 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() {...}
}
```

Figure 3-2. Introductions in AspectJ.

Figure 3-2 contains an aspect `MemberIntroduction` that adds a field `newString` and a method `doSomething` to class `A`. The aspect `InterfaceIntroduction` adds the interface to class `A`. The aspect `TypePatternIntroduction` specifies a new method `doSomething2` for every subclass of `A`.

An often-used idiom in AspectJ is the *container introduction* (cf. [HSU03]): The application of introductions to an interface that is later on assigned to a class. In such a case, AspectJ applies the introduction to all classes implementing the interface. Figure 3-3 illustrates such an introduction. An aspect `IntroductionLoader` introduces a new field `newString` to an interface `Container`. A different aspect introduces this interface to a target class. The result after weaving is that `TargetClass` contains the introduced field `newString` and the introduced interface `Container`.

The main purpose of container introductions is to apply a collection of introductions to several different target classes not known at introduction definition time. The container is then introduced to all target classes without the need to perform any invasive changes in the introduction definition. This reduces the effort of applying introductions since it only needs one `declare parents` statement. Furthermore, introductions can be applied without knowing in detail all elements that are part of the container.

In the ordinary application of introductions since version 1.1, AspectJ binds `this` at weave-time. Thus, at introduction definition time `this` does not refer to any type. This approach works without problems as long as all target classes of an introduction exist, i.e., all possible classes matching the type pattern are known when the introduction is defined. For example, in Figure 3-2 this is true for the aspect MemberIntroduction since the only class that matches the type pattern is a class named A. At weaving, the weaver binds this to the target class A and checks if all occurrences of this match A. If this is not true, the weaver throws an exception. However, if not every possible class exists (as possible up to version 1.1) the introduced code may contain type errors. For example, a type pattern (A*) refers to all classes whose names begin with an A. In such a case, AspectJ cannot determine if the use of this inside the introduced code is type correct.

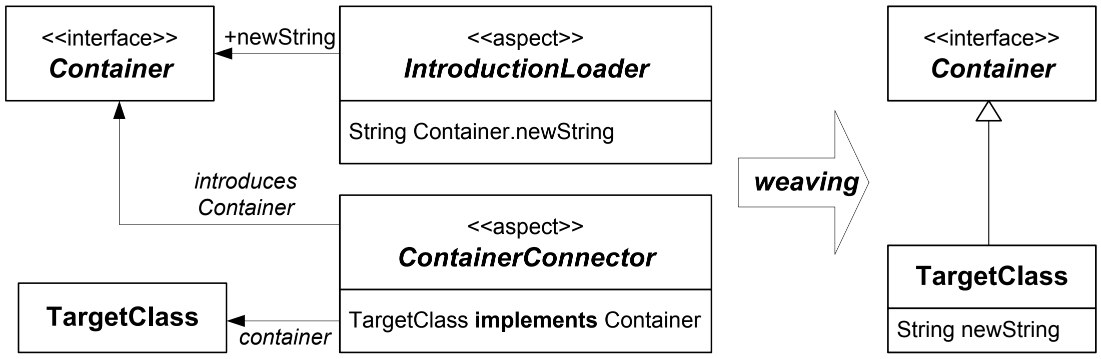


Figure 3-3. Container introduction in AspectJ.

In container introductions, this is bound to the container type and not to the target class (which is somehow misleading since the introduction is performed on the target class and not on an interface).

Hence, AspectJ realizes the binding of this in two ways: Either at weaving time (in the usual introduction application) or at introduction definition time (container connections)[21].

3.2.2 Introductions in Hyper/J

In Hyper/J introductions are realized by defining classes that contain the members to be introduced, and a corresponding *hypermodule* that defines how to weave the participating classes.

Figure 3-4 shows schematically how an introduction in Hyper/J looks like and, furthermore, lists an extract from the corresponding hypermodule specification file: The class MemberIntroduction contains the members to be introduced and is defined in an ordinary class. The corresponding hypermodule lays down that the class MemberIntroduction is to be composed with class A. In fact, the weaver introduces the class MemberIntroduction as a new super-class of A so that class A has all methods defined in the introduced super-class. The use of the composition rule compose permits the introduction of members of one class into several different target classes. However, it does not really "introduce" the members since members of

[21] Since version 1.1 only the latter way of this binding is supported.

the introduced class do not physically become members of the target class. To do so, Hyper/J provides an `equate` relationship. It permits two classes to be woven together to form one single class. However, the disadvantage of this relationship is that it cannot be used to introduce members into more than one target class (see further section 2.3).

```
class A {...}
class MemberIntroduction {
  public String newString;
  public void doSomething() {...}
}
// hypermodule specification
relationships:
  ...
  compose class MemberIntroduction with
            additionally class A;
...
```

Figure 3-4. Introductions in Hyper/J.

Since Hyper/J weaves Java classes, `this` is bound to the corresponding class in all members that are to be introduced. Introduced members are regular Java members and, therefore, can already be accessed from other classes before weaving. If `equate` is used Hyper/J *forwards* all calls to the introduced members to the woven class, that means all calls to the class to be introduced are transformed into calls to the woven class. Due to this forwarding mechanism, Hyper/J does not permit to introduce members to more than one class. If it were possible, it would be ambiguous to what woven class a call has to be forwarded.

3.3 Where Current Implementations Fail

This section presents some typical examples of static crosscutting code that cannot completely be modularized by using introductions in AspectJ and/or Hyper/J. Here, often used implementations of well-known GoF-design patterns [GHJV95] are chosen for two reasons. First, in aspect-oriented programming some of those implementations are usually regarded as aspects that need be modularized and reused in different situations [HaKi02]. Second, the used implementations of GoF design patterns are commonly known. Hence, it is not necessary to motivate the implementation or to explain in what context they usually occur.

3.3.1 Singleton implementations

A straightforward implementation of the **singleton design pattern** [GHJV95] in Java is to add some members to the class that is supposed to become a singleton: A (private) static field of the same type as its class, a private constructor and a public static method that returns the singleton instance[22]. That means, every class that is supposed to become

[22] A discussing of topics like garbage collection in the implementation of the singleton pattern is avoided here. For a more comprehensive discussion see e.g. [Gran98], pp. 127-133.

a singleton contains all these members. If the singleton pattern is applied to more than one class, these members represent static crosscutting code. The code stems from the same concern ("make a class a singleton") that changes the class structure of a system, but differs slightly from class to class.

```
class ASingleton {
  private ASingleton () {}
  private static ASingleton
    instance = new ASingleton ();
  public static ASingleton getInstance() {
    return instance;
  }
  ...
}
class BSingleton {
  private static BSingleton
    instance = new BSingleton ();
  private BSingleton() {}
  public static BSingleton getInstance() {
    return instance;
  }
  ...
}
```

Figure 3-5. Singleton implementations in Java.

For example, in Figure 3-5 two classes implement a singleton. Both have the singleton specific members whose types differ. Obviously, inheritance as defined in the programming language Java does not help in this situation to modularize the singleton implementation because static fields cannot be distributed over different classes in Java using inheritance.

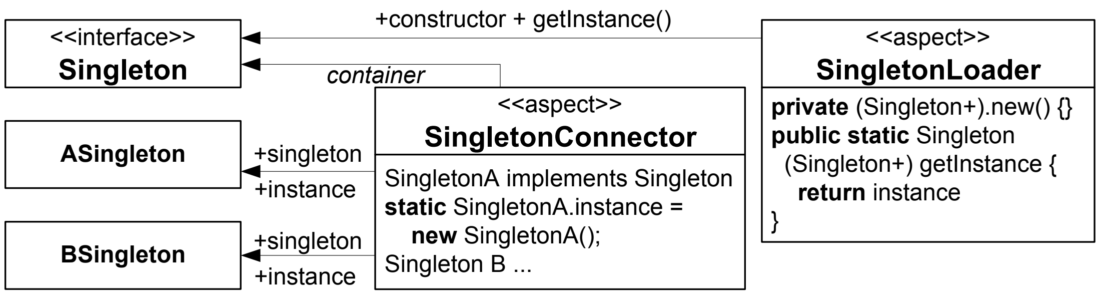


Figure 3-6. Singleton Implementation in AspectJ.

In AspectJ the singleton can be implemented by introducing all singleton specific members to an interface `Singleton`, which represents the container in a container connection (see Figure 3-6). This container is afterwards connected to a target class via an introduction (using `declare parents`). AspectJ does not permit to define the object creation within `SingletonLoader` because there is no possibility to refer to the class to which `Singleton` will be introduced[23]. Instead, this has to be defined in the connector. Hence, AspectJ does not permit to define the singleton aspect in one single module because instance creation is still scattered over different connecting aspects. Furthermore, the return type used in `getInstance` does not correspond to the type of the target classes (`ASingleton`, `BSingleton`). As a consequence, typing information gets lost during weaving and every object requesting the singleton instance has to perform a type cast.

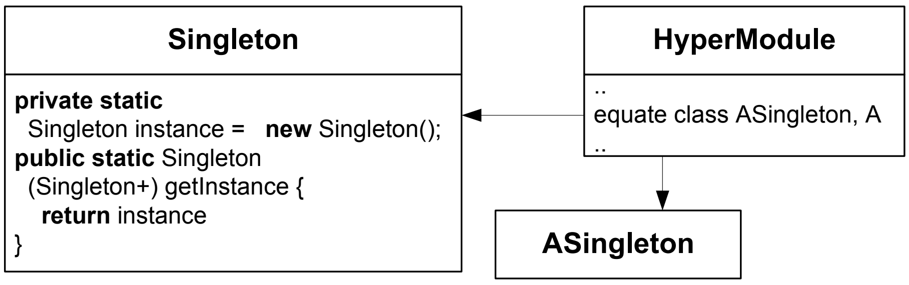

Singleton
private static
Singleton instance = new Singleton();
public static Singleton
(Singleton+) getInstance {
return instance
}
HyperModule
..
equate class ASingleton, A
..
ASingleton

Figure 3-7. Singleton implementation in Hyper/J (only one target).

Figure 3-7 illustrates an implementation of the singleton in Hyper/J by using the `equate` relationship. The result is that the type of the class variable and the return type of `getInstance` correspond to the target class since class `Singleton` and `ASingleton` and, therefore, all occurrences of `Singleton` in the woven code are replaced by `ASingleton`. Thus, the loss of type information that occurred in AspectJ does not happen in Hyper/J. However, if an `equate` relationship is chosen, it is not possible to combine the singleton class with multiple different target classes because of the forwarding mechanism as described above. A work-around in such case is to make as many copies of the singleton class as singletons appear in the application and then perform an equate relationship. But this means to give up the separation of concerns principle since the singleton-aspect would be spread over numerous different classes that contain all the same implementation. If a singleton class containing all singleton specific members were integrated using the compose relationship, the singleton would become a super-class of both target classes. Hence, both target classes would share the same static members.

It should be noted here that another possibility in Hyper/J is to define new classes `ASingleton` and `BSingleton` in different packages than the target classes and then to apply a *merge* relationship. Although this approach is technically possible, it does not modularize the crosscutting code in any way.

3.3.2 Visitor implementations

A **visitor** [GHJV95] encapsulates polymorphic behavior outside of the class hierarchy and is used to implement operations on complex structures (see section 1.2.2.2 for a

[23] It should be mentioned that it is possible to implement the creation by using so-called advice and introspection. However, this implementation has disadvantages like additional type casts whose discussion is outside the scope of this chapter.

more detailed discussion on the crosscutting characteristics in visitor implementations). Let us assume that an object structure of classes A, B, C and D is given. A visitor implements an interface VisitedElement consisting only of the declaration of the double dispatch method accept(Visitor). The interface Visitor declares a method visit for each element to be visited. The operation that is to be performed on the object structure is specified in each implementation of the visitor interface.

In the implementation are different kinds of crosscutting code. First, classes whose objects are to be visited contain the double dispatch method. Hence, this method represents static crosscutting code. Furthermore, the interface Visitor contains the method visit for all classes implementing VisitorElement.

```
interface VisitorElement {
  public void accept(Visitor v);
}
class A implements VisitorElement {
  public void accept(Visitor v) {
    v.visit(this);
  }
  ...
}
class B implements VisitorElement {...}
class C implements VisitorElement {...}
interface Visitor {
    void visit(A node);
    void visit(B node);
    void visit(C node);
}
class ConcreteVisitor implements Visitor {
    void visit(A node) {.....}
    void visit(B node) {.....}
    void visit(C node) {.....}
}
```

Figure 3-8. Visitor implementation in Java.

AspectJ permits up to version 1.1 to modularize the double dispatch method by introducing it to the interface VisitorElement. That means every class implementing this interface will automatically possess such a method (Figure 3-9 illustrates the corresponding implementation). Since version 1.1 this approach is no longer possible because it is not possible to apply member introductions to a number of different classes directly. The only possibility to do so is to perform the corresponding container introduction. In such a case, the type of this is bound to the container's type and the double dispatch characteristic is not fulfilled any longer.

Moreover, AspectJ does not permit (in no version) to handle the different visit methods in the visitor interface. So the interface has to be adapted by hand. As a consequence, a new method has to be defined in VisitorElement for every class that is added to the object structure.

Hyper/J does not permit to modularize the visitor implementation in any way. The compose relationship cannot be used since the double dispatch method requires the dispatch method to be physically present in the target class rather than being inherited.

An `equate` relationship needs to create a new class containing the dispatch method for every visited class. Hence, this does not modularize the crosscutting code.

```
interface VisitedElement {}
interface Visitor {
  visit(A node);
  visit(B node);
  visit(C node);
}
class ConcreteVisitor implements Visitor{
  ...
}
aspect VisitedElementLoader {
  public void VisitedElement+.accept(Visitor v)
  {v.visit(this);}
}
aspect VisitedElementConnector {
  declare parents: A implements VisitedElement;
  declare parents: B implements VisitedElement;
  declare parents: C implements VisitedElement;
}
```

Figure 3-9. Visitor implementation in AspectJ (up to version 1.1).

3.3.3 Decorator implementations

A **decorator** [GHJV95] is used to extend the functionality of a single object during run-time. The typical implementation of a decorator in Java for a given class `A` is shown in Figure 3-10. An interface is extracted from `A` and an abstract decorator implements that interface. The abstract decorator has an instance variable of the implemented interface to which all incoming messages are forwarded to. Concrete decorators extend the abstract decorator and override the methods they need to adapt.

If more than one class is decorated in an application (which is the usual case) all occurring decorators in Java look like this. Hence, the static crosscutting code consists of the following elements: The extracted interface of the decorated class and a class that implements the interface and forwards all messages. The concrete decorators are application specific and, therefore, usually differ from application to application. Hence, concrete decorators are not part of the static crosscutting code.

The problem with decorator implementations is that although they contain a lot of static crosscutting code, they do not contain any fixed implementation. The interface `Component` consists of the public methods of the decorated class. Moreover, the type of the component within the abstract decorator depends on the class to be decorated. Although it is known that an abstract decorator is supposed to only forward all incoming messages to the component, the corresponding signatures are not known since any arbitrary class can be decorated.

In AspectJ, it is not possible to define a decorator aspect independently of the classes to which it is applied. A mechanism is needed that adds the public methods of a class to an interface and adds default implementations to a class that implements this interface.

Since AspectJ does not permit to bind method names nor types at weave time, the introduction implementation is not sufficient to perform such a task[24].

```
class A implements Component {
  public void doSomething() {...}
}
interface Component {
  void doSomething();
}
class AbstractDecorator implements Component {
  public Component component;
  public void doSomething() {
    component.doSomething();
  }
}
class ConcrDecorator extends AbstractDecorator
{
  ....
}
```

Figure 3-10. Decorator implementation in Java.

The argumentation why Hyper/J does not permit to modularize a decorator implementation is similar. Hyper/J cannot extract an interface out of a class and introduce a default implementation of methods with unknown signatures.

3.3.4 Summary so far

The previous examples illustrated typical occurrences of crosscutting code in object-oriented programs. The examples have in common that the crosscutting code varies every time it occurs (cf. [HaUn02a] for a further discussion). In the singleton example the return types vary, in the visitor example the parameter types of the visit methods vary and the number of introductions depends on the number of visited classes. In the decorator example method signatures vary. However, it has been argued why occurrences of such variations are still part of crosscutting code and, thus, are to be modularized using aspect-oriented techniques.

It has been shown that neither the introduction implementation in AspectJ nor the implementation in Hyper/J is sufficient to separate the crosscutting code in single modules. Instead, both implementations force the developer to spread pieces of code over different modules. Thus, neither AspectJ nor Hyper/J follow the principle of separation of concerns. Nevertheless, there are numerous situations where both implementations satisfy the needs at hand. Hence, an extension of the current implementation mechanisms is needed that permits to perform introductions in the known way and furthermore solves the mentioned problems.

[24] It should be noted here that the mechanism for dynamic crosscutting offered by AspectJ permits to decorate classes in a different way by using the reflection API for dynamic crosscutting. However, this approach is highly complex and more a work-around than an acceptable solution.

3.4 Sally

Sally [HaUn03a] is a general-purpose aspect language extending the programming language Java. It is designed to overcome the previously discussed problems. The main focus of Sally is to handle the problem of reusability: Sally gives developers the opportunity to specify reusable aspects to a certain extent. Reusability of aspects means to specify aspects in one single module which can be woven to a number of different situations or applications. For example, according to the previous section, the intention of Sally has been to develop a system that provides language features that permit to specify a reusable visitor: A visitor implementation that can be bound to a number of different class hierarchies without the need to implement redundant code. Sally permits to specify aspects which are generic enough to be applied to different situations. This thesis describes this by the term **generic aspect**.

The design of Sally was highly inspired by AspectJ and shares a number of properties with it. Sally provides

- The composition of aspects, which is completely performed at compile-time,
- The adaptation of method calls as well as method definitions, and
- Language constructs similar to AspectJ's pointcut language, which permit to specify join points in a declarative manner.

On the other hand Sally also differs in a number of details from AspectJ:

- Sally does not provide a new language construct (like aspect) that serves as container for the new languages constructs and that provides its own namespace. Instead, in Sally all new language constructs are embedded into the language construct `class`.
- Pointcuts in Sally serve to specify crosscutting of behavioral as well as structural adaptation. I.e. there is no distinction of join point descriptions that are used for advice and introductions.
- Sally does not provide object-specific pointcuts like for example the `this` pointcut in AspectJ.
- Sally provides some variabilities within the specification of advice and introduction as well as in the pointcut definition.

The pointcut language permits to specify queries on the application using a query language based on a logical programming language. Thereto, the pointcut language relies on information available at compile-time.

In contrast to other general purpose approaches like for example AspectJ, the pointcut language of Sally is used in order to select structural elements as well as behavioral elements that crosscut the application. I.e. the pointcut language of Sally is used to specify those types to which additional declarations should be added to as well as to specify those locations where certain behavior needs to be executed. So, the pointcut language works on a structure reflecting on the static elements of the program. A similar approach can be found for example in OpenC++ [Chib95] and its corresponding Java implementation in OpenJava [TCIK00], which both adapt an application based on its static elements. The pointcut language of Sally permits to select join points not only based on properties that are directly available at such join points, but also based on the relationship between different join points. For example a method definition join point can be selected based on its (statically computed) calls occurring within a different method declaration.

In contrast to other aspect-oriented approaches like for example AspectJ, Sally does not provide the specification of dynamic properties within its pointcut language: If runtime specific information is needed to determine whether or not aspect-specific code is to be executed, programmers needs to specify this on their own using ordinary language constructs provided by Java. Furthermore, the pointcut language of Sally only provides static information to the aspects. As a consequence, the mechanism of context exposure in Sally works quite different from AspectJ: In Sally there is no runtime system passing objects to aspects and advice.

3.4.1 Decomposition of the Underlying Application

Sally decomposes the application into a number of join points that represent the syntactical elements given by the grammar of the programming language Java and the additional language constructs provided by the grammar of Sally (which extends the Java grammar). I.e. the syntactical elements in the application represent the information which can be potentially used for the specification of those join point which are to be adapted. However, Sally provides only API support for a few of them. These elements correspond to the most frequently used elements in AspectJ. All other elements can also be used within the pointcut specification, however less comfortably.

In addition to the pure syntactical information, Sally also provides the available typing and scoping information for each element. To do so, the Sally compiler extracts all static type and scoping information and permits to access them in corresponding pointcut expressions. The main difference to approaches which provide compile-time reflection (like for example OpenJava [TCIK00]) is that Sally reflects not only on the structure of elements like classes or method (which is known as **structural reflection** [Maes87]) but also on inner elements like method calls or `new` calls.

Figure 3-11 illustrates in a simplified way how an application of the **observer design pattern** [GHJV95][25] is decomposed into its join points in Sally. The application consists of a class `Person` which has just one field `name` of type `String` and a class `PersonViewer`. `Person` extends the abstract class `Subject`, which permits to attach and detach observer instances by providing corresponding methods `attach` and `detach`. After assigning a new object to field `name`, the method `notifyObservers` is invoked which informs observers about the new state. Class `PersonViewer` extends class `Observer` and defines the method `update` declared in `Observer`. The method simply prints a person's name on the screen.

The resulting graph consists of multiple nodes, each representing an element as it appears in the abstract syntax tree (AST). In fact, each node is an object, i.e. each node has its own object identity and each expression has a corresponding type which can be used for comparing join points as well as reflecting on a join point's characteristics. For

[25] A typical implementation of the observer design pattern in Java does hardly make use of extends relationships since Java provides only single inheritance. See [Gran98] for an exhaustive discussion on typical implementations of the observer in Java.

reasons of simplicity the object identity is skipped in Figure 3-11 and is just represented by three dots in front of each object's type. Furthermore, some of the direct relationships between the objects in the graph do not exist directly: For example a field assignment is not directly referred by a method declaration, but indirectly via an object of type `MethodBody`.

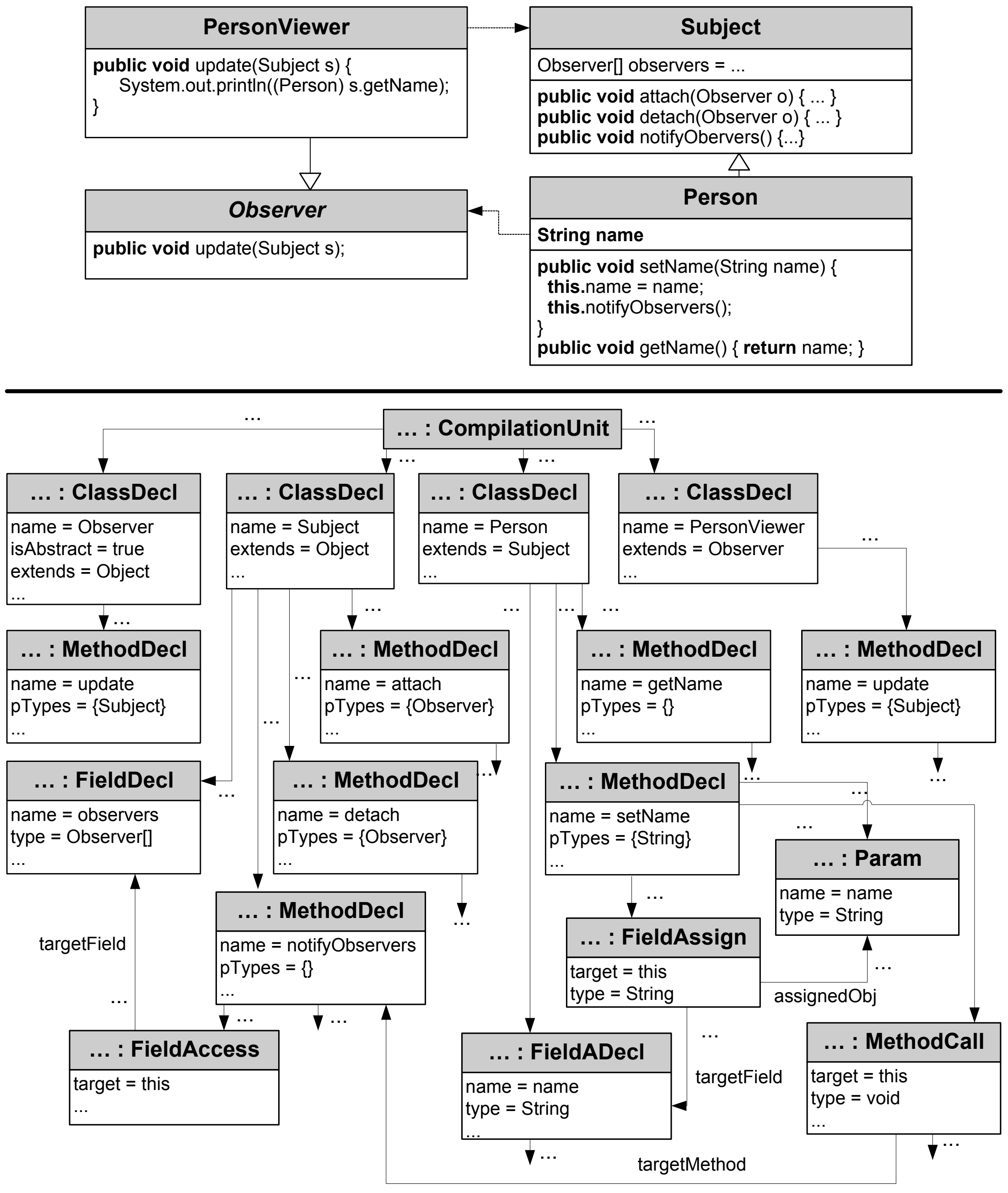


Figure 3-11. Decomposition of a simple Observer implementation.

The type of each object is given by the grammar definition of the underlying programming language. In Sally the type of each node is defined by a corresponding Java grammar element. For example an object which represents a class declaration is of type `ClassDecl`, an object representing a method declaration is of type

MethodDecl, an object representing a field declaration is of type FieldDecl, a field assignment is of type FieldAssign, a formal parameter is represented by an object of type Param and a method call is represented by a method of type MethodCall. There is one object (of type CompilationUnit) from which all nodes in the graph can be reached.

Most of the relationships in the graph come directly from the syntax definition. For example the object relationship between a class declaration and a method declaration can be directly deduced from the programming language's syntax. However, there are also relationships between objects due to the static type system and the single dispatching in Java (cf. for example [Bruc02] for a discussion on single dispatching in general, and [CLCM00] for dispatching in Java). For example, there is a node representing the method call inside the method setName defined in class Person to the method notifyObservers. Due to single dispatching the target method of the method call is statically computed because of the target object's static type and the static types of its parameters. The target object is the self-reference this which has the static type Person because the method is defined in class Person. There are no parameters passed to the method. Class Person does not declare a method notifyObservers. Hence, the target method is the method declaration whose signature corresponds to the statically computed signature which is declared in the nearest class extended by Person. Since class Person extends Subject (which itself extends Object) and Subject contains a method declaration (and definition) matching the signature as computed before, this method is determined to be the target method of the method call. In correspondence, the object representing the method call has a reference to the object representing the method notifyObservers in class Subject. Due to single dispatching, and since Java does not permit **covariant return types** in overriding methods (cf. [Bruc02]), the return type of the method call is known to be void. As a result of dynamic binding, the actual executed method may vary during the execution of a program. The method notifyObervers in Subject is not declared to be final. Consequently, this method might be overridden in subclasses of Person. In case notifyObservers is invoked on an instance of such a subclass, the overriding method is invoked.

In the same way as explained above every node representing an expression (i.e. which has a type) has statically computed type information that can be used by the pointcut language to reason on the application. Hence, all expressions in Java like method calls, type casts, or field and variable assignments have corresponding type information stored in the graph resulting from the decomposition.

Java's scoping information is being used in order to determine relationships between join points referring to certain objects. For example, the relationship between the field assignment in method setName and the field name is being established from the underlying scope information.

3.4.2 Pointcut Specification

In Sally, the design of the pointcut language was mainly influenced by the approach of **logical meta programming** (cf. for example [DVDH99, Wuyt01]): A pointcut is a logical rule whose body refers to other pointcuts or system predicates applied to join points. On the implementation level Sally makes use of **TyRuBa** [DV98] which is a

Prolog-like logical programming language (see [StSh94] for an introduction into Prolog). A pointcut specifies a selection of join points that can be adapted by aspects. Pointcuts export parameters to those units that adapt the corresponding join points.

<pointcutDeclaration>	→	(*<abstractPointcutDeclaration>* | *<pointcutDefinition>*) ";"
<abstractPointcutDeclaration>	→	"abstract" "pointcut" *<pointcutHeader>*
<pointcutDefinition>	→	["final"] "pointcut" *<pointcutHeader>* "=" *<pointcutExpression>*
<pointcutHeader>	→	*<pointcutIdentifier>* "(" [*<logicalVarList>*] ")"
<pointcutIdentifier>	→	*<javaIdentifier>*
<logicalVarList>	→	*<logicalVariable>* { "," *<logicalVariable>* }
<pointcutBody>	→	*<pointcutExpression>*
<pointcutExpression>	→	*<disjunction>*
<disjunction>	→	*<conjunction>* { "||" *<conjunction>* }
<conjunction>	→	*<simpleExpression>* { "&&" *<simpleExpression>* }
<simpleExpression>	→	"(" *<pointcutExpression>* ")" | *<simplePointcutExpr>* | *<negation>* | *<findallExpr>*
<negation>	→	"!" *<pointcutExpression>*
<findAllExpr>	→	"FINDALL" "("*<simplePointcutExpr>* "," *<term>* "," *<term>* ")"
<simplePointcutExpr>	→	[*<target>* "."] *<pointcutIdentifier>* "(" [*<termList>*] ")"
<target>	→	"thisClass" | *<fullQualifiedType>*
<term>	→	*<logicalVariable>* | *<constant>* | *<list>*
<termList>	→	*<term>* { "," *<term>* }
<list>	→	"[" [*<listBody>*] "]"
<listBody>	→	*<termList>* ["|" *<term>*]

Figure 3.1. Syntax of Pointcut Definitions.

3.4.2.1 Syntax of Pointcuts

Pointcuts are declared inside classes, i.e. pointcuts are syntactically defined on the same level like methods or fields. The syntax of pointcut declarations is defined according to the context-free grammar as specified in Figure 3.1 using the keyword `pointcut`. A pointcut may be declared abstract, i.e. the pointcut is declared but not defined. Also, a pointcut may be concrete, i.e. a pointcut is declared as well as defined within its surrounding class.

Each pointcut has a pointcut header consisting of an identifier and a parameter list. The identifier is a simple Java identifier used by other language constructs to refer to the pointcut. The parameter list is a (possibly empty) list of logical variables. The syntax of logical variables is similar to the syntax of logical variables in TyRuBa: A logical variable begins with a question mark followed by a Java identifier. To ease the use of pointcut Sally provides furthermore a special logical variable that simply consists of a question mark. This variable cannot be referred to by other pointcuts within a pointcut expression and is not considered for unification. According to the terminology of logical programming languages, the number of pointcut parameters is called the pointcut's **arity**. The name of a pointcut and its arity represent a pointcut's **signature**, which is used to validate and evaluate a pointcut.

Pointcut definitions have a pointcut body in addition to their (abstract) declarations. The pointcut body is a combination of simple pointcut expressions. A simple pointcut expression consists of a target class (which is according to the Java syntax a fully qualified type), a pointcut identifier, and a list of terms. A term is a logical variable, a constant, or a list, whereby the syntax of all these constructs corresponds to the syntax of logical variables, constants, and lists in TyRuBa.

Simple pointcut expressions can be combined via disjunction ("`||`"), conjunction ("`&&`"), or cuts ("`!`"), which comply with the corresponding operators in logical programming languages like TyRuBa or Prolog. The syntactical definition of the FINDALL construct corresponds to the `FINDALL` predicate in logical programming languages (cf. e.g. [StSh94]): A FINDALL expression consists of the keyword `FINDALL`, followed by a simple pointcut expression and two terms. The semantics of conjunction, disjunction, and FINDALL corresponds to the semantics in Prolog. The semantics of negation corresponds to the semantics of **cuts** in Prolog.

3.4.2.2 Validation of Pointcuts

In Sally pointcuts are class members which are evaluated at compile time. According to the grammar specified in Figure 3.1, a pointcut declaration is either abstract, i.e. the pointcut specifies only the name of the pointcut and its arity, or concrete. A concrete pointcut consists of a number of pointcut expressions, which typically consist of **simple pointcut expressions** (see Figure 3.1). Such simple pointcut expressions refer to other pointcut declarations. These pointcut declarations are either built-in pointcuts (which will be described in the following sections) or user-defined pointcuts which are defined within the inheritance hierarchy of the corresponding class. Alternatively, a simple pointcut expression can refer to global pointcuts which are defined in a class outside the inheritance hierarchy of the current pointcut definition.

In Sally there are two different ways of how pointcuts can be addressed: Either pointcuts are addressed **absolutely** or **relatively**. An absolutely addressed pointcut is a pointcut expression where the target includes a fully qualified type (i.e. the type is identified by its package name and type name). A relatively addressed pointcut refers to a pointcut which is defined in the same class hierarchy as the surrounding pointcut. The target of a relatively addressed pointcut is specified using the keyword `thisClass` or by simply not specifying a target[26]. This thesis uses the term **target** for both, the fully qualified class in an absolutely addressed pointcut and `thisClass` in a relatively addressed pointcut.

After parsing and type checking a Sally application the **validity** of pointcuts is checked. This validity check is a static analysis of the Sally construct `pointcut`, which restricts the usage of pointcuts in a number of ways similar to the known type-checking

[26] This corresponds to the use of `this` in languages like Java where messages which do not have an explicit target are assumed to have `this` as target. In contrast to `this`, `thisClass` refers to a class and not to an object.

rules in Java. Such restrictions pertain on the one hand to the declaration and definition of pointcuts, i.e. they are related to pointcut signatures. On the other hand, the restrictions are related to pointcut expressions.

The first restrictions (or static rules) related to the declarations and definitions of pointcuts are similar to the restrictions of member declarations and definitions in the programming language Java.

1. It is not allowed to declare two pointcuts with the same signature, i.e. with the same name and the same arity defined in the same class. The reason for such restrictions lies in the way of how pointcuts are evaluated which will be explained later in this chapter (see section 3.4.2.4).
2. If a class contains an abstract pointcut, the class also has to be declared abstract. This prevents programmers from accidentally addressing a pointcut outside the inheritance hierarchy, which is not already fully specified, i.e., which does not refer exclusively to defined (and not only declared) pointcuts.
3. Pointcuts that are declared as `final` cannot be overridden. I.e. it is not possible that a subclass of the one containing the pointcut definition contains a pointcut with the same signature as the one declared as final.
4. A class has to be declared abstract if there is at least one abstract pointcut in one of its superclasses, which is not overridden along the inheritance path.

The following restrictions pertain to the validity of pointcut expressions inside a pointcut definition. A pointcut definition is valid, if it corresponds to the restrictions above, and if each pointcut expression occurring in the pointcut definition's body is valid. The validity of pointcut expressions is restricted as follows.

5. A pointcut expression that addresses a pointcut absolutely is valid, if the target class exists (i.e. if there is a class declaration whose name matches the fully qualified type), the target class is not abstract, and either the target class or one of its superclasses defines a pointcut whose signature corresponds to the pointcut used in the pointcut expression.
6. A relatively addressed pointcut is valid, if the target class or one of its superclasses contains a pointcut declaration with the same signature as used in the pointcut expression. According to rule 2 this implies that if the corresponding pointcut is declared but not defined in the class itself or its superclass the class itself has to be declared abstract.

The consequence of these rules is that a class, which is not abstract, always contains pointcuts that are not abstract themselves. I.e. if a pointcut is absolutely addressed by a pointcut expression in this class, there is a pointcut definition in the class or its superclasses. The rules correspond to the usual subtyping rules in statically typed languages like Java. The intention is to prevent developers from accidentally referring to undefined pointcuts.

3.4.2.3 Built-In Pointcuts

Sally contains a number of predefined pointcuts (or system predicates) which permit to reason on the application which is about to be woven. The predefined pointcuts are members of the class `BuiltinRules` (which represents the API of predefined pointcuts).

In general, pointcuts refer to declaration units like class, method or field declarations, or expressions like method calls. In order to ease the use of pointcuts, Sally typically assigns the same pointcut identifier to more than one pointcut of the same kind, each with a different arity. One essential characteristic of all predefined pointcuts is that they provide a value representing a join point's identity: This identity refers to the objects from the application's decomposition.

`typeDeclaration(?typeID,?typeName)`
`classDefinition(?cID,?typeName)`
`interfaceDefinition(?cID,?typeName)`
The pointcut `typeDeclaration` with arity 2 determines all type declarations where the id corresponds to `?typeID` and the type name corresponds to `?typeName`. If `?typeID` is bound, `?typeName` is bound to the type name of that identity. Types in Java are either class definitions or interface definitions[27]. The corresponding pointcuts `classDefinition` and `typeDefinition` bind the corresponding variables to class definitions and interface definitions.

`declaredSuperClass(?cID,?scID)`
`superClass(?cID,?scID)`
The pointcuts `declaredSuperClass` and `superClass` determine the super-classes defined for a class identified by `?cID`. The pointcut `declaredSuperClass` binds the variable `?scID` which is declared to be the direct superclass of the one identified by `?cID`. The pointcut `superClass` binds the id of every super-class of the class identified by `?cID` to the variable `?scID`.

`declaredInterface(?typeID,?iID)`
`interface(?typeID,?iID)`
The pointcuts `declaredInterface` and `interface` determine the interface for a given type. If `?typeID` is bound to a class or an interface, the first pointcut binds `?iID` to the identifier of those interfaces extended (in case `?typeID` is bound to an interface) or implemented (in case `?typeID` is bound to a class) directly by `?typeID`. The pointcut `interface` binds the variable `?iID` to all interfaces which are directly or indirectly implemented or extended by the type identified by `?typeID`.

`methodDeclaration(?mID,?cID)`
`methodDeclaration(?mID,?cID,?name)`
`methodDeclaration(?mID,?cID,?name,?rID,?pIDs)`
`methodDeclaration(?mID,?cID,?name,?rID,?pIDs,?pNames)`
There are four built-in pointcuts `methodDeclaration` with the arities 2, 3, 5 and 6. The first pointcut has two parameters. The first one describes a method identifier, the second one a class identifier. If the second parameter is bound, the pointcut binds `?mID` to all method identifiers of those methods which are declared in the class identified by the bound variable `?cID`.

[27] This statement neglects primitive types in Java and is only related to Java since version 1.4 because Sally is built on top of Java 1.4. Since Java version 1.5 provides generic types. These constructs permit types to be instantiated from corresponing generic types.

The pointcut with arity 3 determines all methods which are identified by ?mID having the method name ?name declared in all classes identified by ?cID. Hence, if ?cID is bound the pointcut binds all method identifiers for methods declared in ?cID to the variable ?mID and their corresponding names to the variable ?name. If variable ?name is bound the pointcut binds the variables ?mID and ?cID to all method identifiers and class identifiers where the classes declare a method having the corresponding name.

The pointcut with arity 5 determines all methods with the id ?mID, the name ?name, the return type with id ?rID, and the parameter types ?pIDs that are declared in class ?cID. The parameter types are delivered as a list where the position in the list corresponds to the position in the parameter list as defined in the base program.

The last pointcut with arity 6 binds the parameter names to the list ?pNames.

methodCall(?callID,?methodID)
methodCall(?callID,?cID,?mName)
methodCall(?callID,?cID,?mName,?rTypeID,?pIDs)

The pointcut methodCall determines method calls within the application. These method calls are determined by the semantics of the underlying programming language Java, i.e. under consideration of **single dispatching**. As a consequence, the target method of a method call is determined by the static types of the participating expressions. MethodCall does not expose any of the participating expressions, nor the static types of the expressions which participate in the method call. Instead, only the statically computed target method is determined.

The pointcut methodCall with arity 2 determines for each method call having id ?callID the method ?methodID being invoked by the call.

The pointcut with arity 3 determines all method calls with id ?callID which invoke a method with name ?mName in class ?cID.

The pointcut methodCall with arity 5 determines all method calls with id ?callID that call a method with name ?mName which return a type identified by ?rTypeID with the parameter types ?pIDs in the class ?cID. The parameter types ?pIDs are a list of object identities.

callExpressionTypes(?id,?tTypeID,?mName,?pTypeIDs)

The pointcut callExpressionTypes exposes all static types on the caller side of a method call. The first parameter ?id is bound to the id of the call expression. The id corresponds to the id used in the pointcut methodCall. I.e. by combining a methodCall pointcut with a callExpressionTypes pointcut it is possible to determine how the types in the client code relate to the types in the method declaration. The second parameter is bound to the type's id of the target expression. ?mName is bound to the method name and the list ?pTypeIDs binds the static types of the call parameters.

callExpression(?cExprID,?tExpID,?mName,?pExpIDs)

In contrast to the pointcuts methodCall and callExpressionTypes the pointcut callExpression exposes the expressions which participate in a method call (as target expressions or parameter expressions). The pointcut binds the id of a call expression to `?cExprID`, the id of each target expression to `?tExpID`, the method name to `?mName`, and the expression which represents the parameters in the method call to the list `?pExprIDs`. The id given by the variable `?cExprID` corresponds to the id of the method call as provided by the pointcuts `methodCall` and `callExpressionTypes`.

`constructorDeclaration(?id,?cID)`
`constructorDeclaration(?id,?cID,?pIDs)`
Similar to the `methodDeclaration` pointcut the pointcut `constructorDeclaration` determines for a given class its constructor declarations. The pointcut with arity 2 binds the variable `?id` to the id of the constructor and the variable `?cID` to the identity of the corresponding class declaration.

The pointcut with arity 3 additionally binds the declared parameter type identities to the variable `?pIDs`.

`fieldDeclaration(?fID,?cID)`
`fieldDeclaration(?fID,?cID,?name,?typeID)`
Similar to `methodDeclaration` the pointcut `fieldDeclaration` refers to field declarations within class declarations. The pointcut `fieldDeclaration` with arity 2 determines all field declarations having id `?fID` that are declared in a class (or interface) with id `?cID`.

The pointcut with arity 4 additionally refers to the field name `?name` and the field's type `?typeID`.

`fieldGet(?id,?fID)`
`fieldGet(?id,?fID,?cID)`
`fieldGet(?id,?fID,?cID,?mID)`
The pointcut `fieldGet` determines reading accesses to fields in the application. The pointcut with arity 2 binds the id of the corresponding expression to `?id` and the id of the field which is about to be read to `?fID`. The pointcuts with arity 3 and arity 4 additionally bind the id of the class where the reading of the field is defined in to `?cID` and the method which contains the reading access to the variable `?mID`.

The target field of the expression representing the reading access of the field is determined by the underlying programming language Java, i.e. based on its type system.

`fieldSet(?id,?fID)`
`fieldSet(?id,?fID,?cID)`
`fieldSet(?id,?fID,?cID,?mID)`
The pointcuts `fieldSet` determine writing accesses to fields. The pointcut with arity 2 binds the id of the corresponding expression to `?id` and the id of the field which is about to be written to `?fID`. The pointcuts with arity 3 and arity 4 additionally bind the id of the class containing the assignment to the variable `?cID` and the method containing the assignment to `?mID`.

`fieldSetTypes(?id,?fID,?targetTID,?assignedTID)`

Similar to the `methodCall`, pointcut `fieldSet` neither exposes any information about the type of the object which is assigned to a field nor the type of the object to whose field an object is assigned to. I.e. it is not possible to determine whether the static type of the assigned object corresponds to the field's type or whether the static type of the assigned object is a subtype of the field's type.

The pointcut `fieldSetType` determines for a field assignment the static type of the assigned object as well as the static type of the object to whose field an object is assigned. The static type corresponds to the static type as determined by the type system of Java. The variable `?id` is bound to the same id as the corresponding `fieldSet` pointcut. The variable `?targetTID` is bound to the id of the static type of the object owning the field. The variable `?assignedTID` is bound to the static type of the object which is assigned.

3.4.2.4 Evaluation of Pointcuts

The evaluation of the known logical constructs like concatenation, disjunction, or negation works in the same way as in logical programming languages (like for example explained in [StSh94]).

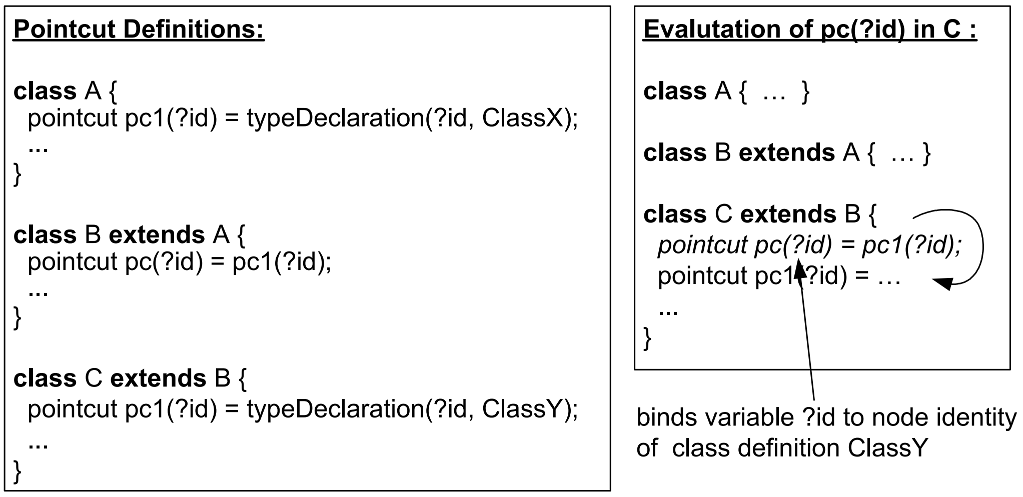
On the other hand, pointcuts are (similar to the pointcut language in AspectJ) declared in constructs which potentially participate in an inheritance relationship, i.e., pointcut can be inherited along an inheritance structure. In contrast to AspectJ, Sally does not provide any special construct for the declaration of aspects; in Sally pointcuts are declared and defined in classes. The reason for not introducing a new language feature lies in the complexity of the resulting language; in AspectJ the new language construct `aspect` comes with a number of new mechanisms which need to be understood and therefore increase the complexity of the language (cf. for example [HBU01, HaUn01]).

The main difference between ordinary logical terms and pointcuts in Sally is that pointcuts potentially participate in an inheritance relationship to permit an incremental modification pointcuts[28]. Pointcut expressions can refer to pointcut declarations in the inheritance hierarchy which are not yet defined (but declared) or which are defined in a subclass by addressing a pointcut relatively. The intention is to permit developers to specify aspects incrementally: An aspect having a number of pointcuts can be specified as complete as possible but leaving some **hooks** to the developer. I.e. it is possible to specify an abstract aspect without knowing all details of how and where the aspect contributes to the application. Also, an incompletely defined aspects permits to weave an aspect to different join points without the need to perform destructive changes on the aspect, i.e. without the need to modify the source code of the aspect. As a consequence, aspects can be specified in one place containing all features belonging to that aspect and the code which connects the aspect to the application can be specified in a different location.

A pointcut definition is evaluated by evaluating its body that consists of a logical term which itself consists of a number of simple pointcut expressions. Since the

[28] Another distinction is of course that a pointcut header only is permitted to contain logic variables.

combination of pointcut expressions, i.e. concatenation, disjunction, and negation of pointcut expressions correspond to the usual way of evaluating logical terms, this is not discussed in this chapter (see e.g. [StSh94]).



```
Pointcut Definitions:

class A {
  pointcut pc1(?id) = typeDeclaration(?id, ClassX);
  ...
}

class B extends A {
  pointcut pc(?id) = pc1(?id);
  ...
}

class C extends B {
  pointcut pc1(?id) = typeDeclaration(?id, ClassY);
  ...
}
```

```
Evalutation of pc(?id) in C :

class A { ... }

class B extends A { ... }

class C extends B {
  pointcut pc(?id) = pc1(?id);
  pointcut pc1(?id) = ...
  ...
}
```

binds variable ?id to node identity of class definition ClassY

Figure 3-12. Pointcut definition and evaluation of pointcut pc in C.

However, the way simple pointcut expressions are evaluated differs from the way logical terms are evaluated. A pointcut is evaluated for a concrete class. According to the validation rules described in section 3.4.2.2 a concrete class contains only concrete pointcuts.

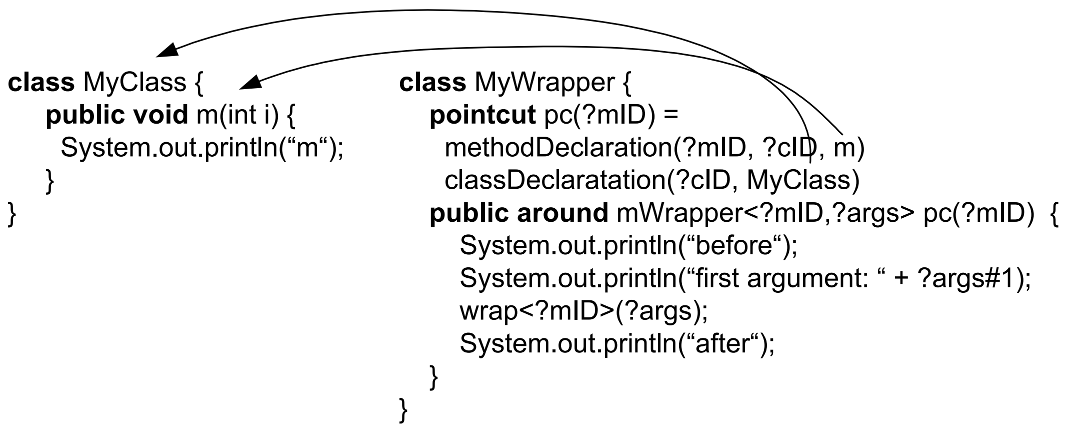
Once a pointcut for a concrete class is evaluated, the corresponding concrete class becomes an implicit parameter of the pointcut expression: Each relatively addressed pointcut becomes evaluated with this implicit parameter. The pointcuts to be chosen for each relatively addressed pointcut are those pointcuts that match the corresponding signature and that are defined last in the inheritance hierarchy. This kind of pointcut binding corresponds to the binding of object members in object-oriented systems.

Figure 3-12 illustrates a pointcut definition with the corresponding pointcut evaluation. The pointcut `pc1` in class `C` overrides the pointcut `pc` defined in class `A`. Consequently, when the pointcut `pc` in class `C` is evaluated (which relatively addresses pointcut `pc1` in class `B`) the pointcut `pc1` is being evaluated. Hence, the variable `?id` in pointcut `pc` is bound to the identity of the node representing the type definition of `ClassY`.

3.4.3 Join Point Wrappers

Join point wrappers are constructs for defining the adaptation of the base application's behavior and can be (to a certain extend) compared to advice in AspectJ. They can be applied to method definitions, method calls, field accesses and field assignments. For each of those different kinds of join points Sally defines corresponding wrapper constructs.

Syntactically, join point wrappers are method-like constructs consisting of a header and a body. The header consists of a modifier (in correspondence to the access modifiers `public`, `private`, and `protected` in Java), the keyword `around`, a name, a parameter list (representing static parameters of the corresponding pointcut) and a corresponding pointcut.



```
class MyClass {                    class MyWrapper {
   public void m(int i) {             pointcut pc(?mID) =
     System.out.println("m");           methodDeclaration(?mID, ?cID, m)
   }                                    classDeclaratation(?cID, MyClass)
}                                     public around mWrapper<?mID,?args> pc(?mID) {
                                         System.out.println("before");
                                         System.out.println("first argument: " + ?args#1);
                                         wrap<?mID>(?args);
                                         System.out.println("after");
                                      }
                                   }
```

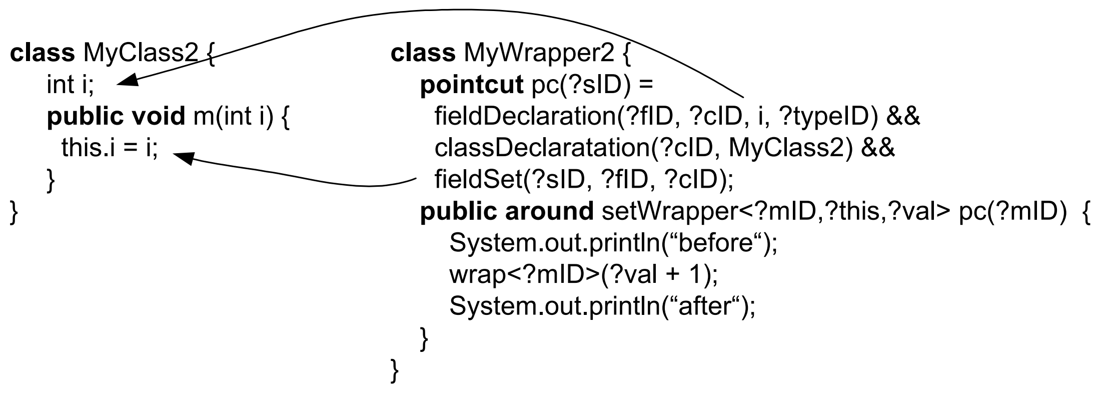
Figure 3-13. Method defininition wrapper for method `m`.

Join point wrappers have three different kinds of parameters. First, there is one special parameter (which is the first parameter) that represents the target join point that is to be wrapped. Second, there are parameters that represent inherent parameters of the corresponding join points. Such inherent parameters are for example parameters of method calls or the assigned value of a field assignment join points. Third, there are parameters that explicitly come from the corresponding pointcut.

Within the wrapper body, all parameters can be used and are replaced when the aspect is woven to the target. `This` within the wrapper body corresponds to `this` as it occurs in the position in the source code where the join point occurs. Furthermore, within the wrapper body the special keyword `wrap` followed by a logical variable represents the join point being wrapped as well as a parameters list of the corresponding wrapped join point. This `wrap` is being replaced at compile time by a corresponding method call to a method representing the original join point. The number of arguments being passed to `wrap` corresponds to the number of arguments of the corresponding kind of join point. Furthermore, it is possible to pass a logic variable as arguments. The compiler then passes the corresponding list represented by the variable as argument to the wrapped join point.

Figure 3-13 illustrates a method wrapper definition for method definition join points. The wrapper has the name `mWrapper` and refers to the pointcut `pc` with arity 1. The pointcut's first variable is bound to the variable `?mID`. The corresponding pointcut selects all method definitions with the name `m` in class `MyClass`. Since there is only one method with such a name in the corresponding application, the pointcut binds just one single identity to the corresponding variable `?mID`.

The variable `?mID` is being passed as the first variable to the wrapper. Hence, the values of this variable represent the targets of this wrapper. The second parameter `?args` is a parameter representing the arguments of the method. The argument `?args` is an ordered list of arguments. Within the body, each element of the list can be used by explicitly referring to its position in the list. In Figure 3-13 the first argument of the method is being used: The notation `#` followed by a number `n` represents the `n`-th argument in the list. Consequently, the expression `?args#1` represents the first argument of method `m`, which is of type `int` in the example. The variable `?args` is being used as argument for `wrap`. Consequently, the effect of the wrapper `mWrapper` is that it prints out the string `before` as well as the value of the method's first argument, then the original method is executed, and finally it prints out the string `after`.

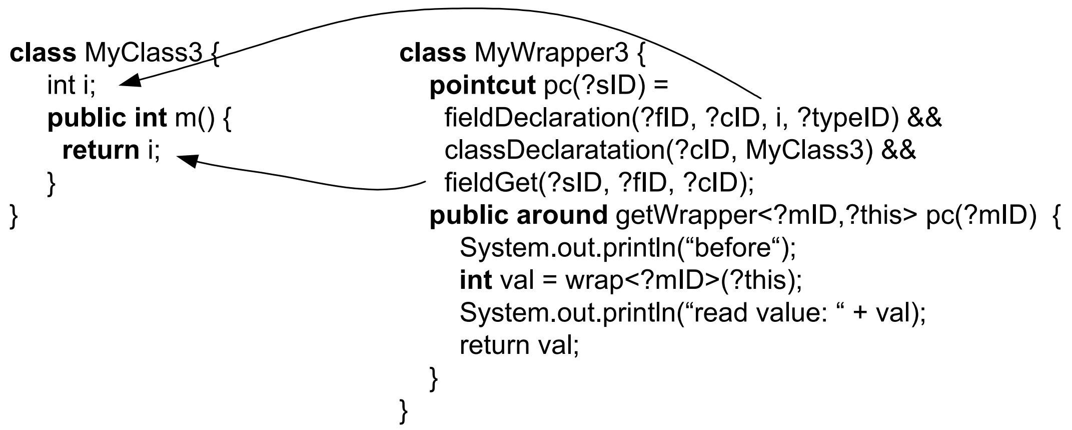


```
class MyClass2 {
  int i;
  public void m(int i) {
    this.i = i;
  }
}
```

```
class MyWrapper2 {
  pointcut pc(?sID) =
   fieldDeclaration(?fID, ?cID, i, ?typeID) &&
   classDeclaratation(?cID, MyClass2) &&
   fieldSet(?sID, ?fID, ?cID);
  public around setWrapper<?mID,?this,?val> pc(?mID) {
     System.out.println("before");
     wrap<?mID>(?val + 1);
     System.out.println("after");
  }
}
```

Figure 3-14. Field set wrapper for field `i`.

Figure 3-14 illustrates a field setter wrapper definition. The corresponding pointcut selects all field sets of field `i` defined in class `MyClass2`. Since there is only one assignment to field `i` (defined in class `MyClass2`) the pointcut binds just one single identity to the corresponding variable `?sID`. In contrast to method definition wrappers, field set wrappers receive 3 inherent join point parameters. The first one represents the field set identity, the second one represents the object whose field is being changed, and the third one represents the object that is being assigned to the field. In Figure 3-14 `before` and `after` are being printed out before and after the field access. The wrapper increases the integer that is assigned to the field by one within the wrap expression.



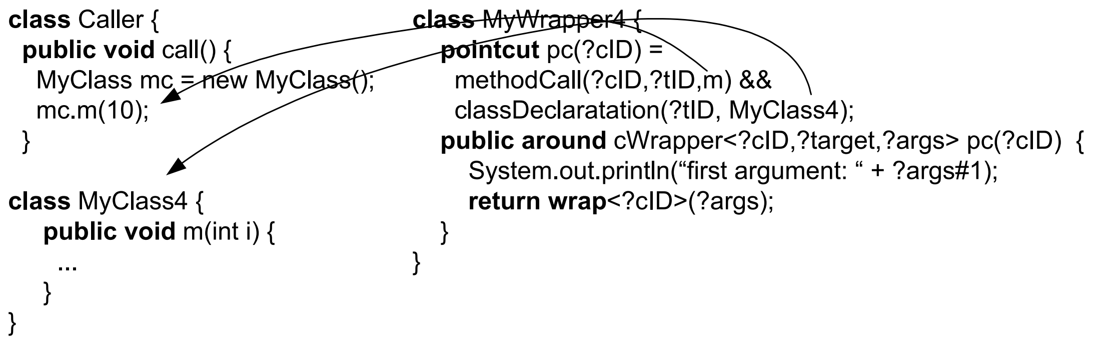
```
class MyClass3 {
  int i;
  public int m() {
    return i;
  }
}
```

```
class MyWrapper3 {
  pointcut pc(?sID) =
   fieldDeclaration(?fID, ?cID, i, ?typeID) &&
   classDeclaratation(?cID, MyClass3) &&
   fieldGet(?sID, ?fID, ?cID);
  public around getWrapper<?mID,?this> pc(?mID) {
     System.out.println("before");
     int val = wrap<?mID>(?this);
     System.out.println("read value: " + val);
     return val;
  }
}
```

Figure 3-15. Field get wrapper for field `i`.

Field access wrappers differ from field set wrappers: There is no third parameter being passed. Figure 3-15 illustrates the definition of such a field access wrapper. The second variable represents the target object whose variable is being read. The wrapper prints out the string `before`. Then the wrapper executes the original join point that returns an `int` and assigns this value to the variable `val`. Finally, the wrapper prints out and returns the field's value.

Method call wrappers are the forth kind of join point wrappers. They are similar to method definition wrappers: They have a variable representing the arguments of the method call (which is the third variable). The second variable represents the target object to which a message is being sent. Figure 3-16 illustrates the declaration of a method call wrapper. The pointcut `pc` refers to all method calls to method `m` defined in class `MyClass4`. Since there is only one such method call in the example, the variable `?cID` is bound to that corresponding identity. The parameter `?target` refers to the target object of the method call, the parameter `?args` represents the ordered list of arguments passed to the method call. The wrapper simply prints out the first argument (i.e. the value `10`) and executes the original method.



```
class Caller {                           class MyWrapper4 {
  public void call() {                     pointcut pc(?cID) =
    MyClass mc = new MyClass();              methodCall(?cID,?tID,m) &&
    mc.m(10);                                classDeclaratation(?tID, MyClass4);
  }                                        public around cWrapper<?cID,?target,?args> pc(?cID) {
                                              System.out.println("first argument: " + ?args#1);
class MyClass4 {                              return wrap<?cID>(?args);
    public void m(int i) {                 }
      ...                                }
    }
}
```

Figure 3-16. Method call wrapper for call to `m` in `MyClass`.

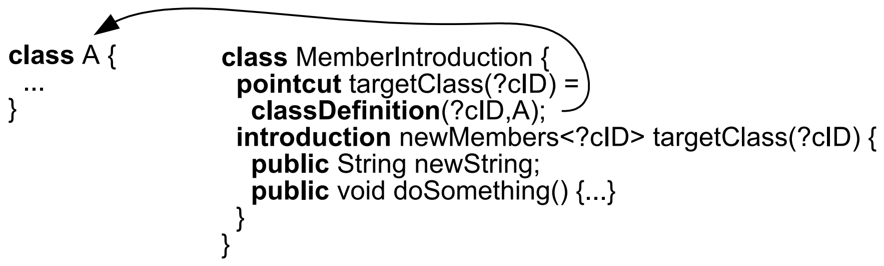
3.4.4 Introductions

Introductions are constructs designed to perform adaptation of types and type hierarchies. Introductions permit to add new method declarations or definitions to types and permit to add new elements to type hierarchies.

In Sally, introductions are members of classes, and as such they consist of a header and a body. The header consists of the keyword `introduction` followed by an identifier, a number of parameters, and a referenced pointcut. The body defines the members that are to be introduced and looks like a usual class body. An introduction's body corresponds to a class body with the exception that logic variables from the header are permitted. Furthermore, the additional `extends` keyword can be used in order to introduce additional parent relationships to the target types.

The replacement of the logic variables within introductions corresponds to the replacement as explained for wrappers in the previous section: The variables are replaced with their corresponding values when the aspect is woven. The only exception is the use of variables representing lists in the header of signatures: If two list parameters are being used within method headers, one list represents the list of type names, the other one represents the list of parameter names. In correspondence to join point wrappers, the first parameter of the introduction represents the target class.

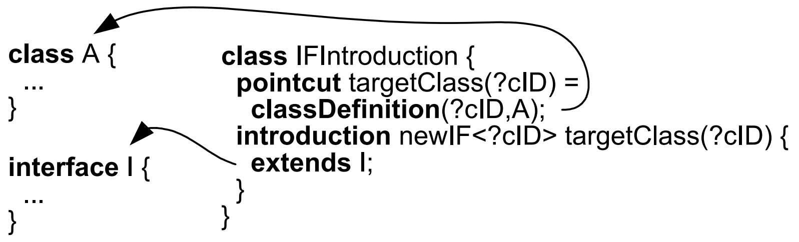
In Figure 3-17 the class `MemberIntroduction` contains an introduction named `newMembers` that has a single parameter `?class` and a corresponding pointcut `targetClass`. In the example the introduction body consists of the instance variable `newString` and the method `doSomething` (for reasons of simplicity the method's body is not shown).



```
class A {            class MemberIntroduction {
  ...                  pointcut targetClass(?cID) =
}                        classDefinition(?cID,A);
                       introduction newMembers<?cID> targetClass(?cID) {
                         public String newString;
                         public void doSomething() {...}
                       }
                     }
```

Figure 3-17. Member introduction in Sally.

The pointcut `targetClass` binds the parameter `?cIS` to the class `A` using the pointcut `classDefinition` with arity 2. Consequently, class `A` is the target of the introduction, i.e. the class to whom the members `newString` and `doSomething` are introduced.



```
class A {            class IFIntroduction {
  ...                  pointcut targetClass(?cID) =
}                        classDefinition(?cID,A);
                       introduction newIF<?cID> targetClass(?cID) {
interface I {            extends I;
  ...                  }
}                    }
```

Figure 3-18. Parent introduction in Sally.

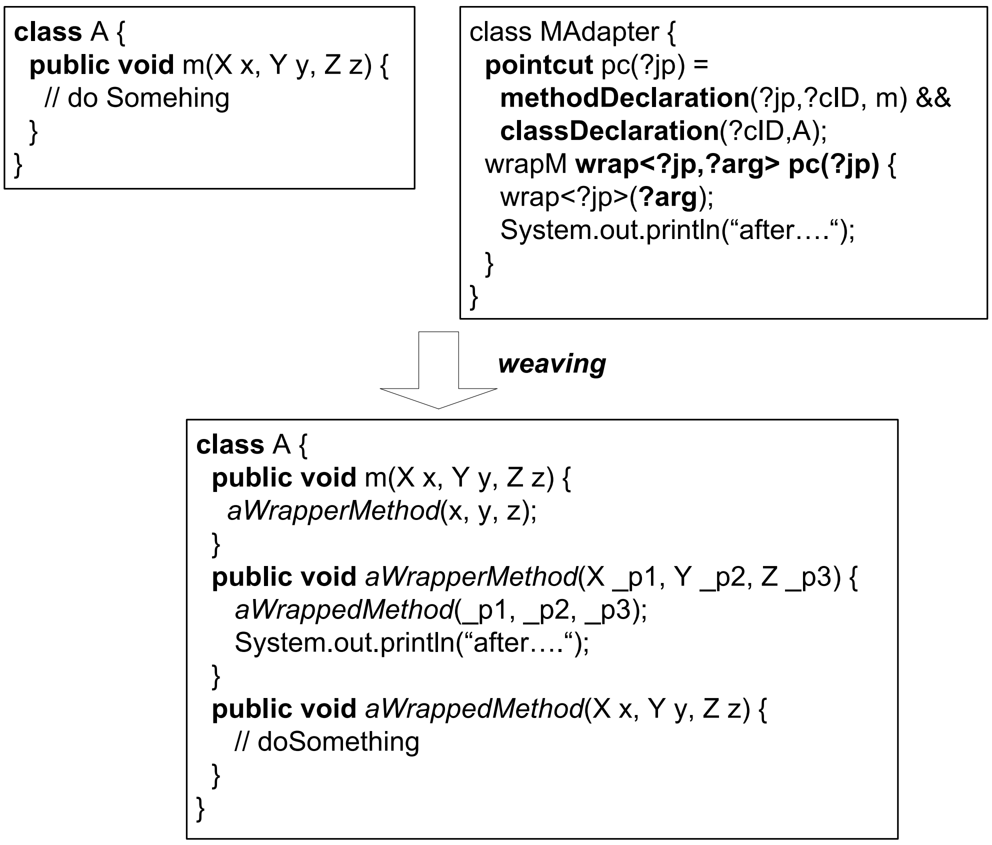
Parent introductions in Sally are declared in the body of an introduction (in its first line) with the help of the keywords `extends` followed by the name of the type being introduced as a supertype or a logic variable. In case the target type is a class and the type being introduced refers to an interface, then a corresponding `implements` relationship is added to the target class. In case the target type is an interface and the introduced type is an interface, a corresponding `extends` relationship is added to the target interface[29].

Figure 3-18 illustrates the application of a parent introduction. The introduction's body declares that the interface `I` should be added to the target class `A`. Consequently, after weaving class `A` is a subtype of type `I`.

[29] Sally does not permit to introduce classes as supertypes. Hence, it is neither possible to add an extends relationships to a class, nor to add a class to the inheritance relationship of an interface.

3.4.5 Weaving

Weaving in Sally is achieved at compile time via source code transformation. Weaving of introductions is performed in a straightforward way, i.e. new members as well as additional `extends` and `implements` relationships are simply added. Hence, this will not be discussed in further detail.



```
class A {
  public void m(X x, Y y, Z z) {
    // do Somehing
  }
}
```

```
class MAdapter {
  pointcut pc(?jp) =
    methodDeclaration(?jp,?cID, m) &&
    classDeclaration(?cID,A);
  wrapM wrap<?jp,?arg> pc(?jp) {
    wrap<?jp>(?arg);
    System.out.println("after…."); 
  }
}
```

```
class A {
  public void m(X x, Y y, Z z) {
    aWrapperMethod(x, y, z);
  }
  public void aWrapperMethod(X _p1, Y _p2, Z _p3) {
    aWrappedMethod(_p1, _p2, _p3);
    System.out.println("after….");
  }
  public void aWrappedMethod(X x, Y y, Z z) {
    // doSomething
  }
}
```

Figure 3-19. Weaving a method wrapper `wrapM` to method `m` in class `A`.

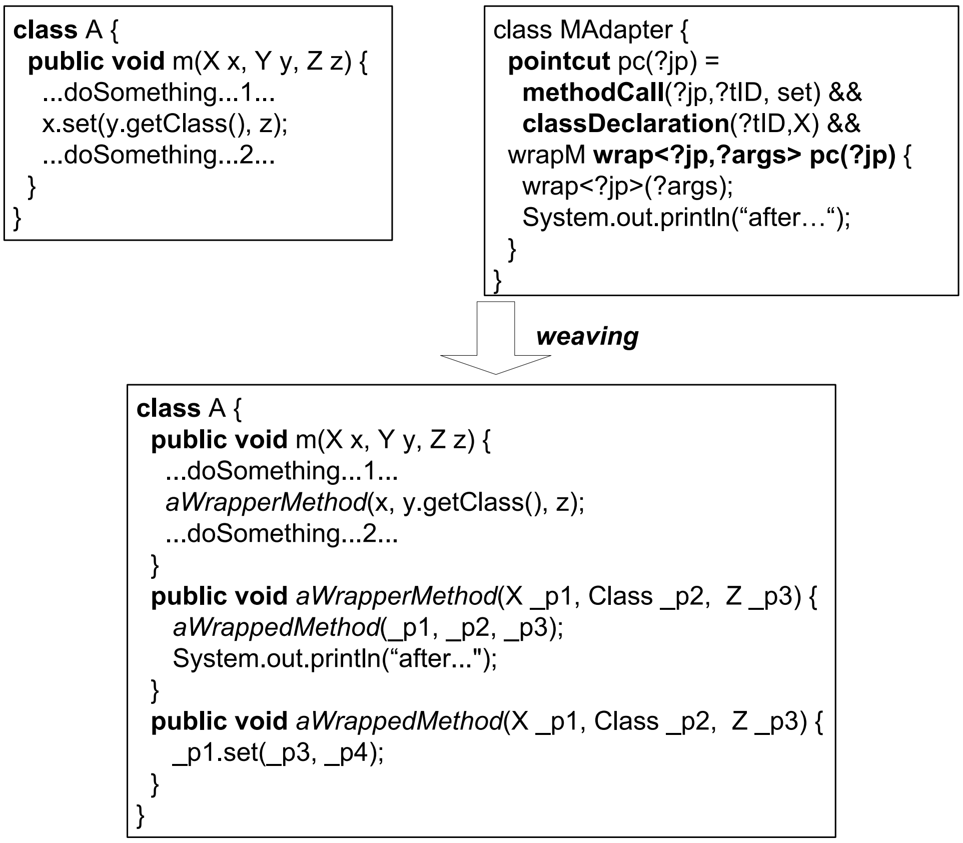
Weaving join point wrappers is slightly more complex. Here, weaving of the method definition wrappers and method call wrappers will be explained in more detail. Since field access as well as field assign wrappers can be explained in terms of call wrappers (since both wrappers simply differ with respect to the provided parameters) they will not be discussed here.

3.4.5.1 Method definition wrappers

Weaving a method definition wrapper results in the construction of two new methods in the same class that defines the corresponding method.

1. The original method becomes the **wrapped method** (with the same signature) by copying the whole method. The signature and the parameter types stay the same and the method name is specified by the system. The system chooses the name in a way that naming collisions are avoided. This prevents the new generated method from accidentially changing the behavior of the woven application.

2. The join point wrapper is translated into a **wrapping method** that contains the body of join point wrapper. The signature of the **wrapping method** corresponds to the signature of the **wrapped method**. For all parameters passed to the method wrapper, the system generates a new identifier. The identifiers are chosen in a way that they do not occur within the body of the wrapper in order to prevent the wrapper to accidentially changing them.
3. Inside the wrapping method all occurrences of logic variables are replaced.
4. The `wrap` statement (which corresponds to the proceed statement in AspectJ) is translated to a method call to the **wrapped method**.



```
class A {
  public void m(X x, Y y, Z z) {
    ...doSomething...1...
    x.set(y.getClass(), z);
    ...doSomething...2...
  }
}
```

```
class MAdapter {
  pointcut pc(?jp) =
    methodCall(?jp,?tID, set) &&
    classDeclaration(?tID,X) &&
  wrapM wrap<?jp,?args> pc(?jp) {
    wrap<?jp>(?args);
    System.out.println("after…");
  }
}
```

```
class A {
  public void m(X x, Y y, Z z) {
    ...doSomething...1...
    aWrapperMethod(x, y.getClass(), z);
    ...doSomething...2...
  }
  public void aWrapperMethod(X _p1, Class _p2,  Z _p3) {
    aWrappedMethod(_p1, _p2, _p3);
    System.out.println("after...");
  }
  public void aWrappedMethod(X _p1, Class _p2,  Z _p3) {
    _p1.set(_p3, _p4);
  }
}
```

Figure 3-20. Weaving a call wrapper `wrapM` to a `set` call in class `A`.

Figure 3-19 illustrates the weaving of wrapper `wrapM` to a method `m` in class `A`. The signatures of the resulting methods are the same (all containing the parameter types `X`, `Y` and `Z`). The wrapped method still contains the same code as it did before. The wrapping method invokes the wrapped method with the parameters that have been passed to itself (the parameters `_x`, `_y`, and `_z`).

3.4.5.2 Method Call Wrappers

Method call wrappers work similar to method wrappers. The only difference is that a method call has different parameters. A method call exposes the calling object, the called object, as well as the parameters of the method call.

1. The original method call is wrapped by a **wrapped call method**. The parameters consist of the called object and the parameters passed by the method call. In case the method call appears from within a static method or the called method is static, the corresponding parameters are not part of the signature.
2. The join point wrapper is translated into a **wrapping method** that contains the body of join point wrapper. The signature of the **wrapping method** corresponds to the signature of the **wrapped call method**. For all parameters passed to the method wrapper the system generates a new identifier. The identifiers are chosen in a way that they do not conflict with any other parameter.
3. Inside the wrapping method all occurrences of logic variables are replaced.
4. The `wrap` statement is translated into a method call to the **wrapped method**.

Figure 5-1 illustrates the weaving of wrapper `wrapM` to a method `m` in class `A`. Note that within the wrapped method the parameters passed to the original method call are simply variables: The corresponding expressions (in the example, the method call `getClass()` to the variable `y`) still reside in the original method.

3.5 Applying Parametric Introductions

The main motivation for providing the generic constructs in Sally was caused by the problem that introductions in AspectJ and Hyper/J do not permit to modularize a number of crosscutting concerns. Hence, this sections illustrates how the generic constructs are able to solve this problem.

Parametric introductions receive parameters during weave-time. The main motivation for this idea is the observation that sometimes static crosscutting code varies when it is woven to different types. The code that is to be introduced contains variables that are bound by the weaver.

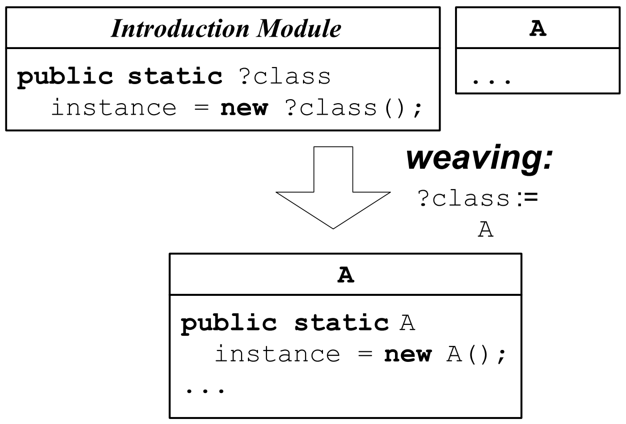


Figure 3-21. Parametric introductions.

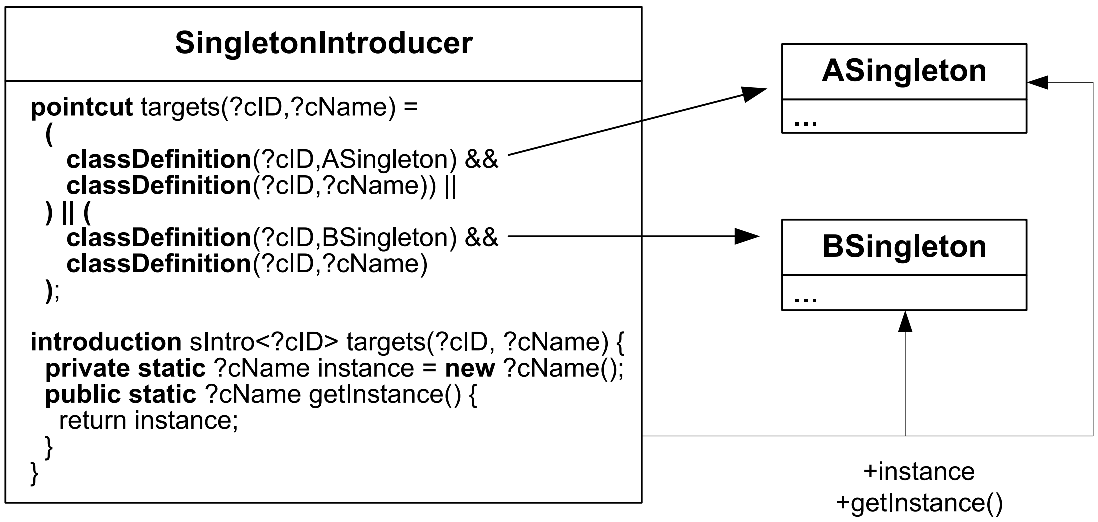
Figure 3-21 illustrates an example of a parametric introduction that is motivated by the singleton implementation from section 3.3.1. The introduction specifies a field `instance` to be introduced to a target class. The type of the field is a variable. Furthermore, the variable is initialized with a new object whose class is determined by the variable. The parameter used within that introduction is described by the identifier ?class. This introduction specifies that whenever the field is introduced to a class, the type of the field and the type of the class to which the new operator is applied are the same. In Figure 3-21, the weaver assigns the target class to the parameter ?class.

Parametric introductions require a mechanism to assign values to its parameters in order to enable the weaver to determine the actual woven code. Moreover, it is sometimes useful to be able to specify introductions without specifying the target classes. This permits to define libraries of introductions that can be adapted to target classes during application development without the need to perform invasive modifications in the introduction module. Furthermore, the motivating examples in the section 3.3 showed that it might be necessary to apply an introduction to target classes more than once with different parameter value pairs.

Parametric introductions are implemented in the aspect language Sally as explained in the previous sections. In the following, parametric introductions are applied in order to overcome the problems described in the beginning of this chapter. Thereto, different uses of parametric introductions are explained.

3.5.1 Concrete Parametric Introductions

Sally permits to use the parameters passed to the introduction within the introduction body. This means, for example, that types can be passed to an introduction where they can be used as parameters. A *concrete parametric introduction* is an introduction where the corresponding introduction and pointcut is non-abstract but where the code to be introduced contains variables passed from the pointcut specification.



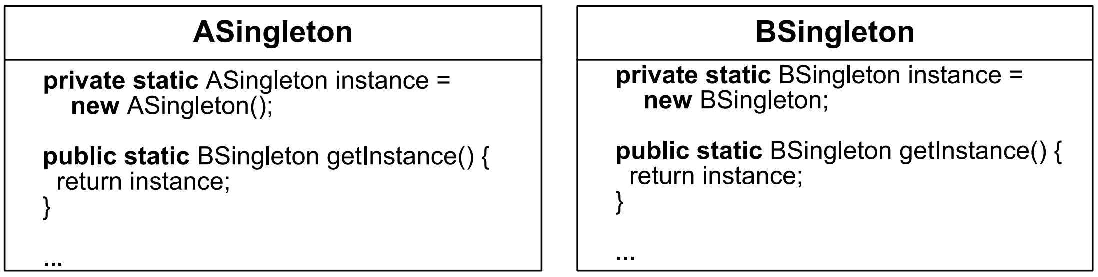
```
pointcut targets(?clD,?cName) =
  (
    classDefinition(?clD,ASingleton) &&
    classDefinition(?clD,?cName)) ||
  ) || (
    classDefinition(?clD,BSingleton) &&
    classDefinition(?clD,?cName)
  );

introduction sIntro<?clD> targets(?clD, ?cName) {
  private static ?cName instance = new ?cName();
  public static ?cName getInstance() {
    return instance;
  }
}
```

Figure 3-22. Singleton implementation in Sally (concrete parametric introduction).

Figure 3-22 illustrates a singleton implementation in Sally. Class SingletonIntroducer contains an introduction sIntro. That refers to the pointcut targets. The pointcut binds the identities of the class definition join points for class SingletonA and SingletonB to the variable ?cID and the class names to the variable ?cName. Consequently, the pointcut describes two tuples: One consisting of `ASingleton`'s identity and its name, and one consisting of `BSingleton`'s identity and its name.

The target of the introduction is the parameter `?cID`. Furthermore, the introduction has the parameter `?cName` from the corresponding pointcut that can be used within its body. The introduction body contains the known methods from the singleton implementation. The type of the instance variable and the return type of the introduced method are determined by the values of the parameter `?cName`.

When the application is compiled, the parameters are replaced by their values and the introductions are performed on the target classes. Consequently, the classes `ASingleton` and `BSingleton` receive the class variable `instance` and a corresponding class method `getInstance()`. In both cases the appropriate types are being introduced: `ASingleton` for the type of the class variable and the return type of the class method in the first case, `BSingleton` for the class variable and the return type of the class method in the second case.



```
private static ASingleton instance =
   new ASingleton();

public static BSingleton getInstance() {
  return instance;
}

...
```

```
private static BSingleton instance =
   new BSingleton;

public static BSingleton getInstance() {
  return instance;
}

...
```

Figure 3-23. Woven target classes.

Figure 3-23 illustrates the impact of the singleton introduction on the target classes `ASingleton` and `BSingleton`.

3.5.2 Abstract Parametric Introductions

The applicability of introductions can be increased substantially if it could be left open at introduction definition time to what classes the introductions are to be applied to. This permits to define libraries of introductions independently of their application. The main benefit is that the introduction can be applied in a different module than its definition. Thus, its application avoids the execution of destructive modifications in the introduction module. This permits a higher level of reusability of aspects.

In AspectJ, such an abstract introduction is achieved by the container introduction. In Hyper/J such a separation of introduction definition and application is much more natural since the members to be introduced are normal Java members while the introduction application is defined within a hypermodule.

Like AspectJ, Sally permits to define abstract pointcuts that can be overridden in a subclass. That means that the abstract pointcut in a superclass does not refer to any point in the program, while the overriding pointcut does. Introductions can be abstract by leaving the pointcut that the introduction refers to abstract. For the singleton example, such a specification of abstract parametric introductions is desirable: Since the singleton-specific elements are often required features of different classes in different projects, it is desirable to specify such introductions within one module without directly specifying the target classes.

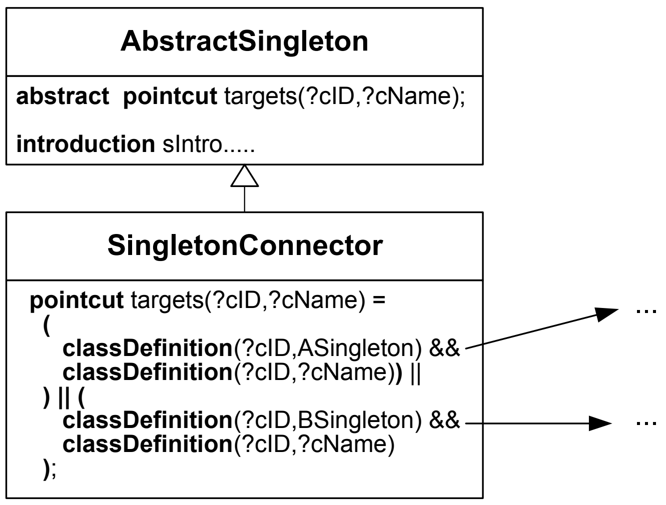

AbstractSingleton
abstract pointcut targets(?cID,?cName);
introduction sIntro.....
SingletonConnector
pointcut targets(?cID,?cName) =
(
classDefinition(?cID,ASingleton) &&
classDefinition(?cID,?cName)) ||
) || (
classDefinition(?cID,BSingleton) &&
classDefinition(?cID,?cName)
);
...
...

Figure 3-24. Abstract parametric introductions for singletons.

Figure 3-24 illustrates the application of an abstract introduction for the singleton example. Class `AbstractSingleton` is specified as in Figure 3-22 except that the pointcut is declared abstract. Hence, this class does not perform any introduction for a given application. Class `SingletonConnector` overrides the pointcut and binds the parameter `?cID` to the identities and `?cName` to `ASingleton` and `BSingleton`. By that, the introduction becomes concrete and is applied at compile-time. The benefit of this kind of introduction is that the module that defines the introduction (`SingletonIntroducer`) does not need to be modified if the introduction is applied to different classes. In order to apply the introduction to other classes than the ones mentioned here either `SingletonConnector` has to be modified or another subclass of `SingletonIntroducer` has to be created.

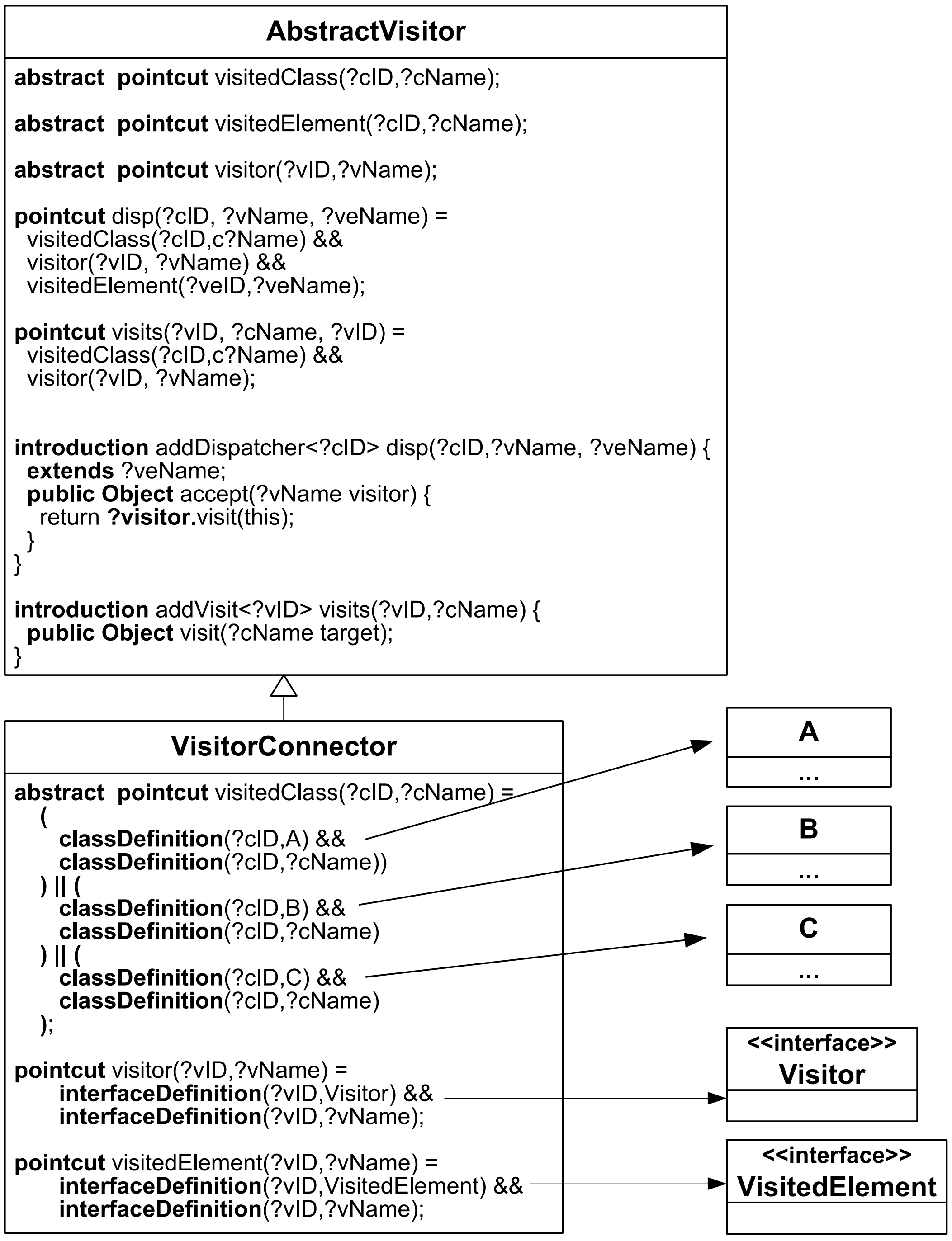
The impact of the abstract parametric introduction on the target classes is the same as described in the previous section.

3.5.3 Parametric Multi-Introductions

The visitor example motivated the necessity to apply an introduction more than once to a target class: The method `visit` needs to be introduced to the interface `Visitor`, however, each time with different parameter types. Furthermore, the double dispatch method needs to be introduced to all classes whose instances can be visited. In Sally introductions are woven as long as there exist different values for the introduction parameters.

Figure 3-25 illustrates the corresponding implementation of the visitor example in Sally. The class `AbstractVisitor` contains the introduction `addDispatcher` that introduces the double dispatch method as well as the interface `VisitedElement` to each class to be visited. Since the referring pointcut `visitedClass` is abstract, the classes need to be defined in a subclass which overrides this pointcut.

The introduction `addVisit` receives two parameters from its pointcut `visits`. The parameter `?vID` determines the target interface and `?cName` represents the name of each type that should be visited. Thus, `addVisit` makes use of a parametric introduction and introduces a method `visit` for each class to be visited to the visitor interface with the corresponding type. The pointcut `visit` binds the parameters `?vID` to the interface `Visitor` and `?cName` to each class defined in `visitedClass`. Since `visitedClass` is declared abstract, `target` does not bind any variables as long as `visitedClass` is not defined.



```
abstract  pointcut visitedClass(?cID,?cName);

abstract  pointcut visitedElement(?cID,?cName);

abstract  pointcut visitor(?vID,?vName);

pointcut disp(?cID, ?vName, ?veName) =
  visitedClass(?cID,c?Name) &&
  visitor(?vID, ?vName) &&
  visitedElement(?veID,?veName);

pointcut visits(?vID, ?cName, ?vID) =
  visitedClass(?cID,c?Name) &&
  visitor(?vID, ?vName);


introduction addDispatcher<?cID> disp(?cID,?vName, ?veName) {
  extends ?veName;
  public Object accept(?vName visitor) {
    return ?visitor.visit(this);
  }
}

introduction addVisit<?vID> visits(?vID,?cName) {
  public Object visit(?cName target);
}
```

```
abstract  pointcut visitedClass(?cID,?cName) =
    (
      classDefinition(?cID,A) &&
      classDefinition(?cID,?cName))
    ) || (
      classDefinition(?cID,B) &&
      classDefinition(?cID,?cName)
    ) || (
      classDefinition(?cID,C) &&
      classDefinition(?cID,?cName)
    );

pointcut visitor(?vID,?vName) =
      interfaceDefinition(?vID,Visitor) &&
      interfaceDefinition(?vID,?vName);

pointcut visitedElement(?vID,?vName) =
      interfaceDefinition(?vID,VisitedElement) &&
      interfaceDefinition(?vID,?vName);
```

Figure 3-25. Visitor implementation in Sally (multi-introduction).

Class VisitorConnector extends VisitorIntroducer, overrides the abstract pointcut visitedClass, and binds the parameters ?cID and ?cName to identities and class names of the classes A, B, and C. Furthermore, the abstract pointcut visitedElement refers to the interface VisitedElement that represents the common type for all classes whose objects can be visited. The pointcut visitor defines the target visitor interface that should be used. Hence, from the connector's point of view, all pointcuts are concrete and bind variables, and thus the aspect can be defined concrete.

Consequently, the introduction addDispatcher is applied to the target classes A, B, and C and the method accept is added to each of these classes. The introduction addVisit uses the first parameter as the target class. Variables of disp belong to the same target class (the interface Visitor), but differ in the value of their second parameter and third parameter. Hence, the introduction is applied three times to Visitor, always with different parameters. Because ?vName is used as a parameter within the introduction, VisitorConnector adds three additional visit methods to the visitor interface that differ in their parameter types.

3.5.4 Unnamed Introductions

Section 3.3.3 (implementation of the decorator pattern) motivated why methods need to be introduced to interfaces whose signatures are not known at introduction definition time. The problem in the context of the decoration implementation is twofold: First, at introduction definition time it is unknown how many methods are to be introduced, and second, the signatures of the methods to be introduced are unknown. The first problem can be handled by the previously introduced introductions. The latter is solved by introducing unnamed methods. These are method introductions whose signatures are determined at weave-time by corresponding queries on the base application.

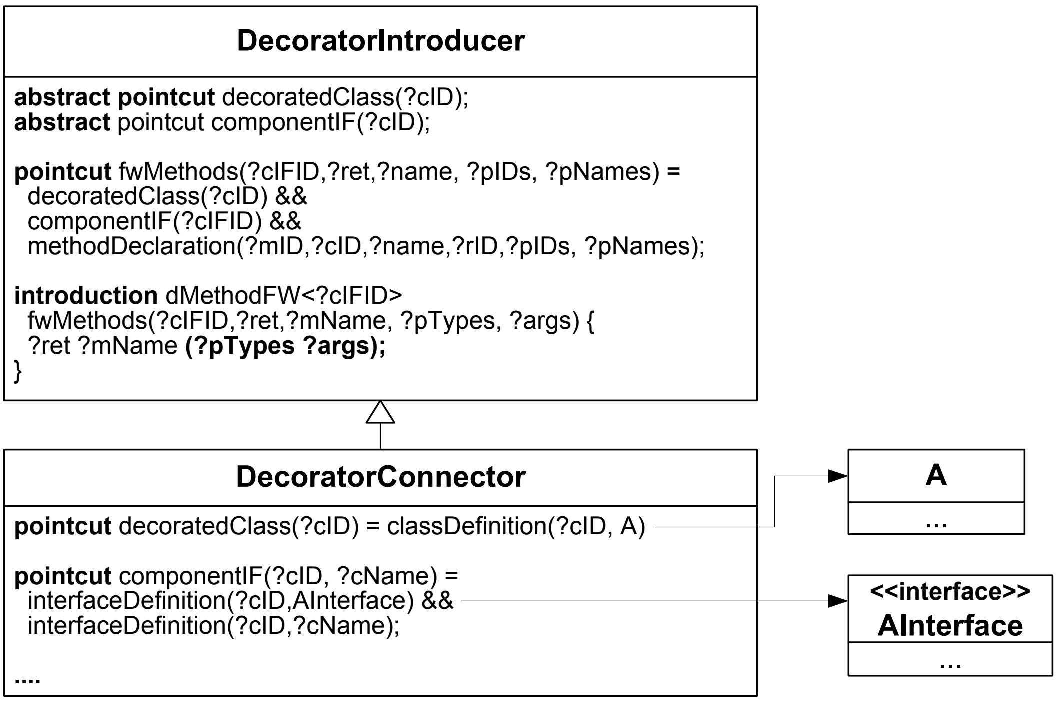
Figure 3-26 schematically illustrates a decorator implementation in Sally. For reasons of simplicity the Figure only concentrates on the introduction of unnamed methods to the component interface and ignores the rest of the implementation.

DecoratorIntroducer contains two abstract pointcuts to declare the class to be decorated as well as the component interface. The concrete pointcut fwMethods determines all methods included in the decorated class. The introduction dMethodFW uses the passed parameters to create the method signature that will be introduced to the component interface.

Special attention has to be paid to the handling of the parameters of the method: Only two introduction parameters are used here (?pTypes and ?args). Sally generates a corresponding list from the parameters from both lists. Consequently, a list of parameters is generated that has the same signature as the corresponding method. In this concrete example, Sally tests if the lists bound to ?pTypes and ?args have the same length and introduces a list of parameter type and value pairs.

The concrete DecoratorConnector overrides decoratedClass and componentIF. Hence, the introduction cMethodIntro can be actually applied.

Consequently, by providing parametric introductions that permit to use weave-time parameters it is possible to handle all problems that resulted from the restricted ability to specify introductions in AspectJ and Hyper/J.



```
abstract pointcut decoratedClass(?cID);
abstract pointcut componentIF(?cID);

pointcut fwMethods(?clFID,?ret,?name, ?pIDs, ?pNames) =
  decoratedClass(?cID) &&
  componentIF(?clFID) &&
  methodDeclaration(?mID,?cID,?name,?rID,?pIDs, ?pNames);

introduction dMethodFW<?clFID>
  fwMethods(?clFID,?ret,?mName, ?pTypes, ?args) {
  ?ret ?mName (?pTypes ?args);
}
```

```
pointcut decoratedClass(?cID) = classDefinition(?cID, A)

pointcut componentIF(?cID, ?cName) =
  interfaceDefinition(?cID,AInterface) &&
  interfaceDefinition(?cID,?cName);

....
```

Figure 3-26. Extract of the Decorator implementation in Sally (unnamed introduction).

3.6 Related Work

So far parametric introductions were directly compared with the corresponding mechanisms in AspectJ and Hyper/J. However, other concepts and mechanisms were introduced that are usually not referred to in the context of aspect-oriented programming but are quite similar to aspect-oriented introductions.

3.6.4.1 Generic Types

Parametric introductions as implemented in Sally on top of the programming language Java are mainly used to pass types as parameters to introductions. Thus, they look quite similar to *generic types* [BOSW98] or *parametric types* [MBL97] in Java that also permit types to be passed as parameters. The most obvious difference between both approaches is that an introduction has a direct impact on the target class, i.e., it directly extends the existing target class with additional members while generic types are new types that need to be instantiated and do not influence the existing type structure. Thus, generic types are preplanned while parametric introduction permit an unanticipated evolution.

3.6.4.2 Roles

Roles [Pern90] are temporary views on objects[30]. A role's properties can be regarded as subjective and extrinsic properties of the object the role is assigned to. During its

[30] The relationship between roles and aspect-oriented systems in general will be explained in more detail in Chapter 6, section 6.8.

lifetime an object is able to adopt and abandon roles. Thus, an environment of an object can access not only its intrinsic, but also its extrinsic properties. Because of this characteristic, roles provide a mechanism that can be compared to introductions (see [HaUn02b] for a comprehensive discussion of aspects and roles). There are numerous different implementations of the role concept that make use of the composition mechanisms of the underlying programming language. For example, [Knie96] proposes an implementation based on object-based inheritance, [GSR96] uses the Smalltalk-specific handling of incoming messages, [NeZd99] a language mechanism called *per-object mixins* and [HaUn02b] *dynamic proxies*. The major difference between roles and aspect-oriented introductions is that a role works on a single object, i.e., a role does not extend the interface of a class, but the interface of the object they are assigned to. As an exception, [NeZd99] also provides a mechanism called *per-class mixin* that permits to add roles to classes. Thus, the interface of the class is extended. Nevertheless, this mechanism does not permit to declare any variability within the role that is "vitalized" when the role is assigned to the target class. Furthermore, since this concept is provided by an untyped programming language it is hard to compare it to aspect-oriented introductions based on typed programming languages.

3.6.4.3 Aspect-oriented logic meta programming

[DVDH99] proposes *aspect-oriented logical meta programming* as a mechanism for modularizing concerns. Aspect-oriented logic meta programming means to write logical programs that reason about aspect declarations. Aspect declarations can be accessed and declared by logical rules. So, the weaver is constructed in a logical programming language that provides a number of rules for generating the woven code. The logical programming language proposed in [DVDH99] for weaving is called *TyRuBa* (in fact, the here proposed mechanism is implemented using TyRuBa, too).

In TyRuBa, *quoted code blocks* can be declared that permit to use pieces of Java code as terms in logical programs. These code pieces may contain logical variables that are substituted during weaving. In fact, this mechanism, in conjunction with the weaver, provides parametric introductions. The difference between both approaches is that the proposed implementation in Sally is an extension of the programming language Java while TyRuBa handles logical programming separately from object-oriented programming. The weaver implementation in TyRuBa is not connected to the object-oriented programming language but generates object-oriented code. That means, before weaving, no checks are performed, neither on the involved classes nor on the involved introductions. So it is not even determined if quoted code blocks contain Java code at all. This makes software development in TyRuBa error prone. In Sally all classes and introductions are parsed before weaving and type-checking is performed on the involved classes[31].

3.7 Chapter Summary and Conclusion

This chapter identified introductions as an important mechanism to modularize static crosscutting code in aspect-oriented programming languages. It showed common

[31] Of course, with the exception of parameterized introductions.

examples of static crosscutting code in which the introductions of AspectJ and Hyper/J fail to modularize those examples.

As a solution, parametric introductions have been proposed, i.e., introductions that receive parameters during weave-time. This chapter gave a detailed overview of Sally's language features (especially its pointcut language, wrappers, and introductions), which permits to specify generic aspects via logic variables that can be used within an aspect's body. This chapter demonstrated how parametric introductions (generic aspects that make use of logic variables within the introduction body) solve the inadequacies of Hyper/J and AspectJ by applying parametric introductions to the examples where the introductions of AspectJ and Hyper/J failed.

Parametric introductions are a powerful mechanism that increases the modularization of crosscutting code. The proposed usage of connecting introductions and the pointcut language in conjunction with an inheritance relationship between classes containing introductions increases the reusability of introductions.

Since the work on Sally has been published at the 2nd international conference on aspect-oriented software development (cf. [HaUn03a]), it is valid to call it an aspect-oriented approach. However, the question what key characteristics aspect-oriented systems have still remains. Furthermore, the underlying design issues of Sally (a quite different kind of pointcut language) seem to differ noteworthy from the systems like AspectJ, Hyper/J, and AspectS. Consequently, it is desirable to find abstract descriptions of aspect-oriented systems that also permit to describe the characteristics of Sally in an abstract way.

A first step toward this direction is already done throughout this chapter: The identification of introductions as one possible way of adapting class definition join points – an observation that is essential for identification of **constructive adaptations** in Chapter 5 (which will be introduced in section 5.5.2). Furthermore, the capability to parameterize an introduction or a wrapper is identified in Chapter 5 as a special design dimensions - the **parameterization of join point adaptations** (see section 5.5.3).

4

MORPHING ASPECTS – CONTINUOUS WEAVING FOR ASPECT-ORIENTED SYSTEMS

4.1 Introduction

The mechanism for integrating aspect modules with an application is called *weaving*. A weaver is responsible for adding all aspects to the application. In order to specify such integration, aspect-orientation makes use of a concept called *join point*. In [KHH+01], join points are introduced as *principled points in the execution of a program*. A typical example of a join point is a method call.

Conventionally, the developer starts the weaving process at a certain point in time and for a number of aspects to be integrated. Thereto, the weaver determines and adapts all locations in the base system that represent join points at runtime where potentially or for sure aspect-specific code needs to be executed. In [MKD03], locations in the code that represent join points during runtime are called **join point shadows**. In the following this chapter refers to shadows whose join points always lead to an execution of aspect-specific code as **unconditional join point shadows**, and those whose join points lead only under some circumstances to aspect-specific behavior as **conditional join point shadows**. For conditional shadows the weaver adds runtime checks determining whether or not aspect-specific code needs to be executed (this thesis refers to these runtime checks as *join point checks*). If such a check succeeds, the aspect-specific code (the *advice* code according to AspectJ terminology [KHH+01]) is executed.

One property of this conventional approach to weaving is that the set of join point shadows associated with a woven aspect remains the same for the aspect's lifetime. In the following, this thesis refers to this kind of weaving as **complete weaving**. Complete weaving is a process that determines and adapts all join point shadows including the creation of corresponding join point checks upfront and in advance. After weaving, all shadows in the application where aspect-specific code might be executed are adapted. Consequently, the set of join point shadows associated to an aspect is fix and does not change at runtime. Aspect-oriented systems like AspectJ, Hyper/J, and Sally [HaUn03a] that provide *pure static weaving*, i.e. weaving at compile time, necessarily need to perform a complete weaving since all join point shadows to be adapted have to be determined at a certain point in time (at compile time[32]).

In more complex applications complete weaving can lead to a huge number of adapted join point shadows whose join point checks fail and just produce runtime overhead. Especially shadows with join points that rarely trigger the execution of aspect-specific code in the execution of the program are useless time-consumers because most of the time the corresponding join point checks do not succeed and with that do not invoke an aspect's advice. In the worst case an aspect adapts a large number of join point shadows that never invoke an aspect's advice. In such cases the adapted shadows cause runtime overhead without ever accomplishing any benefit at all.

A large number of conditional shadows that rarely or never lead to an advice's execution are often not tolerable, especially not in performance critical parts of the system. Thus, it is desirable to reduce the number of conditional join point shadows as much as possible to reduce the number of failing and with that unnecessary join point checks.

In order to reduce the number of conditional join point shadows this chapter introduces the concept of *morphing aspects* which are incompletely woven aspects in combination with *continuous weaving*, an extension of *dynamic weaving* [Hirs02, PGA02].

The next sections, two typical examples of aspects based on complete weaving are provided that illustrate the necessity of handling the problem of conditional shadows whose join points rarely or never execute an aspect's advice. Section 4.3 introduces the concept of morphing aspects. There, dependency relationships among join points are discussed and the way they can be used to adapt join point shadows at a later point in time are described. Section 4.4 discusses implementation issues of morphing aspects by proposing an implementation in AspectS. An overview and a discussion of some experiments with morphing aspects are illustrated in section 4.5. After comparing morphing aspects to related work in section 4.6, section 4.7 summarizes and concludes this chapter.

4.2 Examples

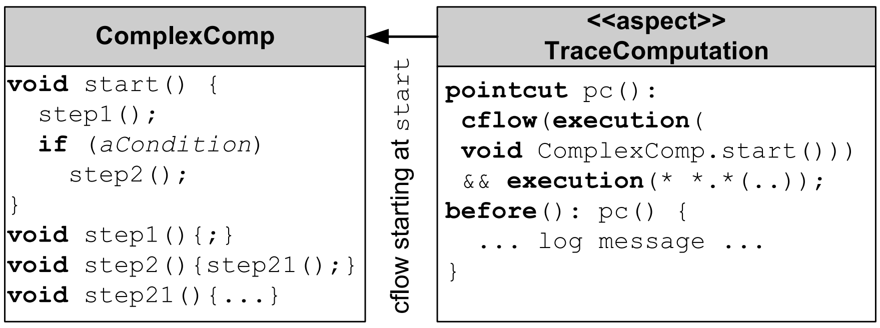
According to for example [Hirs02, Lope04, GyBr03, SHU02, VeHe03] *tracing* and *subject-observer* implementations are well-known and accepted candidates for discussions and illustrations of aspect-oriented programming. Because of its popularity this thesis uses AspectJ in those examples to better illustrate the problems associated with complete weaving[33].

[32] It should already be emphasized here that the term complete weaving is not equivalent to static weaving. A form of complete weaving also occurs in systems that provide dynamic weaving. This will be discussed in more detail in section 6.

[33] Please note that the intention here is neither to discuss AspectJ in detail nor to compare the here proposed approach with AspectJ. The intention here is to discuss the impact of complete weaving on the number of join point checks in the woven application.

4.2.1 Tracing

A woven tracing aspect captures messages sent to or from particular objects, e.g. to a log file. Usually, developers want to trace the control flow starting at a certain point in the execution of a program. For example, a developer wants to capture the behavior of critical modules in order to analyze their behavior either later or right at runtime.



```
void start() {
  step1();
  if (aCondition)
    step2();
}
void step1(){;}
void step2(){step21();}
void step21(){...}
```

```
pointcut pc():
 cflow(execution(
 void ComplexComp.start()))
 && execution(* *.*(..));
before(): pc() {
  ... log message ...
}
```

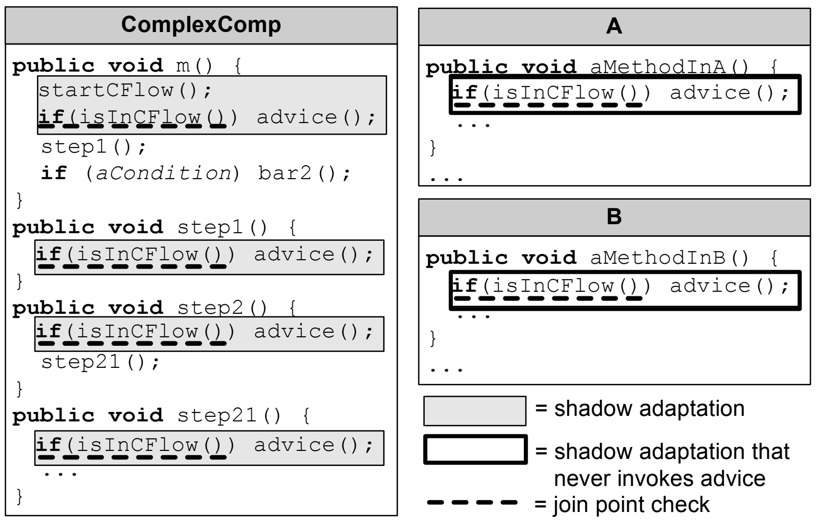
Figure 4-27. Tracing aspect in AspectJ logging methods in the control flow starting at method `start` in `ComplexComp`.

A typical approach to implementing tracing in AspectJ is to use the `cflow` pointcut designator [KHH+01]. Figure 4-27 shows the corresponding code in AspectJ where a tracing aspect `TraceComputation` logs all messages once the control flow passes the method `start` in class `ComplexComp` that starts a complex computation[34]. One advantage of this implementation is the declarative pointcut definition that describes all join points where the tracing aspect needs to execute some advice. Hence, developers do not need to examine the code on their own, i.e. they do not need to determine what methods are potentially called within the control flow starting from method `start` in class `ComplexComp`.

However, this implementation has some drawbacks due to complete weaving. In general, the exact computation of methods that are executed within a certain control flow is impossible. For example, it is hard to compute upfront whether the condition in method `start` will ever be satisfied and methods `step2,` and `step21` (and methods invoked by `step21`) will ever be invoked from the control flow passing `start`. To guarantee the correct behavior of the aspect the weaver must consider these methods in addition to methods `start` and `step1`, which will be definitively invoked in the control flow. For all these methods the weaver has to determine whether they can also be executed in control flows that do not pass method `start`. In such cases the weaver needs to decorate shadows with join point checks that check at runtime if the current method is part of the control flow to be traced or not. If a large number of different control flows in the application use methods of `ComplexComp` (other than `start`) the join point checks fail most of the time and only cause runtime overhead. If

[34] This use of the `cflow` construct for implementing tracing corresponds (with minor changes) to the implementation like for example proposed in [SLL03]. A similar use for a different purpose can be found for example in [CHJ03].

the condition in method `start` is *never* satisfied, the join point checks at method `step2` and `step21` only cause runtime overhead when they are invoked from different methods without ever executing the advice in `TraceComputation` at all. The problem becomes even bigger if `bar21` executes a large number of other methods. The corresponding shadows would also never invoke the advice in `TraceComputation`.



```
ComplexComp
public void m() {
   startCFlow();
   if(isInCFlow()) advice();
   step1();
   if (aCondition) bar2();
}
public void step1() {
   if(isInCFlow()) advice();
}
public void step2() {
   if(isInCFlow()) advice();
   step21();
}
public void step21() {
   if(isInCFlow()) advice();
   ...
}
```

```
A
public void aMethodInA() {
   if(isInCFlow()) advice();
   ...
}
...
```

```
B
public void aMethodInB() {
   if(isInCFlow()) advice();
   ...
}
...
```

Figure 4-28. An illustration of the woven tracing aspect including additional classes A and B.

In AspectJ the situation is somewhat different. AspectJ hardly analyzes control flows. For the example from Figure 4-27 AspectJ determines all methods matching the last part of the pointcut, i.e. (`execution(* *.*(..))`), and creates corresponding join point checks. Since this matches every existing method, AspectJ creates checks for every existing method in the entire system (except for those in system libraries). Figure 4-28 illustrates the code woven by AspectJ, not only for class `ComplexComputation` but also for two additional classes `A` and `B` also present at weave time. Before executing the original code of any method in the system, join point checks are performed. The benefit of this approach is that no cost-intensive computation is necessary which would slow down the compilation process. On the other hand the performance of the whole system decreases in the presence of the woven aspect. This implies that the performance decreases even in classes like `A` and `B` whose methods will never be executed in the control flow of interest. Performance measurements in [BHMO04] showed that a single woven `cflow` substantially decreases the overall performance of the system.

The overall problem in the tracing examples is that it is usually not fully computable at weave-time what methods are invoked within the control flow of interest. Consequently, a large number of conditional shadows exist in the system whose execution causes runtime overhead but rarely lead to an execution of the aspect specific code.

4.2.2 Subject-Observer Protocol

Perhaps the most frequently used example in the area of aspect-oriented programming is the implementation of the **observer design pattern** [GHJV95] as discussed for example in [GyBr03, SHU02, VeHe03]. The pattern permits a set of objects (called *observers*) to be attached and detached to and from other objects (called *subjects*) to become informed about their state changes.

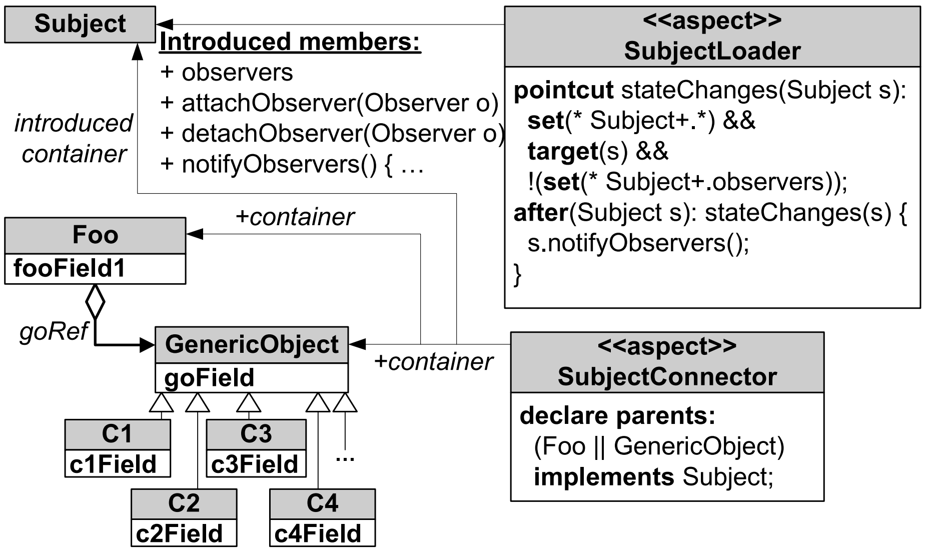
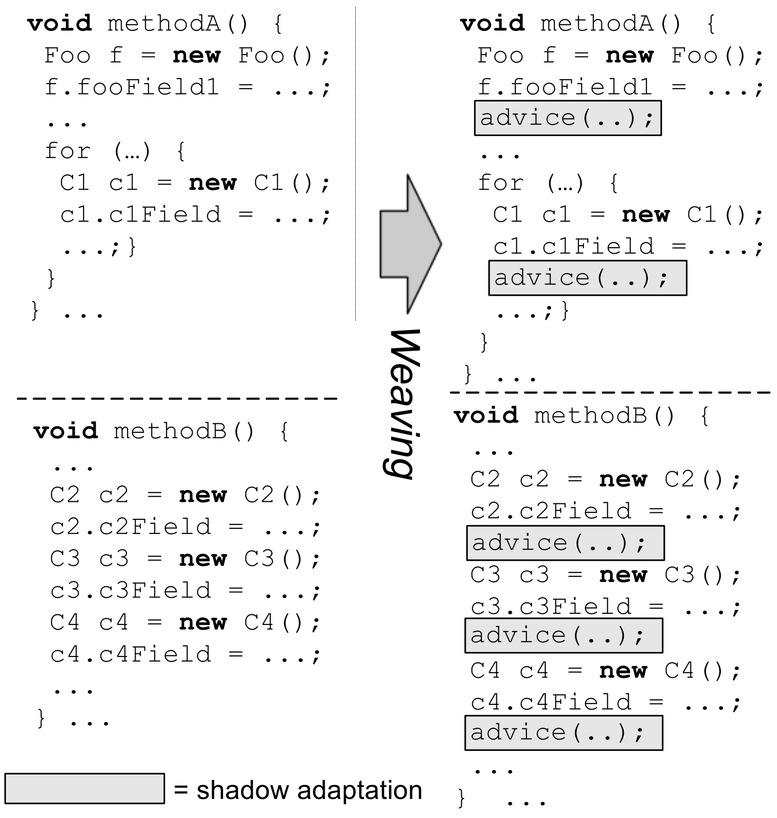


Figure 4-29. A subject-observer implementation in AspectJ.

Observers are interested in state changes, i.e. changes of fields associated with a subject. This includes fields that are directly part of the subject as well as fields of objects, which are directly or indirectly referenced by the subject (see for example [HHUK03, GyBr03], see also section 1.2.2.1 for a more detailed discussion of the observer pattern).

Figure 4-29 illustrates a typical implementation of the subject-observer protocol in AspectJ based on the **container introduction idiom** ([HSU03], see also section 3.2.1). `SubjectLoader` states that observers can be attached to and detached from instances of `Subject` by introducing appropriate fields and methods to `Subject`. The aspect's pointcut `stateChanges` and the corresponding advice define that an assignment to any field declared in `Subject` (i.e. an assignment to a field declared in a class implementing `Subject`) yield the notification of observers. The pointcut language of AspectJ does not permit to declare the state change of every referenced object for a given subject (cf. [HHUK03,GyBr03] for further discussion). So, developers have to enumerate explicitly every class whose objects should inform observers about state changes (in `SubjectConnector`). In order to permit the observation of `Foo` instances and its referenced objects of type `GenericObject`, the developer connects `Subject` to both classes (Figure 4-29).

Again, the implementation suffers from some drawbacks resulting from complete weaving. In general, it is not completely possible to determine what instances are ever referenced by subjects. For example, it is usually not computable if instances of class `C1`, `C2`, etc. are ever referenced by an instance of `Foo` at runtime. Consequently, join point checks need to be created that check at runtime at every field assignment, whether the current object is referenced by an instance of Foo. These checks become problematic if a class is frequently used in the application, whose instances are in fact never referenced by a Foo at runtime.



```
void methodA() {
 Foo f = new Foo();
 f.fooField1 = ...;
 ...
 for (…) {
  C1 c1 = new C1();
  c1.c1Field = ...;
  ...;}
 }
} ...
-----------------
void methodB() {
 ...
 C2 c2 = new C2();
 c2.c2Field = ...;
 C3 c3 = new C3();
 c3.c3Field = ...;
 C4 c4 = new C4();
 c4.c4Field = ...;
 ...
} ...
```

```
void methodA() {
 Foo f = new Foo();
 f.fooField1 = ...;
 advice(..);
 ...
 for (…) {
  C1 c1 = new C1();
  c1.c1Field = ...;
  advice(..);
  ...;}
 }
} ...
-----------------
void methodB() {
 ...
 C2 c2 = new C2();
 c2.c2Field = ...;
 advice(..);
 C3 c3 = new C3();
 c3.c3Field = ...;
 advice(..);
 C4 c4 = new C4();
 c4.c4Field = ...;
 advice(..);
 ...
} ...
```

Figure 4-30. Application using observed classes.

In AspectJ the problem is slightly different. Since AspectJ's pointcut language does not permit to specify classes whose objects are referenced by Foo, developers need to add the subject functionality to each class manually (compare to Figure 4-29). As a result, advice activations are inserted for each state change of instances of GenericObject as well as for its subclasses[35]. Figure 4-30 illustrates an application with a woven subject-observer aspect. Advice activations are created for each assignment of fields declared in GenericObject and its subclasses, even in those cases where the instances are not referenced by an instance of Foo. None of the objects on the right hand side are referenced by a Foo instance since the objects are newly created. Hence, advice execution is futile. If subclasses of GenericObject are frequently used in an application the performance decreases perceivably as every single assignment leads to the execution of the corresponding advice. Typical examples of

[35] Due to limitations of its pointcut language, AspectJ's shadows are unconditional. However, the shadows have to be conditional logically because it must be checked whether an object is referenced by a subject or not.

such-often used classes are collection classes or classes that serve as root classes in large frameworks.

The overall problem in the subject-observer examples is that the set of classes whose instances are referenced by subjects is usually not computable upfront. Hence, a complete weaver adapts shadows for field assignments of all classes whose instances are potentially referenced by a subject. If such classes are frequently used in the application while their instances are never referenced by a subject, the shadows just cause a runtime overhead. In the worst case, there is no observed object in the runtime system at all, yet still a large number of failing join point checks are executed.

4.3 Morphing Aspects and Continous Weaving

Morphing aspects are a new approach to reduce the number of join point checks by reducing the number of adapted shadows. In contrast to the conventional way of complete weaving used by known AO systems, morphing aspects are *incompletely woven* aspects. Morphing aspects are not entirely woven to an application by a weaving process that begins and ends at a certain point in time, computing and adapting shadows whose join points possibly execute aspect-specific code. Instead, the necessary shadows to be adapted are continuously computed and adapted (or released) by the aspects itself at well-defined points in the execution of the program, i.e. at certain join points. When a morphing aspect is woven it starts with a small set of initial join point shadows and dynamically adapts or releases shadows just when they are needed. Hence, the number of shadows associated with a morphing aspect changes during the aspect's lifetime. This thesis calls this process of computation, adaptation, and release of an aspect's shadows *morphing*. This thesis refers to the whole weaving process, i.e. initial weaving of morphing aspects, the morphing during their lifetime, and unweaving as *continuous weaving*.

Figure 4-31 illustrates a morphing aspect and its set of join point shadows at runtime. The ovals represent all join point shadows that are potentially associated with an aspect during its lifetime, as they would have been computed during complete weaving. The ovals within the aspect's border represent shadows adapted for the aspect. As long as the aspect is not woven, there are no shadows adapted for the aspect (Figure 4-31a). Initially, when the developer weaves the morphing aspect, a relatively small number of join point shadows is adapted by the aspects (Figure 4-31b). The set of actually adapted shadows changes during the aspect's lifetime. At a later point in time (Figure 4-31c), the aspect has nine more shadows in addition to the original join points. Even later (Figure 4-31d), five more shadows were adapted and most of the previous ones were released. In contrast to this, a completely woven aspect adapts all shadows that are potentially associated with an aspect (and creates corresponding join point checks), i.e. all ovals in Figure 4-31 right from the beginning. Morphing aspects adapt fewer shadows in the system. Hence, morphing aspects cause less runtime overhead due to failing join point checks as there are fewer join point checks in the system.

As a key characteristic of morphing aspects, they themselves determine at runtime at what points in the execution of the program the adaptation or release of join point shadows is necessary. Hence, join points in morphing aspects serve two different purposes. On the one hand the aspect's functionality (like logging, or notification of observers) is invoked, on the other hand the morphing process is started whenever particular join points are reached.

For the specification (and implementation) of morphing aspects and the corresponding morphing processes, developers are confronted with the following questions: What are the join points the aspect gets initially woven to, i.e. what shadows need to be adapted initially? When does the morphing process need to be carried out? How should the new set of join point shadows be determined?

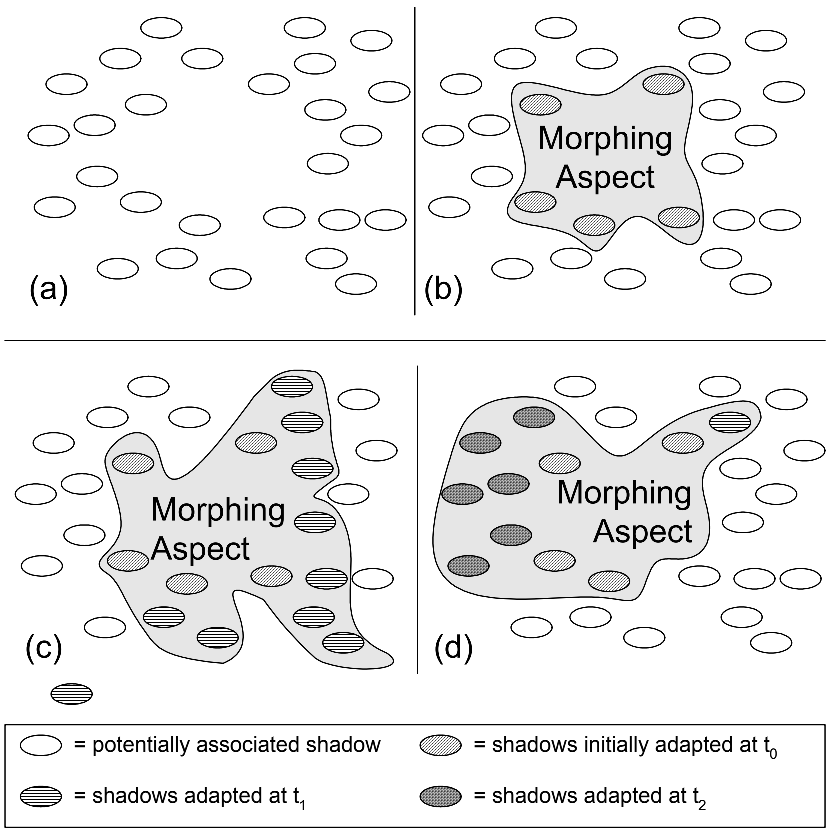

Morphing Aspect
Morphing Aspect
Morphing Aspect
(a)
(b)
(c)
(d)
= potentially associated shadow
= shadows initially adapted at t_0
= shadows adapted at t_1
= shadows adapted at t_2

Figure 4-31. A morphing aspect that changes its set of associated join point shadows during runtime.

In the following section dependencies among join points and shadows are discussed. Those dependencies determine a minimal set of shadows that need to be initially adapted. Furthermore, these dependencies determine what join point shadows can be adapted at some later point in time. Afterwards, it is described how these properties can be utilized to specify the morphing process.

4.3.1 Join Point Dependencies

In order to determine when new shadows need to be adapted or can be released, developers of a morphing aspect have to analyze how those join points (and their shadows) which are relevant for the aspect to be specified depend on each other. Dependencies among join points describe that a certain join point associated to an aspect (the *dependent* join point) can only be reached if another join point associated with the same aspect has been reached before. This thesis calls the corresponding shadows *dependent shadows*. All join points that do not depend on any other join point are *independent* (and are represented by independent shadows). On the technical level a dependency between join points expresses that join point checks of dependent shadows fail as long as the join points they depend on have not been reached before. Consequently, shadows for dependent join points do not need to be adapted as long as the join points they depend on have not been reached yet. This allows the adaptation of dependent shadows to be shifted to a later point in time. This situation is different for independent shadows. Their join points potentially occur in the execution of a program independently of any other join point associated to the same aspect and the adaptation of their shadows cannot be shifted to a later point in time. Hence, when the morphing aspect is initially woven at least all independent join point shadows need to be adapted.

In the following, the dependencies among join point shadows for the examples presented in section 4.2.1 and 4.2.2 are illustrated. A potential join point shadow is illustrated by an oval whereby an oval's label describes the shadow. A directed edge from a shadow `A` to a shadow `B` represents a dependency relationship which expresses that the join point represented by shadow `A` depends on the join point represented by shadow `B.` This also implies that shadow A depends on shadow `B`. The edge's label describes the kind of dependency. A shadow without any outgoing edge is an independent shadow, while a shadow with at least one outgoing edge is a dependent shadow.

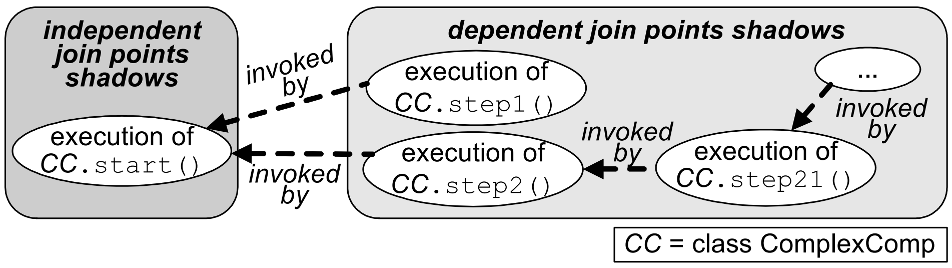


Figure 4-32. Dependencies in the tracing example.

Figure 4-32 illustrates the dependencies among join points shadows for the tracing example introduced in section 4.2.1. The execution of method `step1` or the execution of `step2` only needs to lead to an execution of aspect-specific code, if method `start` in class `ComplexComp` is executed and `start` invokes either `step1` or `step2`: The join point checks for the shadows at `step1` and `step2` fail as long as method `start` has not been executed and has not been invoked `step1` or `step2`. Hence, both shadows directly depend on the shadow for method `start`. For the same reasons the shadow at `step21` depends directly on the shadow at `step2` and the shadows for all methods that are eventually invoked by `step21` depend on the shadow at `step21`. In the tracing example, all join point shadows either directly or indirectly depend on the shadow representing the join point for the execution of method `start`. The shadow at `start` does not depend on any other shadow, i.e. this is an *independent shadow*. On a more abstract level, shadows of all methods that are either directly or indirectly invoked by `start` depend on the shadow for `start`.

The dependencies of join points in the subject-observer aspect are slightly more complex (Figure 4-33). The join points (and their shadows) to be handled by the aspect are the state changes of subjects (instances of `Foo`) and their referenced objects. So, all assignments to fields declared in `Foo`, `GenericObject`, and subclasses of `GenericObject` are join point shadows, which are potentially associated with the aspect. Those assignments execute aspect-specific code only if there is at least one object observing a `Foo` instance. This in turn depends on invocations of method `attach` which registers observers. Hence, all shadows for `fooField` and `goRef` assignments depend on the shadow at method `attach`[36]. The same is true for assignments to fields declared in `GenericObject` and its subclasses. However, their dependency is more complex. First, when an observer is attached, those shadows need to be adapted only for classes whose instances are referenced by a `Foo`. For example, as long as no `GenericObject` instance is referenced by a `Foo` instance no `goField` assignments need to be adapted. Second, assignments to `goField` depend on the `goRef` assignment since an instance of `GenericObject` becomes referenced by an instance of `Foo` by assigning it to `goRef`. For the same reason, all further assignments to fields declared in subclasses of `GenericObject` depend on the `goRef` assignment.

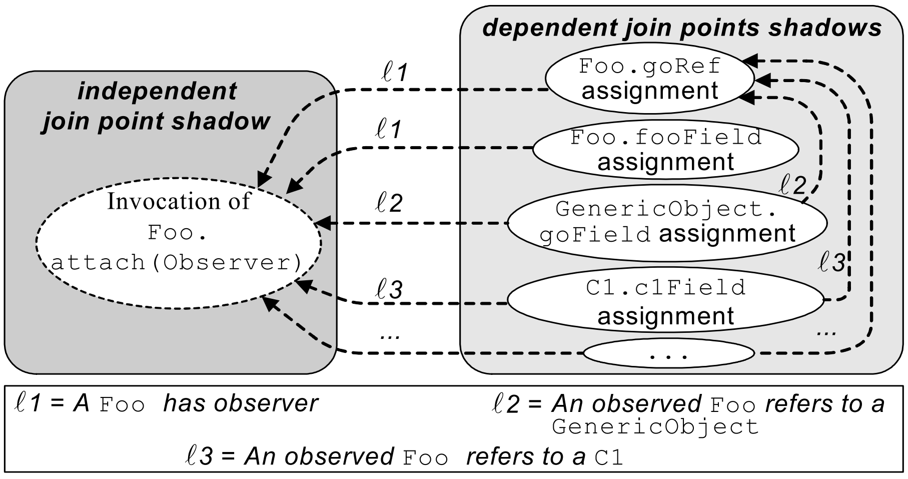

dependent join points shadows
independent join point shadow
Invocation of Foo. attach(Observer)
Foo.goRef assignment
Foo.fooField assignment
GenericObject. goField assignment
C1.c1Field assignment
. . .
ℓ1
ℓ1
ℓ2
ℓ3
...
ℓ2
ℓ3
...
ℓ1 = A Foo has observer
ℓ2 = An observed Foo refers to a GenericObject
ℓ3 = An observed Foo refers to a C1

Figure 4-33. Dependencies in the subject-observer example.

As exemplified in Figure 4-33 the only independent join point is the one for the invocation of method `attach`. In the subject-observer example the different natures of join points associated to the subject-observer aspect becomes manifest: The field assignments depend on a join point which does not lead to an execution of the aspect-specific code because field assignments depend on the execution of method `attach`. A field assignment join point informs the observers about a state change, while the join point at method `attach` does not. In order to emphasize that fact, the `attach` shadow is rendered using a different style (Figure 4-33).

[36] For the same reason, field assignments depend on method `detach` because after invoking `detach` no object observes `Foo` anymore and no aspect-specific code need to be executed. For reasons of simplicity this dependency is omitted in Figure 4-33.

4.3.2 Specifying the Morphing Process

Once the dependencies are determined, developers have to decide how to utilize them for the specification of a morphing aspect's morphing process. First, developers have to specify what shadows that involve join point checks are to be initially handled. Next, developers have to specify what initial join points start the morphing process. Then, developers have to define the morphing process itself.

At least all independent join point shadows have to be initially adapted because they do not depend on any other shadows. Hence, it is not possible to utilize their dependencies for a late shadow adaptation. Additionally, the developer can decide to adapt some additional dependent shadows at initial weave time: In case the dependent join points are reached very often, the developer may not want to adapt their shadows during the weaving process but right from the beginning.

The morphing process consists of the following parts. First, the process has to determine a number of dependent shadows to be adapted (or to be released). For that purpose, the morphing process can make use of *reflection* [Maes87]: The process reflects on the join points starting the morphing process and computes the dependent shadows. Second, the morphing process specifies the join point checks for all shadows to be created. Third, it has to be determined for each newly adapted shadow whether it's join points invoke aspect specific code and/or the morphing process.

In the following two reasonable specifications of the morphing process for the tracing and subject-observer examples are illustrated based on the discussion of join point dependencies from section4.3.1. The morphing processes proposed here are kept as simple as possible to illustrate how to utilize join point dependencies.

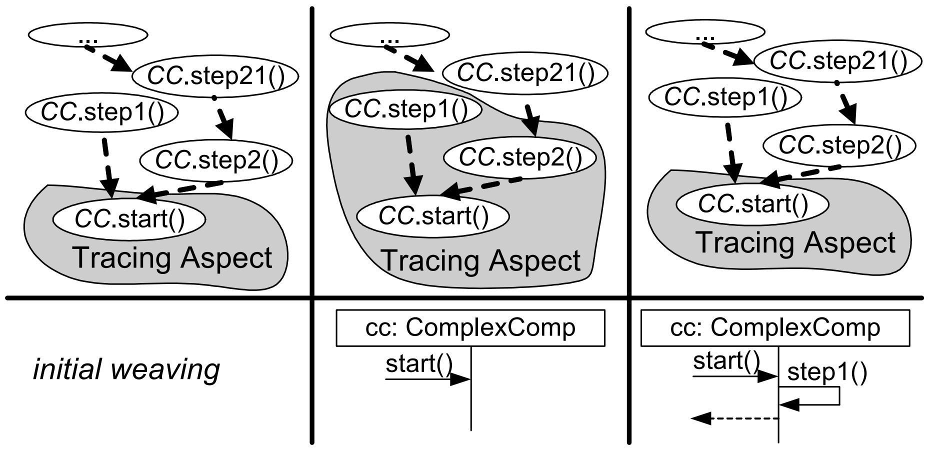


Figure 4-34. Morphing tracing aspect after initial weaving, after execution of `start`, and after execution of `step1`.

For the tracing aspect (Figure 4-34) at least one (independent and unconditional) shadow at method `start` in `ComplexComp` has to be initially adapted. For reasons of simplicity, it was decided to adapt only this shadow to keep the number of join point checks low and to adapt all dependent shadows as late as possible. Hence, Figure 4-34 illustrates that only one shadow is initially adapted for the tracing aspect.

Once a join point of this shadow is reached, the tracing code is executed and the morphing process starts. The morphing process determines all methods that potentially are invoked by the method enabling the process. The shadows for all these methods are adapted. The corresponding join point checks examine if the method is invoked within the control flow being traced[37]. If the join point check succeeds, the tracing code is executed and the morphing process starts once again. Whenever a method within the control flow is no longer executed, all dependent shadows are released. So, if a `ComplexComputation` receives a message `start` (and the message is logged), the morphing process computes all methods that are potentially invoked by `start` and adapts the corresponding shadows (`step1` and `step2`) (in the middle of Figure 4-34). When `step1` is invoked by `start` the method is logged and the morphing process starts once again. Since no other methods are potentially invoked by method `step1` no further shadows are created. If `step2` is not invoked and `start` is no longer executed the shadows at `step1` and `step2` are released (right hand side of Figure 4-34). So, as long as the condition in method `start` does not lead to an execution of `step2`, no shadows for `step21` (and methods invoked by `step21`) are created.

For the subject-observer aspect at least one unconditional shadow for method `attach` needs to be initially adapted[38] (see left hand side of Figure 4-35). Similar to the previous example it was simply decided to adapt only this shadow to keep the number of adapted join point shadows low (and to simplify the morphing process). An invocation of the method `attach` in class `Foo` starts the morphing process.

A simple morphing process for this aspect works as follows. First, the process adapts shadows for all assignments to the fields `fooField` and `goRef` whose execution leads to notifications of observers. Second, the process reflects on the `Foo` instance whose join point started the morphing process. It determines the referenced object and adapts the field assignment shadows for notifying observers. And finally, the morphing aspect adapts the `goRef` join point to start the morphing process.

Figure 4-35 illustrates the above-described morphing process. After an observer is attached the morphing process adapts shadows for all assignments to fields declared in `Foo`. Furthermore, the morphing process determines the object referenced by `goField` of object `foo`. Since in Figure 4-35 `foo` does not refer to any `GenericObject`, no further shadows are adapted. When an instance of `C1` is assigned to `foo`, the morphing process starts once more (because assignments of `goRef` start the morphing process). The process determines the fields of the assigned object. Since `c1` is an instance of `C1`, shadows are adapted for all assignments to `goField` (declared in `GenericObject`) and `c1Field` (declared in `C1`).

37 There are different ways to implement such a condition. In AspectJ the current thread is stored when the control flow starts, and each join point check determines whether the current thread is stored. Languages like for example Smalltalk permit to analyze the call stack to determine whether the current method occurs in the control flow of interest.

38 Like in the previous section this thesis skips for reasons of simplicity the discussion about method `detach` here which is also an independent shadow.

4.4 Implementation Example

In this section an exemplary implementation of a morphing aspect in AspectS [Hirs02] is introduced. AspectS is an aspect-oriented system providing dynamic weaving in the Smalltalk dialect Squeak. This section concentrates here only on the implementation of the tracing aspect. See [HHUK03] for a morphing implementation of the subject-observer implementation.

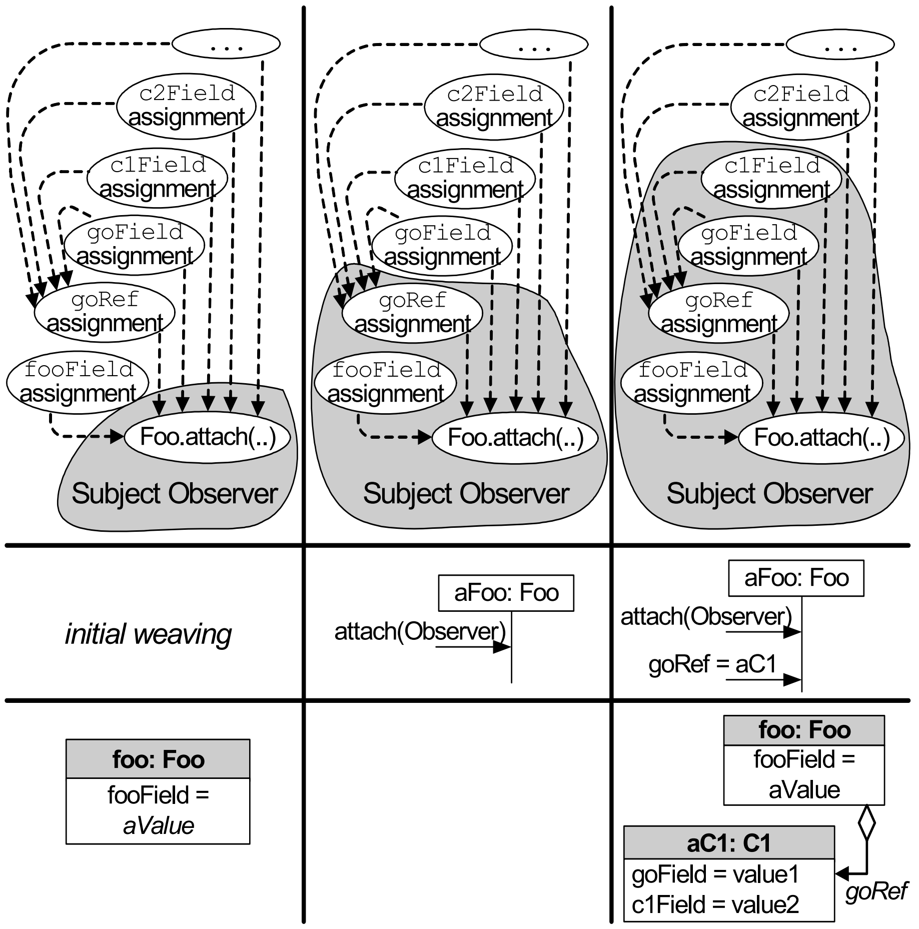
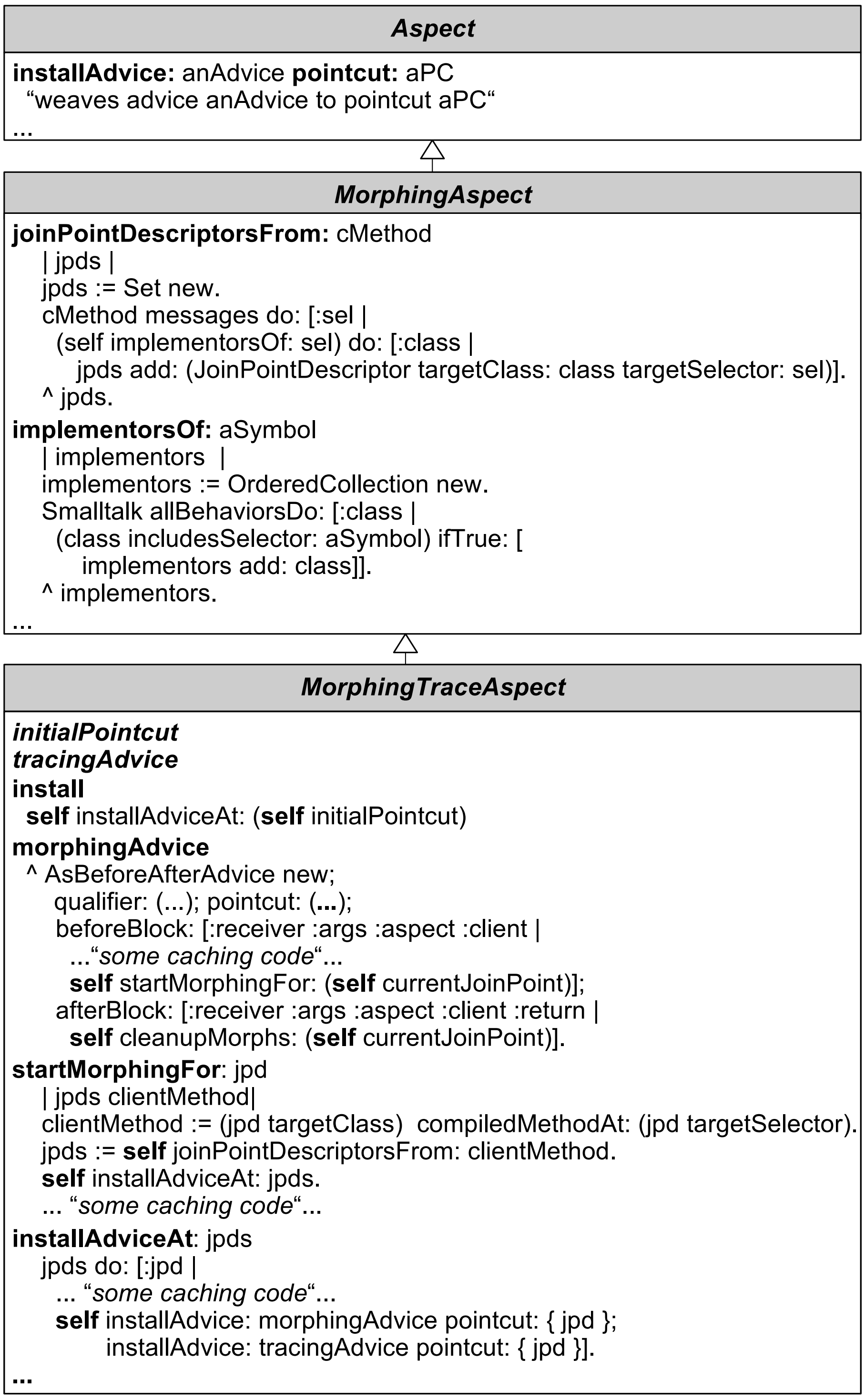


Figure 4-35. Subject-observer as a morphing aspect at initial weave time, after observer attachment, and after assigning an instance of `C1`.

AspectS is based on method wrappers [BFRJ98] (see section 2.4). A shadow for a method execution join point or a method call join point is adapted by wrapping the receiving method. The method to be wrapped is specified by a join point descriptor, which refers to a class and to a method selector. Advice directives in AspectS are runtime objects that refer to a pointcut. Pointcuts are collections of join point descriptors. If advice directives are installed at runtime, all methods referenced by the join point descriptors are wrapped. The wrappers handle the execution of qualifiers (which correspond to join point checks) and the execution of the advice. Advices are implemented by blocks (see section 2.4).

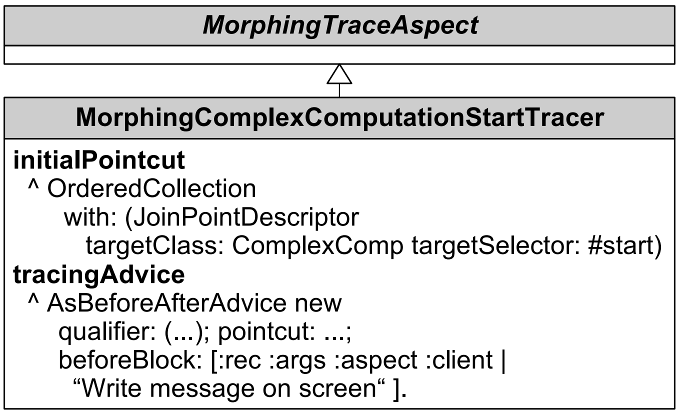


```
Aspect

installAdvice: anAdvice pointcut: aPC
  "weaves advice anAdvice to pointcut aPC"
...

MorphingAspect

joinPointDescriptorsFrom: cMethod
    | jpds |
    jpds := Set new.
    cMethod messages do: [:sel |
      (self implementorsOf: sel) do: [:class |
        jpds add: (JoinPointDescriptor targetClass: class targetSelector: sel)].
    ^ jpds.
implementorsOf: aSymbol
    | implementors  |
    implementors := OrderedCollection new.
    Smalltalk allBehaviorsDo: [:class |
      (class includesSelector: aSymbol) ifTrue: [
         implementors add: class]].
    ^ implementors.
...

MorphingTraceAspect

initialPointcut
tracingAdvice
install
  self installAdviceAt: (self initialPointcut)
morphingAdvice
  ^ AsBeforeAfterAdvice new;
      qualifier: (...); pointcut: (...);
      beforeBlock: [:receiver :args :aspect :client |
        ..."some caching code"...
        self startMorphingFor: (self currentJoinPoint)];
      afterBlock: [:receiver :args :aspect :client :return |
        self cleanupMorphs: (self currentJoinPoint)].
startMorphingFor: jpd
    | jpds clientMethod|
    clientMethod := (jpd targetClass)  compiledMethodAt: (jpd targetSelector).
    jpds := self joinPointDescriptorsFrom: clientMethod.
    self installAdviceAt: jpds.
    ... "some caching code"...
installAdviceAt: jpds
    jpds do: [:jpd |
      ... "some caching code"...
      self installAdvice: morphingAdvice pointcut: { jpd };
             installAdvice: tracingAdvice pointcut: { jpd }].
...
```

Figure 4-36. Tracing as a abstract morphing aspect in AspectS.

Tracing aspects based on morphing aspects are subclasses of the (abstract) class `MorphingTraceAspect` (Figure 4-36). `MorphingTraceAspect` contains the abstract method `initialPointcut` that returns the set of join point descriptors specifying the independent join points whose shadows need to be initially adapted.

The aspect refers to two advice objects, both returned by corresponding methods `tracingAdvice` and `morphingAdvice`. The (abstract) method `tracingAdvice` returns the advice to be executed during tracing; `morphingAdvice` provides the advice starting the morphing process. Our morphing advice contains two blocks which are invoked before and after the corresponding join point is reached. The before block starts the morphing process at the given join point by invoking method `startMorphingFor`. Method `startMorphingFor` determines the runtime-object for the invoked method and computes all join point descriptors that depend on that method (see methods `joinPointDescriptorsFrom:` and `implementorsOf:` in class `MorphingAspect` in Figure 4-36). A shadow is adapted for each of those join point descriptors that invokes the morphing advice as well as the tracing advice (method `installAdviceAt:pointcut:`). In order to use the morphing trace aspect developers have to extend `MorphingTraceAspect` and override `initialPointcut` and `tracingAdvice`. Figure 4-37 illustrates a sample class `MorphingComplexComputationStartTracer`. A tracing aspect is initially woven by instantiating the corresponding class and invoking method `install`.



```
initialPointcut
  ^ OrderedCollection
      with: (JoinPointDescriptor
        targetClass: ComplexComp targetSelector: #start)
tracingAdvice
  ^ AsBeforeAfterAdvice new
     qualifier: (...); pointcut: ...;
     beforeBlock: [:rec :args :aspect :client |
       "Write message on screen" ].
```

Figure 4-37. Concrete tracing aspect as a morphing aspect.

Creating and integrating shadows (i.e. method wrappers) in AspectS is a time-consuming task, yet to be optimized. Hence, it is typically not desirable to start the morphing process at every possible join point. For example, if the methods to be traced are executed quite often, the time consumed for the morphing process can be higher than the benefit of dismissed join point checks. Therefore, the implementation of the tracing aspect in AspectS uses *lazy morphing* by default. Dependent shadows are adapted whenever the join points they depend on are reached and the morphing process did not already start at these join points. Lazy morphing does not release shadows during morphing. Instead, shadow adaptations reside in the system until the developer uninstalls the whole aspect. So, every execution of the morphing process potentially increases the number of adapted shadows for the aspect, but does not delete any.

The use of lazy morphing turned out to be practical in a number of experiments. Those experiments showed that the number of shadows adapted by lazy morphing is still significantly smaller than a complexly woven trace aspect.

4.5 Experimental Results

Table 4-38 summarizes a number of performance measurements in AspectS on a Pentium 4.2 GHz with the Squeak Virtual Machine version 3.4.4. The Smalltalk image contained the **Comanche Http Server**[39] as well as the **Squeak CommandShell**[40]. Overall, the image contained more than 2,200 classes with more than 35,000 compiled methods. For 100,000 times the execution time was measured for the adaptation and release of shadows, the execution time for (empty) methods and the execution time for methods with adapted shadows whose join point checks always pass as well as with shadows whose checks always fail. The adapted shadow was implemented with the successful join point check by an empty around advice whose join point condition immediately succeeds without any additional computation. The adapted shadow with the failing join point check was implemented by a join point condition that immediately fails without any additional computation. The measurement showed that the execution of an unconditional shadow (which always executes the advice without a corresponding join point check) was approximately 90 times (0.0368 ms / 0.0004 ms, see row 4 and 3 in Table 4-38) slower than the execution of an empty method. The execution of a dead shadow (i.e. a shadow whose join point check always fails) was about 9 times slower (0.3269 ms / 0.0368 ms, see row 5 and row 4 in Table 4-38) than the execution of an unconditional shadow[41].

	Average	Mininum	Maximum
Single shadow adaptation	0.1813	0.1756	0.1998
Single shadow adaptation	0.3113	0.1259	0.3445
Method execution (ME) without adapt.	0.0004	0.0002	0.0005
ME with successful join point check	0.0368	0.0346	0.0371
ME with failing join point check	0.3269	0.2555	0.3282

Table 4-38. Experimental Results for shadow creation, deletion and method execution time (in ms) in AspectS.

Next, a single (lazy) morphing tracing aspect was created to trace the execution commands in the command shell. The corresponding advice simply wrote all messages to the screen. The initial weaving of the tracing aspect just adapts a single join point shadow and took 0.45 milliseconds. As soon as the method to be traced has been invoked for the first time, the morphing process started for 35 times creating 253 shadows. This process took about 9.5 seconds. Starting the control flow afterwards did not lead to any additional execution of the morphing process. From then on the execution of the method to be traced took about 2.5 seconds.

[39] Comanche http server, version 6.1, `http://squeaklab.org/comanche/httpserver/`

[40] CommandShell for Squeak - Version 3.0.1, `http://minnow.cc.gatech.edu/squeak/1914`

[41] The reason for the slow execution of dead shadows lies in the way how wrappers and wrapped methods are implemented. Wrappers store the wrapped method in a field. When a join point check fails the original method is executed by calling the time-consuming *value:* method.

This result was compared with a corresponding complete weaving (see Table 4-39). The computation of all potentially invoked methods was not practicable (the computation took more than 3900 seconds). Hence, the same approach like AspectJ was done to weave the aspect to all existing methods in the image (except some system methods). To do so, the aspect was woven to more than 35000 methods. This complete weaving took about 6.8 seconds. The control flow execution afterwards took the same time like the morphing tracing aspect.

	# Adapt.	Time
Morphing Tracing Aspect, Initial Weaving	1	0.45 ms
Complete Weaving with Shadow computation	7930	3949 s
Complete Weaving (no Shadow computation)	35852	6.82 s
1st cflow execution in Morphing Aspect	253	9.54 s
1st cflow execution in Non-Morphing Aspect	35852	2.51 s
cflow execution in Morphing Aspect	253	2.51 s
cflow execution in Non-Morphing Aspect	35852	2.51 s

Table 4-39. Tracing in an experimental environment as morphing aspect and completely woven aspect.

As a result, this experiment showed that the initial weaving of the morphing aspect and the first tracing of the control flow took about 9.54 seconds while the completely woven aspect and a first execution of the control flow took about 9.33 seconds (Table 4-40). The difference of 0.19 seconds is the price for using a morphing aspect instead of a completely woven one. However, the number of adapted shadows by using a morphing aspect is only 1 % of the number of adapted shadows of the completely woven aspect. These shadows decrease the performance of the whole system, because each shadow whose join point check fails causes a runtime overhead of more than 0.3 milliseconds (according to Table 4-38). Because of that, as soon as in the average more than about 640 ((9.54 – 9.33) / 0.0003269) methods with an unsuccessful join point check caused by imprecise join point shadow computation are invoked, the approach of morphing aspects turns out to improve the performance of the overall system.

	Time
Complete Weaving and first cflow execution:	9.33 s (=6.82 s + 2.51 s)
Incomplete Weaving and first cflow execution:	9.54 s (=9.54 s + 0.45 ms)

Table 4-40. Consumed Time for tracing control flow in a completely, and an incompletely woven aspect.

Preliminary experiments showed for example that the response time of the http server contained in the image was a few hundred times slower than before weaving the tracing aspect. This is because weaving the tracing aspect according to the weaving strategy of AspectJ adapted a large number of shadows even in those classes that will never be invoked in the control flow to be traced.

4.6 Related Work

Dynamic weaving in combination with *just-in-time aspects* as proposed in [PGA02, PGA03] is closely related to morphing aspects. Just-in-time aspects are dynamically woven to the system when they are really needed. Furthermore, just in time aspects are woven to the application in *one atomic step* (see [PGA03], page 101). Consequently, just in time aspects do not perform any additional join point checks as long as they are not woven. In that way just in time aspects overcome the problem of unnecessary shadows in comparison to static weaving. Nevertheless, just-in-time aspects are woven completely because of the atomicity property. Hence, after dynamically (and completely) weaving an aspect the problem of unnecessary shadows arises just like in static woven systems.

Another approach that relates to our work on morphing aspects is the *selective just-in-time weave*r as proposed in [SCT03], an extension to the work of just-in-time aspects. The (Java based) selective weaver permits developers to choose between two different kinds of join point shadows: Either as breakpoints in the JVM or as statically embedded hooks. While breakpoints can be created much faster, their execution is time consuming (see [SCT03] for a detailed discussion on the performance issues). Embedded hooks on the other hand execute faster while their creation is quite slow in comparison to that of breakpoints. Selective just-in-time weavers try to overcome the performance overhead caused by frequently executed shadows by embedding such shadows statically. From that point of view, a selective weaver and morphing aspects are similar. The selective weaver causes a performance overhead for embedding hooks in order to achieve a performance advantage for the further execution of the program. Similarly, the morphing process executed by morphing aspects causes a performance overhead to achieve a performance advantage for the further execution of the program. However, the main difference between both approaches is that a selective weaver does not reduce the number of conditional shadows.

The virtual machine *Steamloom* [BHMO04] belonging to the aspect-oriented language *Caesar* [MeOs03] also tackles the problem of time-consuming join point checks. Steamloom implements join point checks and advice invocations at the virtual machine level. In [BHMO04] the performance of advice making use of the join point checks on VM level and the statically woven aspects in AspectJ based on the cflow construct is measured. The result shows that the Steamloom VM has a significant performance advantage over the completely woven approach of AspectJ. The intention of Steamloom and morphing aspects is very similar since both tackle the performance overhead caused by join point checks. The difference between Steamloom and the implementation of morphing aspects as proposed in this paper is that while weaving in Steamloom is performed by redirecting messages at the VM level our AspectS-based implementation carries out changes to the runtime representation of methods at the application level.

Besides the approaches that provide pure dynamic weaving there are also approaches that remove unnecessary runtime checks based on a static analysis. For example [MKD03] describes a *partial evaluator* based on the definitional interpreter specified in [WKD02] to reduce the number of unnecessary join point checks. In [SeMo04] a reduction of join point checks is achieved by a static analysis of the call stack. Currently, there are experimental results that compare the number of failing join point checks caused by these approaches with the number of failing join point checks caused by morphing aspects within an experimental environment.

4.7 Chapter Summary and Conclusion

This chapter addressed the problem of unnecessary join point shadows caused by complete weaving. The problem was motivated by illustrating two typical examples for aspect-oriented programming and their implementation in the aspect language AspectJ. The chapter proposed morphing aspects to overcome the problem of unnecessary join point checks. Morphing aspects are incompletely woven aspects that change their set of join point shadows at runtime based on a continuous weaving process.

With incomplete weaving, not every shadow within the base system whose join points potentially execute aspect-specific code is adapted. Instead, morphing aspects utilize dependencies among join points and their shadows that permit to delay the adaptation of shadows just to the point when join points they depend on are reached. As a result, the number of adapted shadows of a morphing aspect is much smaller in comparison to that of completely woven aspects. This is because dependent join point shadows are not adapted initially, but at a later point in time when they are actually needed. Experiments with morphing aspects in the aspect-oriented system AspectS showed that by using morphing aspects the number of join point shadows is significantly reduced. In that way, the performance overhead caused by failing join point checks is reduced, too. However, it should be noted that the performance overhead of join point checks in AspectS is quite high as shown in section 4.5. Hence, the benefit of morphing aspects is much higher in a system where the costs of join point checks are rather expensive than in systems where join point checks are less expensive.

The benefit of realizing an aspect as a morphing aspect depends on a number of influencing factors. In general, a prerequisite for the successful application of morphing aspects is a large number of failing join point checks during the execution of a program. According to the examples in section 4.2, such a prerequisite is fulfilled if, for example, a tracing aspect is to be implemented in an application with a large number of threads that never invoke the method where tracing should begin. In addition, such a prerequisite is fulfilled if instances of a class are only very rarely observed during the execution of a program. The prerequisite is usually not fulfilled if the aspects in the system hardly rely on join point checks, i.e. if the woven application mainly consists of unconditional join point shadows.

The morphing process needs additional time to determine and create dependent join point shadows. Developers must trade-off between the runtime overhead caused by unnecessarily introduced runtime checks caused by unnecessary adapted shadows and the overhead caused by the morphing process itself.

Morphing aspects impose a number of requirements on the underlying aspect-oriented system. This restricts their application to a number of systems. The most fundamental requirement is that the underlying system must permit dynamic weaving, i.e. weaving of aspects during runtime. A number of systems such as *PROSE* [PGA02, PGA03], *AspectS* [Hirs02], *JAC* [PSDF01], *Object Teams* [VeHe03, Herr02], or Caesar [MeOs03] fulfill this requirement while systems like *AspectJ* [KHH+01] or *Sally* [HaUn03a] do not. As another requirement morphing aspects typically require the computation of dependent shadows at runtime, i.e. the shadows to be associated with an aspect are statically not known. However, not every system providing dynamic weaving permits the computation of join points at runtime. For example, Object Teams assumes that the shadows are statically declared.

The contribution of this chapter for the whole thesis is mainly that morphing aspects are a special way of specifying aspects. The reflective characteristics of the underlying programming languages are utilized in order to determine join points and the characteristic of dynamic weaving is utilized in order to achieve continuous weaving. The approach is not really a new system in the sense that a software framework is provided. Instead, a conceptual framework is provided that determines how to specify morphing aspects.

Obviously, morphing aspects represent a very special form of aspect-oriented system that relies on a special kind of weaving. The design dimensions that are described in the following chapter identify the underlying design decision as a special kind of aspect-oriented system in respect to weaving (see section 5.6.3).

5

DESIGN DIMENSIONS OF ASPECT-ORIENTED SYSTEMS

5.1 Introduction

There is already a large number of systems which are (commonly accepted) called aspect-oriented systems. However, no commonly accepted definition is available which precisely determines whether a certain system is aspect-oriented. As a consequence, whenever a new system appears it is difficult to determine whether it is legitimate to call such a system aspect-oriented. Furthermore, it is difficult to compare different aspect-oriented systems because underlying criteria for such a comparison are missing.

The main problem that needs to be addressed is formulated in [Film01]: Answering the question *"Is X an AOP system where the answer is not based on whether the creator of the system called it AOP"*.

Nowadays, the only possibility to determine whether or not a system is aspect-oriented is to trust on an author who proposes a new system and calls it aspect-oriented, or to trust the program committee of conferences like the **International Conference of Aspect-Oriented Systems** (AOSD, [Kic02, Aks03, Lieb04]) that accepts papers (or demonstrations) about systems which contribute to the aspect-oriented community. However, this is far from being satisfying because it simply delegates the analysis of a system with respect to its *aspect-orientedness*, but it does not provide any differentiating attributes that can be used for such an analysis.

Although there are a number of systems that are called aspect-oriented, it turns out that some systems are more appropriate to modularize a given crosscutting problem than other ones (see for example section 3.3). Actually, certain systems are even inappropriate to modularize a given crosscutting problem.

From a developer's perspective such a situation is not satisfying because it is not possible to determine on an abstract level what characteristics a system needs to provide in order to solve a given problem. Instead, a detailed knowledge about system-specific features is needed in order to determine a system's adequacy. For example, if developers need to determine whether AspectJ permits to solve the crosscutting problem caused by a tracing feature, they need to study the language features of AspectJ: The mechanisms and the underlying terminology of AspectJ need to be understood in detail in order to determine whether AspectJ permits to solve the problem. In case the system turns out to be inappropriate, they need to study the language features of for example Sally. In case Sally turns out to be inappropriate, it is necessary to study Hyper/J, etc. Consequently, developers need to study in detail a large number of systems that claim to modularize crosscutting concerns. In the worst case, they study all systems that claim to be aspect-oriented and finally might conclude that there is no system that permits to modularize their crosscutting concern in a desired way.

The same problematic situation arises for developers who want to build their own aspect-oriented system. They need to study those features of existing systems that address the modularization of crosscutting concerns. Since there are no system-independent characteristics for aspect-oriented systems available, developers need to study a large number of systems in detail and analyze on their own the impact of a certain feature in order to modularize a crosscutting concern. If such developers have a certain crosscutting problem in mind, they need to analyze on their own a) if there is a system that solves this problem and b) what features of this system are responsible for solving this problem. Finally, developers need to select all desired features and analyze whether such features can be added to a new system in parallel. Currently, this process cannot be shortened because high-level abstractions of aspect-oriented systems are missing; therefore, a detailed analysis of each aspect-oriented system is necessary.

In order to address the previously described problems, high-level abstractions of aspect-oriented systems are needed that describe

- The inherent characteristics of aspect-oriented systems in order to analyze a system with respect to its aspect-orientedness,
- The distinguishing variations of aspect-oriented systems in order to compare existing systems, and
- The distinguishing variations in order to demonstrate the different alternatives for the design of new systems.

Furthermore, it is desirable to characterize a given crosscutting problem in terms of an abstract problem description. This permits to map a given problem to systems that potentially solve the problem without relying on system-specific mechanisms and terminologies. The benefit of such an abstract description is that problems can be classified into different categories and mapped to different aspect-oriented systems without the need to have a detailed knowledge about system-specific features.

5.1.1 Aspect, Join Point, and Weaving

In the aspect-oriented literature, in general, as well as in the papers describing the approaches discussed in the previous chapters, there are three commonly used terms: **Aspect**, **join point,** and **weaving**.

- An aspect (on the programming language level) is a modularized unit containing elements, which would crosscut the base program if not handled by an aspect-oriented system. From the implementation's point of view, if the aspect-specific code would be factored out into known language constructs, there would be a number of code redundancies in the base system. From a more abstract point of view (see for example [SGC02], [McHs03]), an aspect represents a unit that modularizes a crosscutting concern[42].

- The term join point is defined in [KHH+01] as a *principled point in the execution of a program*. Aspects potentially affect the base system at a number of such join points. A **weaver** that composes a number of aspects with the base system achieves this affection.
- Weaving describes the process of integrating aspects with the application (the **base system**). In order to weave a number of aspects, a weaver determines those join points in the base system the aspects refer to. At those join points the weaver establishes the connection to aspect-specific code.

Figure 5-1 schematically illustrates the integration of two aspects with the base system. The base system provides a number of join points the aspects refer to (the join point are not explicitly illustrated). The weaver takes the base system and the aspects as input parameters and composes the woven system – a system that contains the aspects in addition to the base system.

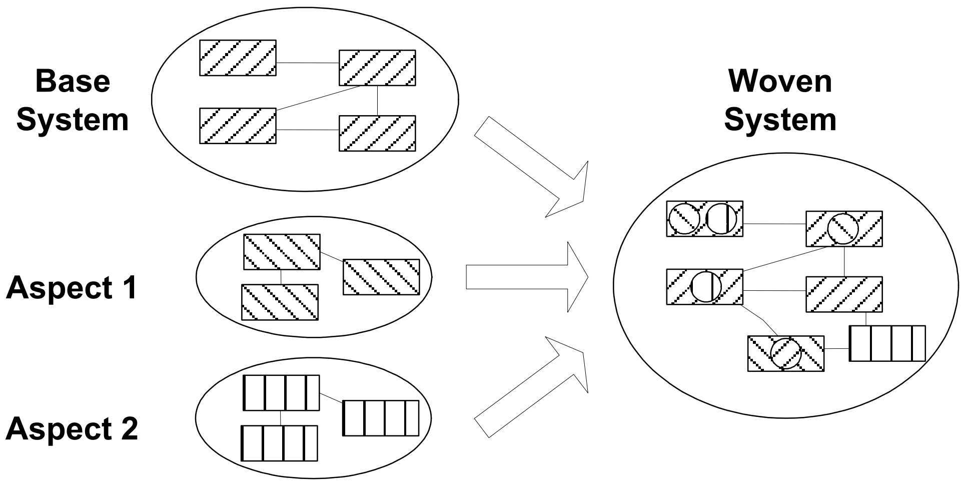


Figure 5-1. Schematical illustration of weaving two aspects to a base system [HSU03].

Although from this generic point of view there seems to be some agreement on these terms, the terms are differently used in different situations. For example, the conceptual model of join point differs in different aspect-oriented systems, the way an aspect describes its join points differs, and the way the weaving process is implemented in different aspect-oriented systems differs, too. The differences in implementations of different aspect-oriented systems has a large impact on how the systems can be applied and how appropriate a system is in order to solve a given crosscutting problem. Furthermore, the terms aspect, join point, and weaver are not well defined. It is necessary to determine the characteristics of aspects, join points, and weavers and their

42 Filman noted in [Film01] that this definition contains some circularity, because once a crosscutting concern is handled by an aspect-oriented system it is no longer a crosscutting concern. [LLM99] uses the term **aspectual paradox** to describe the same problem of circularitiy in the defintion.

commonalities and differences in known approaches. For example, it is necessary to study whether a class-construct known from class-based object-oriented programming languages is able to fulfill the characteristics of aspects. Although there is some agreement in the literature that known language constructs like classes cannot represent aspects, it is not clear on what observation this agreement is based on.

The previously introduced terms do not permit to distinguish between different kinds of aspect-oriented systems: Whether or not two systems are able to modularize the same kinds of crosscutting concerns cannot be characterized by these terms, because such terms only represent some commonalities among aspect-oriented systems.

Consequently, the previous terms represent a first common vocabulary for aspect-oriented systems, but they are not sufficient to distinguish aspect-oriented and non-aspect-oriented systems, to distinguish between different aspect-oriented systems, or to analyze a system with respect to its appropriateness to solve a crosscutting problem.

5.1.2 Quantification and Obliviousness

On a more abstract level (that leaves out the terms aspects, join points and weaving) a large number of authors agree on **quantification** and **obliviousness** as described in [FiFr00] as the core elements of aspect-orientation (see for example [TuKr03, KHH+01, Nord01] among many others).

Quantification describes the capability of aspect-oriented systems to *make quantified statements about which code is to execute under which circumstances* [FiFr00]. This corresponds to the previously explained idea that aspects influence an application at a number of join points: Join points are the circumstances; the code to be executed under these circumstances is the aspect-specific code for such join points.

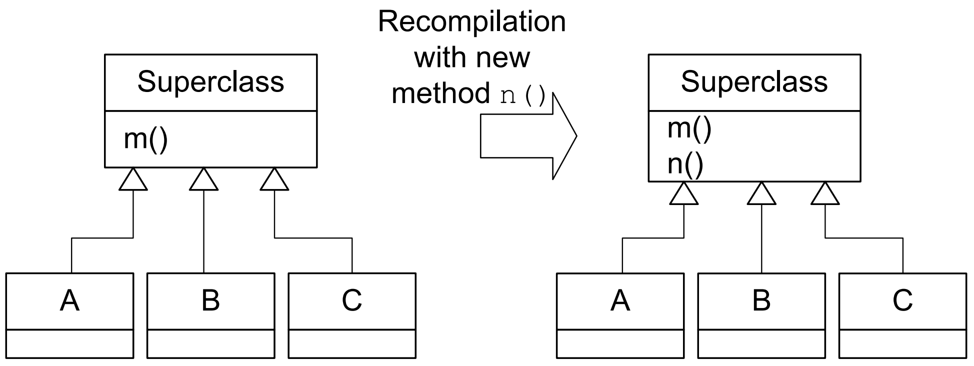
Obliviousness describes the capability of aspects to extent an application without the need to preplan such an extension[43]: The developer of a certain module or application does not need to consider additional concerns that might influence the code. Consequently, the developer does not need to provide explicit hooks that can be used later on for supplementing additional concerns to the code. Instead, aspects specify on their own those hooks in the application to determine where aspects contribute to the original module. An alternative term for obliviousness (or oblivious extension) is the term **unanticipated reuse** [Knie00] (see also [USE04]).

However, although the terms quantification and obliviousness are appropriate to give a general impression of aspect-oriented software development, they are too general and too closely related to existing approaches in order to represent distinguishing characteristics that separate aspect-oriented from non-aspect-oriented systems. For example, inheritance in object-oriented programming languages like Java can be regarded as a language construct providing oblivious extensions that make quantified statements about the class hierarchy (see for example [FiFr00, ClLe03]). Each method defined in a superclass can be used by all subclasses and by all clients that access such

[43] The preplanning problem was addressed for example by [CzEi00]. However, it should be noted that there are some authors that do not agree on obliviousness as a core characteristic of aspect-oriented system. For example [RaCh03] argues that there are aspects that need to be explicitly addressed within the application and which are in that way not completely oblivious.

subclasses (or objects of the corresponding types). A superclass can be compiled later on after its subclasses are compiled. Hence, adding a new method to a superclass represents a "quantified extension".

Figure 5-2 illustrates a simple class hierarchy in Java. A class `Superclass` provides a method `m` that can be used by all clients accessing instances of the subclasses `A`, `B` and `C`. However, the message `n` is not accepted by such instances. After adding a new method `n` and recompiling `Superclass`, this method is available for all instances of `A`, `B` and `C`. Spoken in aspect-oriented terms: The new method crosscuts all subclasses. From the quantification point of view a method definition in a superclass is a quantified statement for all instances in the subclass. The corresponding statement is "*whenever an object that is instantiated by a class subclassing* `Superclass` *receives a message* `n`*, execute the corresponding method defined in* `Superclass`"[44].



```
Valid calls:                Invalid calls:              Valid calls:
A a = new A(); a.m();       A a = new A(); a.n();       A a = new A(); a.m(); a.n();
B b = new B(); b.m();       B b = new B(); b.n();       B b = new B(); b.m(); b.n();
C c = new C(); c.m();       C c = new C(); c.n();       C c = new C(); c.m(); c.n();
```

Figure 5-2. Quantified and oblivious extensions via superclass recompilation.

Obviously, the intention of aspect-orientation is not to reinvent new terms for already existing mechanisms. Hence, it seems rather undesirable to call the previously described extension an aspect-oriented approach. Furthermore, it seems clear that changing a class definition in an invasive way and recompiling the class cannot seriously be regarded as a new approach: The extension of the original class is not supported by extension mechanisms of the underlying language. However, the previous example can be described by the terms obliviousness and quantification. Hence, those terms seem to be rather inappropriate to describe inherent characteristics of aspect-oriented systems.

[44] It should be noted that there are reasons why such an approach is rather problematic. Since Java is a stronlgy typed programming language, classes that include invocations of the the new method `n` in type `Superclass` cannot be compiled as long as the new method is not included in `Superclass`. Consequently, classes using `n` must be compiled after recompiling `Superclass`.

5.1.3 This Chapter's Outline

This chapter provides an conceptual description of aspect-oriented systems by providing so-called **design dimensions** aspect-oriented systems are based on. These dimensions are extracted by comparing similar points of view in the aspect-oriented literature on the core ingredients of aspect-orientation, and by comparing the systems introduced in previous chapters. Each aspect-oriented system can be described in terms of how each design dimension is applied. Consequently, different systems can be compared with respect to how each dimension is met in the aspect-oriented systems.

As pointed out by some authors (see for example [RaSu03]), AspectJ is a relatively *rich* aspect-oriented system. Therefore, a number of design decisions of an aspect-oriented system can be explained in terms of AspectJ. Since AspectJ is currently the most popular aspect-oriented system, the proposed design dimensions will be exemplified in AspectJ whenever possible.

Section 5.2 discusses briefly the core ingredients of aspect-oriented systems. This section introduces those elements of aspect-oriented systems whose design dimensions are introduced more deeply in the following sections. Section 5.3 introduces different design dimensions of join points, and section 5.4 introduces different design dimensions of join point selection constructs. In section 5.5 the kinds of join point adaptations are described, and section 5.6 discusses different design alternatives for weavers. Section 5.7 discusses the relationship between the design dimensions and their implementations. Section 5.8 critically discusses the term design dimensions chosen to describe the approach in this chapter. Finally, this chapter is concluded.

5.2 A Model for Aspect-Oriented Systems

The introduction gave a first impression of the elements of aspect-oriented systems: Aspects are modules that are woven to the base application at certain join points. So, the intention of aspect-oriented systems is to add new concerns to the base system. However, it does not clearly determine the tasks an aspect-oriented system has to perform and the ingredients an aspect-oriented system consists of.

In an aspect-oriented system, developers have to specify aspects in a way that expresses:

1. *Where* the aspects are woven to the base system, and
2. *How* the aspects adapt the base system.

The "where" inside this specification is expressed in terms of join points, i.e. developers depend on some features provided by the underlying system to describe those join points the aspect should be woven to. For specifying the "how" developers also depend on features provided by the underlying system which permit to describe exactly how each chosen join point is to be adapted.

As a consequence, the aspect-oriented system must provide the following elements:

1. A join point representation of the base system,
2. Language constructs that permit developers to select join points,

3. Language constructs that permit developers to specify how selected join points are to be adapted, and
4. A weaver that composes the resulting system.

A **join point representation** is a specific view on the base system, which needs to be extracted directly from the base application itself: It is the aspect-oriented system's task to decompose the base application into a number of join points as illustrated by means of cycles in Figure 5-3. Therefore, developers of an aspect-oriented system need to decide what elements of the base application represent join points. This decision is based on what **join point model** the aspect-oriented system is based on. The design dimensions of join point models are discussed in section 5.3.

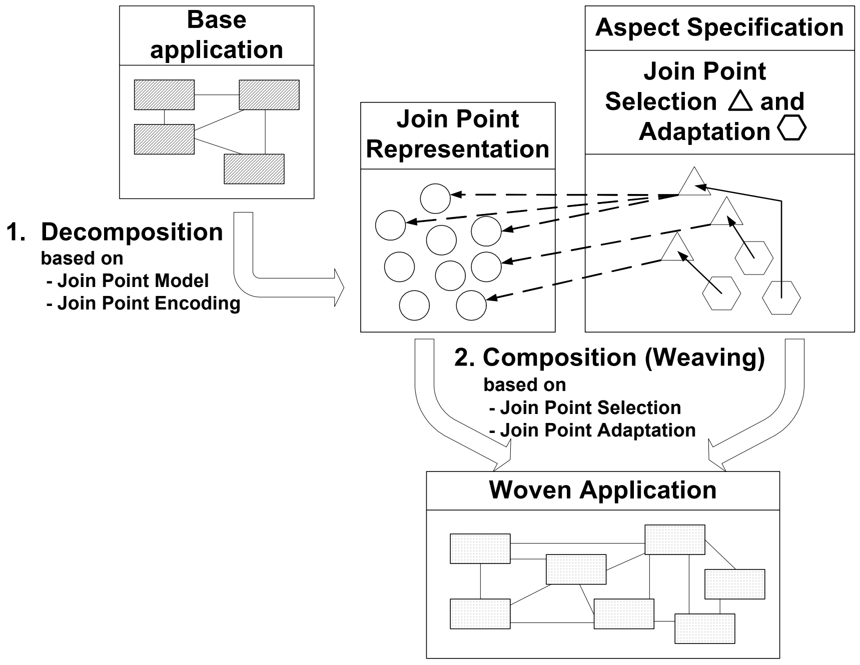


Figure 5-3. Schematical illustration of the ingridients and tasks of an aspect-oriented system.

Developers must be able to select join points. This means the aspect-oriented system must provide a **selection language** that permits to address those join points the base system is decomposed into. The selections are user-defined elements that refer to the system's join point representation as illustrated by means of triangles referring to the cycles in Figure 5-3. Furthermore, this implies that the system must equip join points with a number of characteristic marks or properties that permit the developer to decide whether or not a join point is to be selected. These characteristics are used for the join point representation, i.e. when the base application is decomposed the selection language is provided to the developer to address this representation. The extraction of characteristic marks of join points is called **join point encoding** in this thesis and is discussed in conjunction with characteristics of **join point selection languages** in section 5.4.

The aspect-oriented system needs to provide developers with language features that refer to a join point selection and that specify how the selected join points are to be adapted. Such **join point adaptations** are user-defined elements as well as illustrated by means of pentagons referring to the triangles in Figure 5-3. The design dimensions of the corresponding language constructs are discussed in section 5.5.

Finally, the aspect-oriented system needs to weave the resulting system based on the user-defined selections and adaptations and based on the modules in the base-application. Thereto, the **weaver** uses the base system, its join point representation, and the aspect specifications and composes a woven system. The different alternatives of weaving are discussed in section 5.6.

Consequently, this thesis regards a system to be aspect-oriented, if it provides a join point model, constructs for selecting join points, constructs for adapting join points, and a weaver that composes a resulting system.

5.3 Join Points

Join point is the central concept that underlies all aspect-oriented techniques. Nevertheless, there are different interpretations of the join point concept which result in different implementation techniques for aspect-oriented systems.

The term join point originates from Kiczales et al in [KLM+97] who define join points as *those elements of the component language semantics that the aspect programs coordinate with*[45]. However, the phrase "elements of a semantics" is quite imprecise. It could refer to an **evaluation rule** in the programming language, like a beta-reduction rule in the **lambda calculus** (see [Bare84] for a detailed introduction into the lambda calculus). Or, it could refer to the rules that define in the programming language Java how expressions are evaluated. Likewise, only certain ingredients of such rules could represent join points.

It is difficult to get a common understanding of the term join point from this definition. Programming languages and their semantics differ widely. For example, in statically typed languages like Java the static types that are assigned to every expression in the application are used by the semantics of the programming language: Static types of parameters (i.e., types that result from a static analysis of the program) are used to compute methods used for the single dispatching[46]. From that point of view, static types of expressions could represent join points. However, in untyped languages like Smalltalk, expressions do not have any static types. Because of that, such types cannot represent join points in an application written in an untyped base language.

Consequently, the term join point as introduced in [KLM+97] does not really help to understand what a join point actually is. Developers of new systems face the problem again and again to investigate on their own what a join point is and what design alternatives for join points exist. For developers that simply want to use an aspect-oriented system, this definition is not satisfying because it does not clearly determine what a join point is. Consequently, the decision of whether or not a certain system is

[45] The term **component language** describes the language that is used to specify the base program. Hence, this thesis uses in the following the term **base language**.

[46] See for example [Bruc02, CLCM00] for descriptions of the single dispatching of Java.

able to handle a given crosscutting problem cannot be determined from this vaguely described term.

Based on the ingredients of an aspect-oriented system as introduced in section 5.2, it is possible to define join points with respect to their role within the system.

> **Join Point**: A join point is an element within an aspect-oriented system that can be selected by the underlying join point selection constructs (specified in a separate module) and that can be adapted by the underlying adaptation mechanisms.

This join point definition has some similarity to [WKD02] where the authors state that a join point is *"[…] an event in the execution of the program at which advice may run*". According to this definition, the ability of being modifiable (and previously selectable) is an inherent characteristic of join points. However, the join point definition leaves open what kind of element is selectable and adaptable because different aspect-oriented systems differ with respect to the nature of join points. This will be discussed in the following sections.

5.3.1 Static and Dynamic Join Points

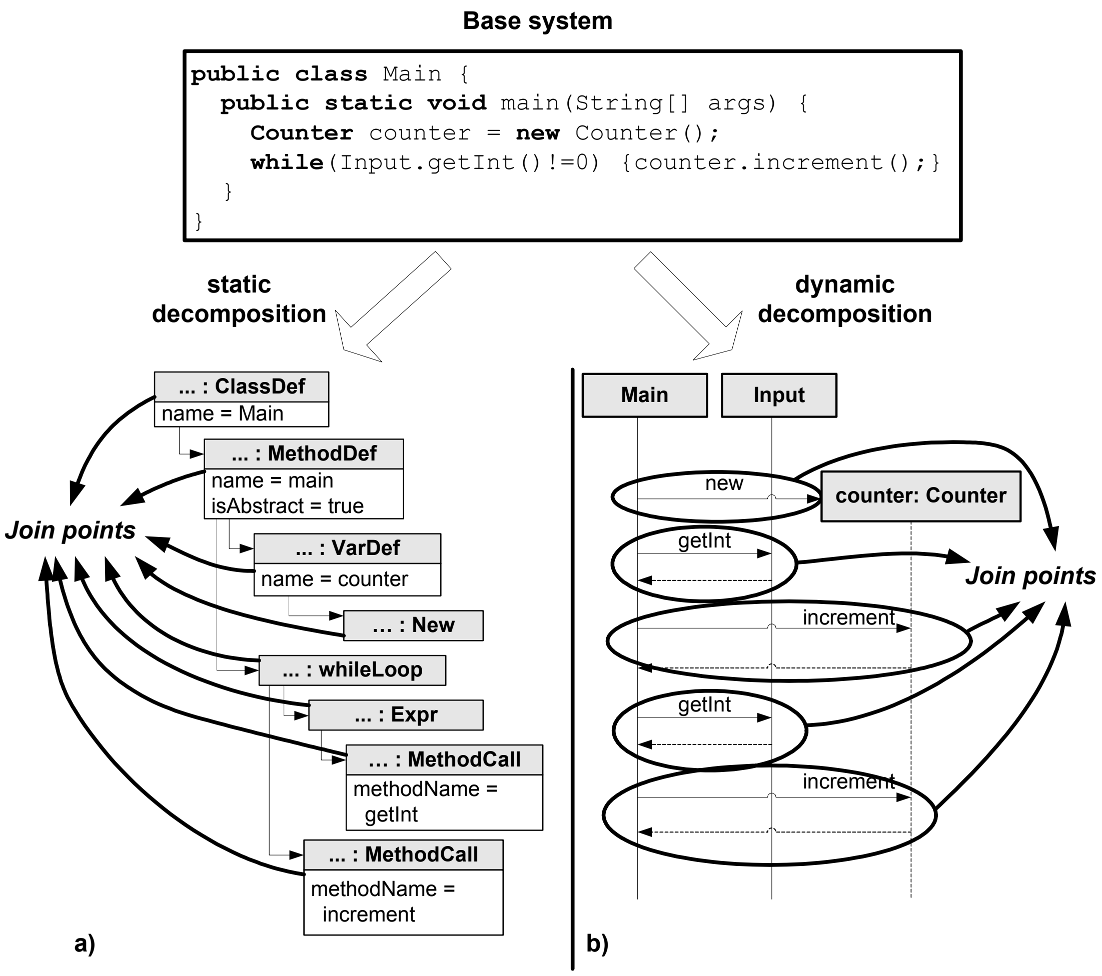
In [KHH+01] Kiczales et al present a slightly more specific interpretation of the term join point (which focuses on AspectJ): *Join points are principled points in the execution of a program.* From this point of view, runtime-specific actions like the evaluation of expressions are join points. Static properties that do not have a runtime representation like static types of expressions in programming languages similar to Java or C++ are no join points according to this point of view.

A different interpretation of the term join point that does not rely on any specific implementation can be found in [Film01]. There, Filman describes join point as *systematic loci* in an application. Similar interpretations can be found by Aßman and Ludwig in [AßLu99] who describe join points as *points in a component where aspects are integrated* or by Batory in [Bato03] who describes a join point simply as *an element of a program.* Also, a similar interpretation of the term join point by Ossher can be found in [EAK+01] who states (from the Hyper/J perspective) that join points are elements in a hyperslice (whereby each element is a class, method, or field declarations). According to this definition, join points are deemed to be static elements of an application. Such static elements might be method or class definitions as well as method calls (i.e., those syntactical elements in the application that represent a method call) or even operators (i.e., those syntactical element in the application which represent the application of operators).

The main difference between the interpretation of join points as "principled points in the execution of a program" or as "systematic loci" lies in their implications with respect to the computability of join points. The latter interpretation extracts all join points from the underlying code base; join points have a direct correspondence in the base application's code. As a consequence, all existing join points can be determined by examining the application and decomposing it into its elements. Considering that an application's code has a finite length, the decomposition leads to a finite number of join points.

The "principled execution point" interpretation of join points does not permit to statically determine all existing join points because join points appear not before runtime and cannot be statically computed upfront in general. In case the application does not terminate, there are potentially an infinite number of join points.

The difference between these interpretations of join points is illustrated in Figure 5-4. The method `main` in class `Main` repeats the while loop as long as the user input (retrieved by the class method call `getInt` in `Input`) is not 0. Every time the loop is executed and the user input is not `0`, the method call to `increment` is executed.



```
public class Main {
  public static void main(String[] args) {
    Counter counter = new Counter();
    while(Input.getInt()!=0) {counter.increment();}
  }
}
```

Figure 5-4. Decomposition of a simple Java application into join points using a) a static decomposition and b) a dynamic decomposition.

In this example the interpretation of join points as systematic loci means that all loci in the application represent join points. Figure 5-4 a) illustrates a possible static decomposition of the application according to its syntactical elements: The class `Main` is decomposed into its syntactical elements class definition, method definition, variable definition (and initialization), loop, expression, and method call (which is typically a specialized expression). Each element in the diagram represents a locus in the application and each locus in the application represents a join point. The number of join points is statically computable (i.e. prior to runtime) and does not change during an application's runtime.[47] Furthermore, for a given element in the code it can be determined whether it represents a join point. For example, single keywords like **class** do not have a representation in the given decomposition. Hence, they do not represent a join point[48].

Figure 5-4 b) illustrates the interpretation of join points as principled points in the execution of a program. This is done by a message-sequence-chart. Each message, as well as the creation of objects, represents a join point in the message-sequence-chart. For example, the first invocation of the `new` operator is one join point, and each invocation of method `getInt`, as well as method `increments` is an additional (and new) join point. Since it cannot be statically computed how often the loop inside the application will be executed, the number of join points in the application cannot be computed before runtime[49].

A number of authors use the term **dynamic join point** to refer to the "principled point in the execution" interpretation of the term join point (see for example [ClLe03, CHMB03, Läm03, MKD03, WKD02]). However, no common definition for this term is provided that is precise enough to characterize the underlying interpretation of join point. For example, dynamic join points are described as *runtime actions* [MKD03], *runtime events* [ClLe03] or *nodes in the runtime call-graph* [ClLe03]. However, such definitions are not appropriate to describe the difference to join points as systematic loci because each runtime event is caused by a corresponding locus in the application. Furthermore, there are runtime actions that are usually not considered join points. For example, garbage collection in systems like Java enforces a number of runtime actions. However, these actions are typically not enforced by the underlying application base, but by the underlying programming language.

The interpretations of join points as "principled points in the execution of a program" and "systematic loci" are not unrelated. Some loci in the application potentially represent a number of points in the execution of a program. The term **join point shadow** [MKD03] is used to map both views (see also Chapter 4 for a detailed discussion of join point shadows). According to [MKD03], a join point shadow is an element within the code that represents a number of join points at runtime. The term join point shadow cannot be applied to loci in an application that does not have a direct correspondence in the application's runtime. In its easiest case, such loci are syntactical elements like white spaces in the Java syntax that are extracted during parsing and that do not influence the runtime behavior of the system. In other situations, some code

[47] This assumes than the underlying language is not **reflective** (see [Maes87]). For example, Java is not completely a reflective language although it is **introspective** (i.e. the **intercessional** characteristics is missing, see for example [Chib00] for a detailed discussion of both terms).

[48] This decomposition of the application into a number of join point corresponds to the decomposition of applications into static metaobjects as realized by static metaobject protocols like for example OpenC++ [Chib95] or OpenJava [TCIK00].

[49] This concrete example is due to the unknown input by the developer. In general, it is due to the **haltproblem** formulated by A. Turing and discussed in many books on theoretical computer science like [HMU01].

elements are replaced by other elements for optimization reasons. Consequently, the original loci in the source code may not exist.

This thesis distinguishes between two different kinds of join points. This distinction is based on their **dynamicity**, i.e. on how they are represented in the base application's code. This thesis distinguishes between **static join points** and **dynamic join points**.

Static Join Point: A static join point is a selectable and adaptable item that represents an element in the base program's code.

Dynamic Join Point: A dynamic join point is a selectable and adaptable runtime-element from the application's execution context. Dynamic join points are represented by elements in the code, which are called **join point shadows**.

The definition of static join point relies on the phrase "element in the base program's code". Such an element may be either a pure syntactical element that can be extracted while parsing the application's sources, like parse nodes defined by the base language's grammar or single tokens in the base application's sources. Alternatively, such elements can be parts of an application's binaries (assuming a compiled language) or (static) metaobjects (assuming an underlying language providing metaobjects).

Examples for dynamic join points are method calls from one specific object to another one. In class-based languages like Java objects do not have any direct correspondence in an application's code but are represented by classes and variables. In this case, objects themselves could represent dynamic join points in class-based programming languages.

5.3.2 Structural and Behavioral Join Points

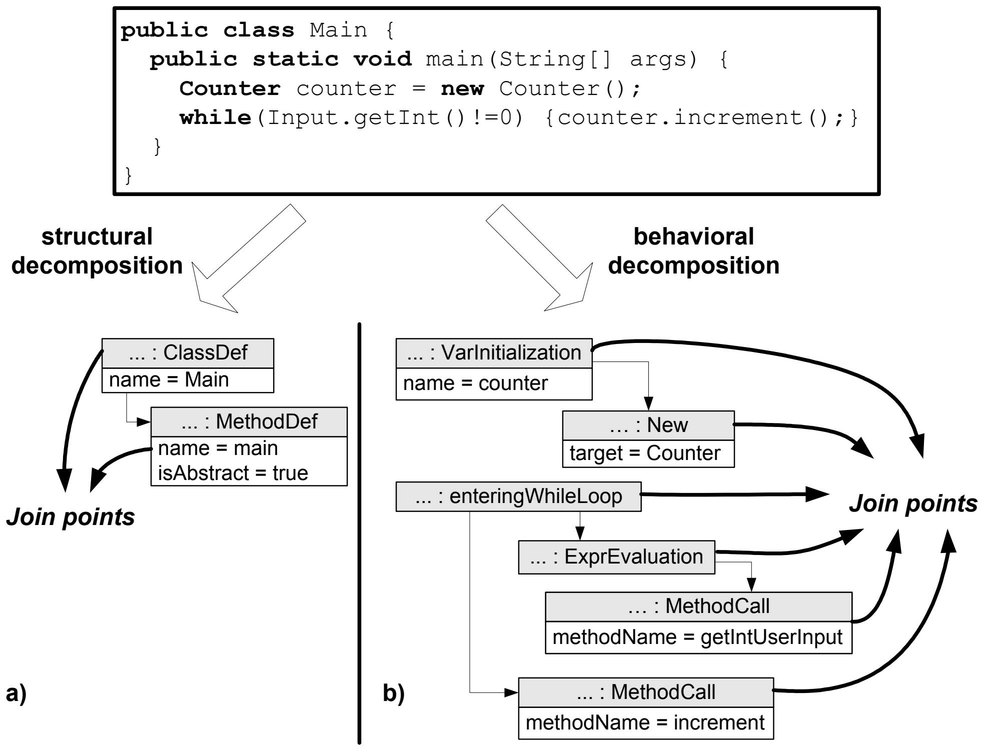
In [EAK+01] Ossher discusses the interpretation of join points as structural elements of an application. As an example he names classes, interfaces, methods, and variables (i.e. abstraction constructs of the underlying programming language) as join points.

For example, within a Java application a certain class might represent a join point. The underlying language provides the class construct and it is up to the developer to use this construct in its concrete application. Class definitions represent **building blocks**[50] that structure the underlying code and define an interface for its objects. In the same way, methods in class-based object-oriented languages represent such building blocks on a finer level of granularity since in such languages methods are part of classes.

This differs from the interpretation of join points as a node in the dynamic call graph as argued by Lieberherr in [EAK+01] and which is closely related to [KLM+97]. Method calls appear at runtime (i.e., they represent nodes in the call graph). They do not represent a structural abstraction in the underlying programming language but are encapsulated within method definitions. In a number of programming languages like Java or C++, method calls are completely hidden from the developer. In such

50 cf. [Meye98], pp. 165

languages, it is not possible to reflect on method calls that are potentially performed by a certain object[51].



```
public class Main {
  public static void main(String[] args) {
    Counter counter = new Counter();
    while(Input.getInt()!=0) {counter.increment();}
  }
}
```

Figure 5-5. Decomposition of a simple Java class into (a) structural and (b) behavioral join points (based on static join points).

Obviously, both views on the term join point differ widely – while Ossher considers structural elements that encapsulate and hide information as join points, Lieberherr focuses on elements representing the encapsulated behavior. This thesis identifies a design dimension of join point models that permits to distinguish between both. The underlying criterion for this distinction is the **level of abstraction**. This thesis uses the term **structural join point** to describe the interpretation of join points as structural elements of an application and **behavioral join points** to describe the interpretation of join points as elements that represent the behavior of an application.

Structural Join Point: A structural join point is a selectable and adaptable element that represents a structural abstraction within the application based on an abstraction provided by the underlying programming language.

Behavioral Join Point: A behavioral join point is a selectable and adaptable element in the application that represents a part of the application's behavior that is encapsulated by corresponding structural building blocks.

51 The reflective capabilities of Java permit only to introspect classes and methods, but no expressions within methods.

Examples of potential structural join points within class-based, object-oriented programming languages are class definitions, method definitions, or objects. Examples of behavioral join points are method calls, field assignments, and operators.

Figure 5-5 illustrates the difference between structural and behavioral join points in a simple Java application. The illustration is based on a static decomposition of the application (i.e., it makes use of the static join point interpretation) as discussed in the previous section[52]. Structural join points decompose the application according to structural elements, (e.g. classes and methods). Accordingly, in Figure 5-5 (a) the application is decomposed into a representation of the class `Main` and a representation of the method `main`. In contrast to that decomposition, behavioral join points decompose the application into all elements that represent the behavior of the application (e.g. expressions that are evaluated at runtime). For example, in Figure 5-5 (b) such elements are the initialization of the variable `counter`, the entry into the loop, or the invocation of method `getInt` in class `Input`.

Whether a certain kind of join point is behavioral or structural can be derived from the underlying language, because the base language's specification clearly specifies what constructs encapsulate behavioral elements and what elements represent behavioral elements that do not structure the application's code.

5.3.3 Orthogonality of Design Dimensions

The dimensions **level of abstraction** and **dynamicity** can be applied independently of each other, i.e. both dimensions are orthogonal. As a consequence, join point models underlying aspect-oriented systems can be divided by both dimensions at the same time.

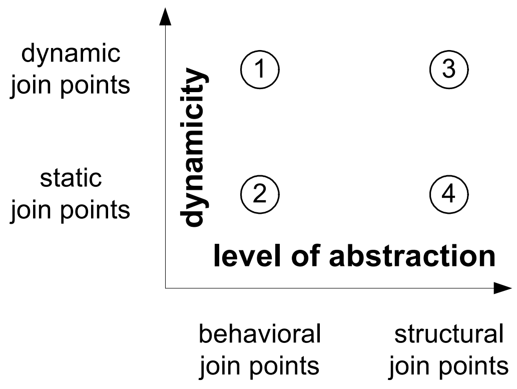


Figure 5-6. Orthogonal dimensions of join point models.

Both orthogonal dimensions are illustrated in Figure 5-6. On the one hand there is the axis representing the criteria of dynamicity of join points separating static and dynamic join points. The other axis represents the level of abstraction separating behavioral and structural join points. As a result, there are four models of join points in aspect-oriented systems:

[52] Obviously, the decomposition of the base application in Sally shares this view on static join points (see section 3.4.1 for further discussion).

- The **dynamic and behavioral join point**,
- The **static and behavioral join point**,
- The **dynamic and structural join point**, and
- The **static and structural join point**.

In the dynamic and behavioral join point model, an application is decomposed into a number of behavioral elements (like for example method calls in the (dynamic) control flow of the application). In the static and behavioral joint point model the application is decomposed into its static elements (e.g. method calls in the syntax tree). Each node in the resulting graph that contributes at runtime to the application's behavior represents a join point. The model of dynamic and structural join points decomposes the application at runtime into its structural elements (e.g. objects), but also dynamic types where each of these elements represents a join point. The model of static and structural join points decomposes an application before runtime into its structural elements that represent join points.

This thesis uses the term **abstract join point model** to refer to an aspect-oriented system's interpretation of the join point concept.

Abstract Join Point Model: The term abstract join point model describes the conceptual join point model underlying an aspect-oriented system. The abstract join point model describes the underlying join point concept with respect to its dynamicity and its level of abstraction.

The phrase "system A has a static join point model" describes that the system provides only static join points (which might be structural or behavioral). The phrase "system B has a structural join point model" describes that the system provides only structural join points (which might be static or dynamic).

However, a system's abstract join point model does not necessarily need to be a single point in the matrix of Figure 5-6. It can combine different kinds of join points. For example, it is possible that an aspect-oriented system provides static structural join points (like class definitions) and dynamic behavioral join points (like method calls) at the same time. This thesis refers to a single point in the 2x2 matrix as an **abstract join point model element**.

5.3.4 Concrete Join Point Models

The previous sections introduced different kinds of join points abstracted over concrete programming language constructs. For example, the term behavioral join point simply states that a join point is an encapsulated element that represents parts of the application's behavior. However, it does not automatically mean that two aspect-oriented systems are equal because they are based on the same join point model (e.g., behavioral join points). There is still a large variety of what elements potentially represent join points. What concrete join point model is to be chosen depends not only on the design of the aspect-oriented system, but also on base language's design. Consequently, different aspect-oriented systems can be based on behavioral join points and can still vary considerably.

Aspect-oriented systems based on the same programming language and the same join point model possibly provide different kinds of join points. For example one aspect-oriented system based on behavioral join points possibly provides just method calls as behavioral join points (i.e., those elements in the AST representing method calls) while another system potentially provides operators as join points (i.e., nodes in the AST representing the application of an operator).

If two aspect-oriented systems are based on different programming languages they can provide different join points because of the different abstractions and mechanisms provided by the underlying languages. This is obvious in case the underlying languages are based on different paradigms because language constructs of one language do not necessarily have a correspondence in the other language. For example, an aspect-oriented system based on Java (like AspectJ) can provide different join points than an aspect-oriented system based on C (like AspectC [CKFS01]) since both languages are based on different paradigms. In Java, method definitions represent static structural building blocks that can be provided as structural join points. Objects represent dynamic building blocks that can be provided as structural join points as well. In C, function pointers are dynamic structural building blocks that can be provided as structural join points.

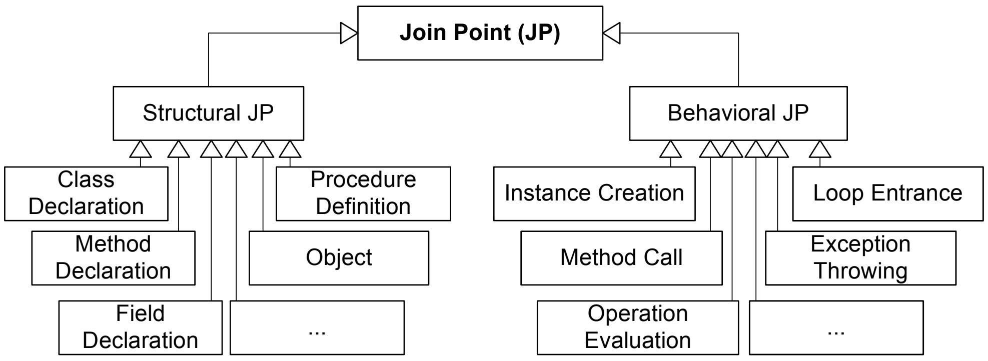


Figure 5-7. Join points classes.

Even if two languages are based on the same paradigm they can provide different abstractions and mechanisms. For example, an aspect-oriented system based on C++ could provide **operator overloading** as structural join points. Since Java does not permit to overload operators, such constructs cannot be provided as join points. Another example from the Java-C++ world (up to Java 1.5) are templates in C++, which are potentially structural join points but which do not have any correspondence in Java. In the same way, inner classes in Java can represent structural join points. Due to the absence of such constructs in Smalltalk, aspect-oriented systems for Smalltalk cannot provide such join points.

The elements that represent join points can be classified in all known aspect-oriented systems by a small number of classes. This thesis uses the term **join point class**[53] to

[53] In [WKD02] the term **kind of join point** is used. This thesis uses the term "class" instead of "kind". From our point of view the term "class" is more intuitive because of its use in the object-oriented literature.

describe a particular classification for a number of join points. Figure 5-7 illustrates different classes of join points. An example of a class of join points provided by an aspect-oriented system is **method call** or **instance creation,** as well as **class** or **method declaration**. A join point class can be directly extracted from the base language's syntax and semantics: Method call join points can be provided because the underlying language provides the method call construct (and corresponding method definitions), field assignments can be provided because the underlying language provides the field construct. Thus, the programming language itself already describes the possible set of join points.

This thesis refers to the set of join point classes as **the concrete join point model**.

Concrete Join Point Model: The term concrete join point model describes all classes of join points provided by an aspect-oriented system grouped into the dimensions defined by the abstract join point model.

For example, a simple concrete join point model for a Java-based aspect-oriented system could consist only of the static and behavioral join point class "Method Call", where method calls (i.e., the syntactical elements that represent method calls) can be selected and adapted by the aspect-oriented system.

5.3.5 Incomplete and Complete Coverage

The abstract and concrete join point models provide a first foundation for the comparison of aspect-oriented systems. Two aspect-oriented systems can be compared by comparing their abstract and concrete join point models. However, such a comparison is problematic if the comparison turns out that the abstract and concrete models are not equal because the consequences of "being not equal" are not clear.

Concrete join point models can be characterized by how they cover the base language. For example, if the base language provides only class definitions, method definitions, and field definitions as structural elements (this corresponds for example to **Featherweight Java** [IPW99] which is a small subset of the Java language), an aspect-oriented system's concrete join point model can consist with respect to static and structural join points of at most all those three classes. If the (class-based) base language provides only object creation, field assignments, field accesses, method invocations, and if-constructs as expressions, an aspect-oriented system can provide at most five static and behavioral join point classes.

Obviously, if an aspect-oriented system covers all elements from the base language as join points, it is "more complete" than systems that do not cover all elements from the base language. Furthermore, an aspect-oriented system that extends an existing system by simply adding a new join point class can be considered to be more complete than the extended system. For example, [Pras03] proposes an extension of AspectJ that adds typecasts as a new join point class to AspectJ. Consequently, the extension includes all join point classes the original system includes and provides an additional join point class. Hence, the new system has a higher coverage of the base language than the original system.

Consequently, for comparing aspect-oriented systems the **completeness** of the underlying join point models is a valid qualitative criterion. The term coverage and completeness can be applied to each single element in the 2x2 matrix described in

Figure 5-6 (i.e. to each abstract join point model element). For example, an aspect-oriented system's static join point model can cover all expressions of the underlying language, but cover only some parts of the structural elements provided by the base language. Consequently, the static behavioral part of the join point model completely covers the base language and the static structural part of the join point model covers the base language incompletely.

Complete Join Point Model: An aspect-oriented system's concrete join point model is complete with respect to an abstract join point model element, if there is a corresponding join point class for each construct of the base language.

Incomplete Join Point Model: An aspect-oriented system's concrete join point model is incomplete with respect to an abstract join point model element, if there is for this element at least one construct in the base language that is not covered by the concrete join point model.

Since the characteristics of completeness can be applied to all elements in the 2x2 matrix, the completeness represents a new design dimension – the level of coverage - for aspect-oriented systems as illustrated in Figure 5-8.

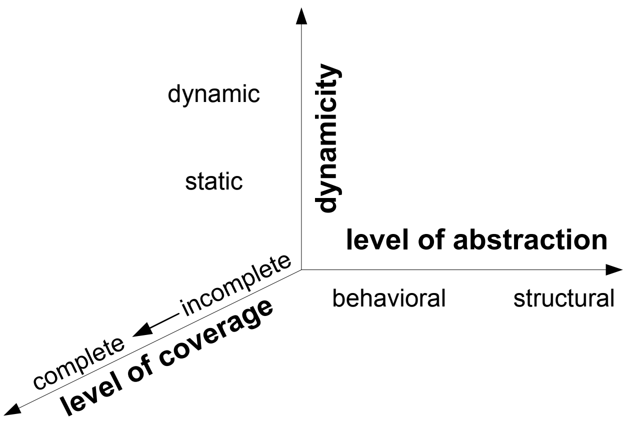


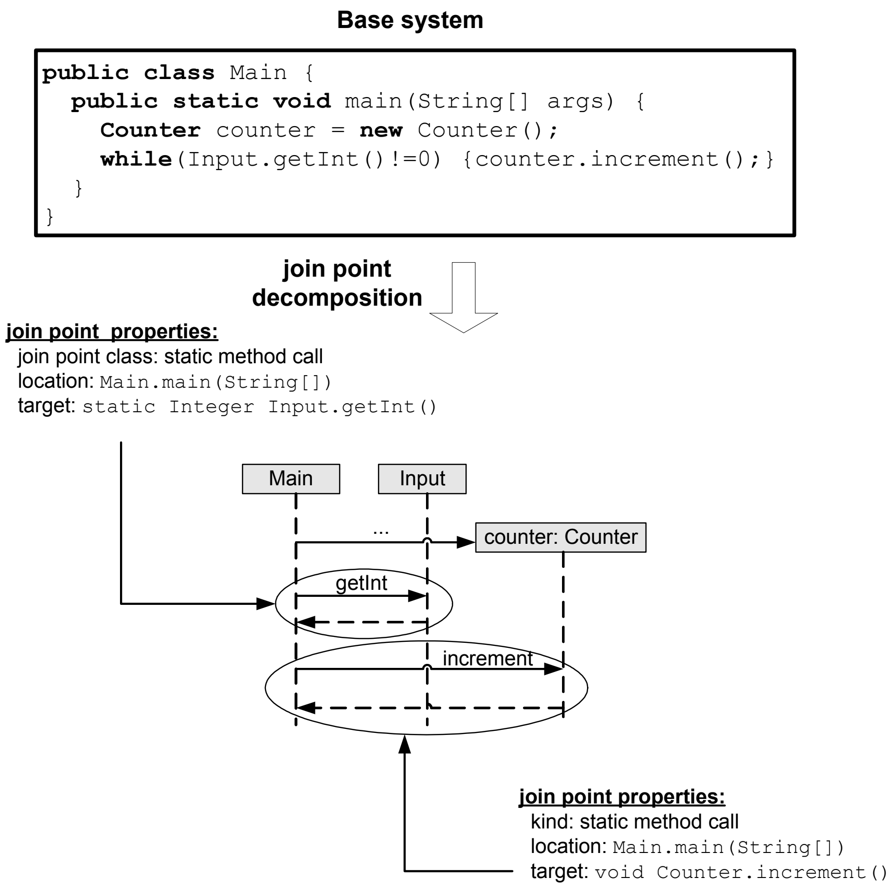
Figure 5-8. Three design dimensions of abstract join point models.

As discussed above, this third dimension differs noteworthy from the dynamicity and the level of abstraction. With respect to the dynamicity and the level of abstraction, this thesis identifies two different ways of how join point models can differ. With respect to the level of coverage, different aspect-oriented systems can vary between incomplete and complete coverage because there are systems that provide for a given base language more join point classes than other ones.

In principle, it is possible to determine a numerical value for the level of coverage. It is possible to divide the number of join point classes provided by a certain system with the number of expressions provided by the base language. For example, a base language possibly provides only object creation, field assignments, field accesses, method calls, and `if`-constructs as expressions. One aspect-oriented system provides only `if`-constructs as join points, while another one provides only method calls as join points. Both systems cover the same "number" of join point classes (which means 1/5 coverage). However, such a number is almost meaningless: Whether or not a certain aspect-oriented system is able to modularize a given crosscutting problem does not depend on the *number* of join point classes, but on the *appropriateness* of join point classes.

5.4 Join Point Selection

Because of the quantification property [FiFr00], the intention of aspects is (typically) not to adapt only a single join point per aspect. Also, aspects do not adapt every existing join point of the whole system. Instead, typically a (small) subset of all join points in the base-system are to be adapted by an aspect. When developers specify an aspect they have to specify at what join points the aspect contributes to the application. This selection is typically specified using declarative constructs of the underlying system.



Base system

```
public class Main {
  public static void main(String[] args) {
    Counter counter = new Counter();
    while(Input.getInt()!=0) {counter.increment();}
  }
}
```

join point properties:
join point class: static method call
location: `Main.main(String[])`
target: `static Integer Input.getInt()`

join point properties:
kind: static method call
location: `Main.main(String[])`
target: `void Counter.increment()`

Figure 5-9. Join point properties (subset) in a sample AspectJ application.

In [KHH+01] the term **pointcut** is used to specify such a selection of join points or the language constructs for specifying such a selection. The term **pointcut language** is also frequently used (see for example [GyBr03] or [Hirs02]) to describe the language constructs for selecting join points. However, this thesis uses the term **join point selection** to describe the selection of join points specified by the developer, and the term **join point selection language** (JPSL) to describe the constructs of the aspect-oriented system that permit developers to specify a join point selection. The reason for avoiding the term pointcut is twofold. First, the term pointcut is closely related to AspectJ and is often not used in other aspect-oriented systems. For example, the original paper on aspect-oriented programming [KLM+97] does not use the term pointcut. Second, the use of pointcut language as a synonym for join point selection in a sense as used by AspectJ is slightly misleading because in AspectJ the target of an introduction also represents a join point because new methods, interfaces, or superclasses can be added to an existing class. However, the target of an introduction in AspectJ is not specified using pointcuts (see section 2.2.1).

Join Point Selection Language (JPSL): The term join point selection language describes all language constructs provided by an aspect-oriented system in order to specify the selection of join points.

In order to select join points, each join point class must have a number of distinguishing characteristics in order to discriminate one join point from another. For example, if an aspect-oriented system's concrete join point model provides (static and behavioral) method calls and field accesses as join points, it is desirable from the developer's perspective to distinguish in general between method calls and field access join points, as well as between specific method calls and field accesses. Aspect-oriented systems provide for each join point class a number of predicates or properties the join point selection language's constructs refer to. Developers use these language constructs to select those join points which are to be adapted by certain aspects.

Figure 5-9 illustrates a number of join point properties in AspectJ for method call join points from the example introduced in section 5.3.1. Each method call join point has the property *location* that contains the signature of the method containing the method call and the property *target* that contains the signature of the method which is about to be invoked.

The properties provided by the aspect-oriented system for a certain join point depend on the design of the aspect-oriented system (i.e. the design of the join point selection language) itself, and also on the available information of the elements in the application. For example, if the underlying programming language is (statically or dynamically) typed, typing information can be used to provide properties for certain join points (e.g. method calls). If the programming language (or the underlying IDE) permits to annotate certain elements, such annotations can be used to provide corresponding properties for these elements[54]. In general, the more information from the application is available the more properties can be potentially assigned to a certain join point; i.e. a statically typed base language permits in general to provide more information for join points than an untyped base language.

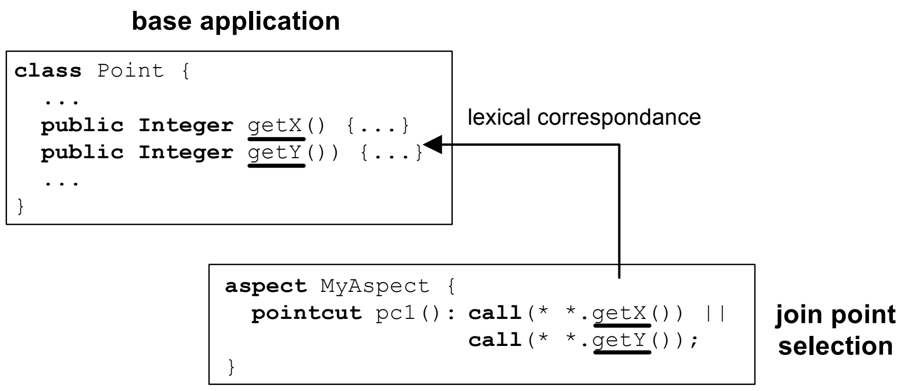
Join point property: A join point property is a join point's **characteristic** which is provided by the aspect-oriented system to permit developers to select join points

[54] For example, in [BCDM02] such annotations (which are not directly supported by the language, but by the IDE) are used to determine certain places in the code.

according to such characteristic[55]. A join point property has a **name** which represents the kind of information underlying the property and a **value** which describes the information itself.

Different aspect-oriented systems differ widely with respect to how they permit to select join points. For example, AspectJ provides a number of pointcut designators. They can be combined with operators to select join points to be adapted by an aspect. AspectJ permits also to address classes to perform introductions. Hyper/J permits to select classes and operations of a certain concern to be composed with other concerns (see section 2.3). In the same way, AspectS permits to select join points using join point descriptors (see section 2.4). Also, different aspect-oriented systems with the same join point model potentially have different kinds of join point selection languages. For example, the (abstract and concrete) join point model of AspectJ hardly changed since version 0.6. Nevertheless the join point selection language (i.e. the pointcut language in AspectJ terminology) increased substantially by introducing the + operator, for example.

The way that join points are selected is probably the most essential part of aspect-oriented systems. Hence, to make different aspect-oriented systems comparable it is necessary to study the design of join point selection languages and to understand the different design possibilities, i.e. the different dimensions of join point selection language design. For the development of new aspect-oriented systems, such design dimensions illustrate the different design alternatives for selection languages.



```
class Point {
  ...
  public Integer getX() {...}
  public Integer getY()) {...}
  ...
}
```

```
aspect MyAspect {
  pointcut pc1(): call(* *.getX()) ||
                  call(* *.getY());
}
```

Figure 5-10. Lexical selection of join points in AspectJ.

Since the main goal of the join point selection language is to select exactly those join points that need to be adapted by aspects, the **expressiveness** of join point selection languages is essential (the term *richness* is also frequently used instead of expressiveness, see [RaSu03]). Additional essential features of JPSLs are whether the JPSL provides constructs that permit to specify **reusable join point selections**, i.e. join point selections that can be used by a number of different aspects. Furthermore, a topic of interest is whether the JPSL permits to specify join point selections that are **stable** against changes in the base application (cf. for example [HOU03]).

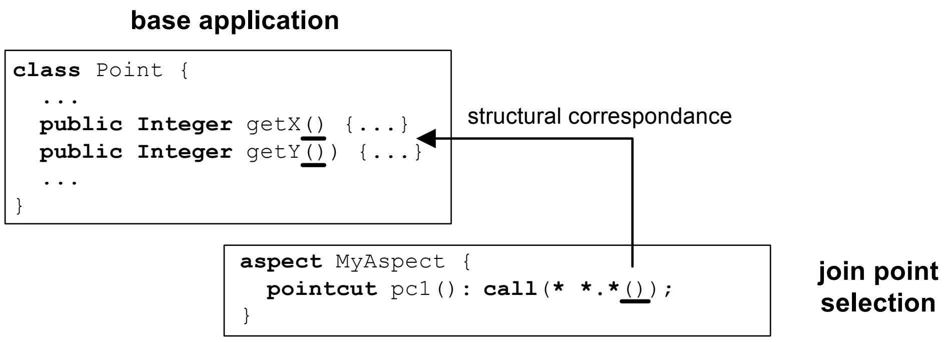
[55] Other authors [GBN+03, GyBr03] use the term **predicate** to describe the same concept. The reason for using the term property is motivated by its use in the object-oriented literature.

This section introduces design dimensions of join point selection languages. In the next section, some terms from the literature are discussed that already try to distinguish between different kinds of aspect-oriented systems and it is argued why such terms are not appropriate. Section 5.4.2 discusses the need for distinguishing between the properties provided for certain join points and addressing constructs that refer to such properties. Section 5.4.3 introduces different design dimensions of join point properties, section 5.4.4 introduces design dimensions of join point addressing constructs. Section 5.4.5 reflects on the relationship between properties and addressing.

5.4.1 Lexical, Property-based, and Semantic Crosscutting

There are already some works which distinguish between different ways join points can be selected. For example [LLM99] introduces the term **lexical crosscutting** to describe the characteristic of aspect-oriented languages that *the join points [...] consist of names that appear in the implementation [...]*. The intention of the term lexical crosscutting is to describe the dependency of a join point selection and the syntax of the underlying application. The term **property-based crosscutting** ([KHH+00, GyBr03]) is frequently used to describe the same situation.

Figure 5-10 illustrates the meaning of the term lexical crosscutting based on a join point selection in AspectJ. The aspect `MyAspect` selects those method call join points whose called methods have either the name `getX` or the name `getY`. This kind of join point selection is lexical because the addressed join points refer to a method definition with a certain name (`getX` or `getY`) that needs to be specified in the selection (i.e. in the pointcut definition): There is a **lexical correspondence** [HOU03] between some elements in the join point selection and elements in the application's source code. Because of this characteristic of the join point selection the authors in [LLM99] argue that AspectJ is based on lexical crosscutting. However, this view on join point selection is quite simplified and will be discussed in more detail below.



```
class Point {
  ...
  public Integer getX() {...}
  public Integer getY()) {...}
  ...
}
```

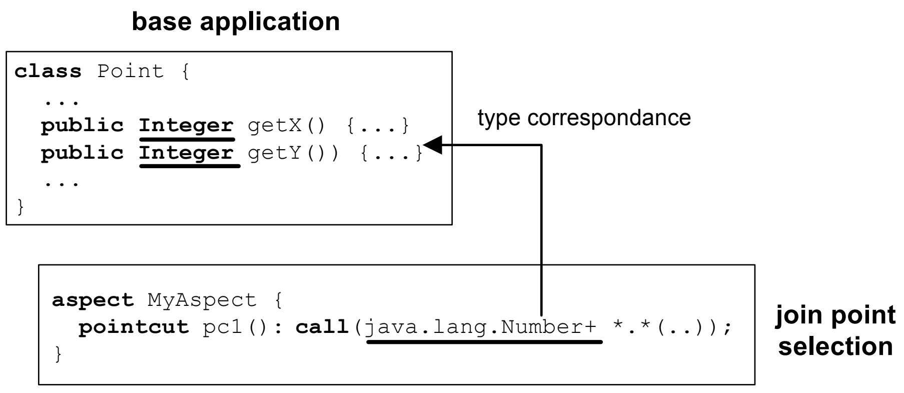
```
aspect MyAspect {
  pointcut pc1(): call(* *.*());
}
```

Figure 5-11. Non-lexical selection of join points in AspectJ based on structural correspondances.

In fact, AspectJ provides different ways to specify the selection of join points that are not necessarily lexical. Figure 5-11 illustrates the selection of join points in AspectJ where the join point selection does not contain any lexical element that is also contained in the base application. The method call join points are selected independent of the names and return types of the called methods: Method call join points are selected because the called method does not have any parameters (i.e. the number of parameters is zero). On a more abstract level, this means that join points are selected because of a **structural correspondence** of a parameter list in the base application and in the pointcut specification.

Obviously, this kind of selection has a different quality than the previous one because a selection based on a structural correspondence is more stable with respect to simple syntactic changes in the base program than a selection based on a lexical correspondence. For example, applying the **rename method refactoring** [Fowl99], i.e. changing a method's name, does not have any effect on the selected join points. Hence, the selection is stable with respect to changes of method names (see also [HOU03]).

Figure 5-12 illustrates a different way of selecting join points in AspectJ. The return types of the join points to be selected are specified as `Number+`, which describes all subtypes of `Number` (see also section 2.2.2). According to the definition of lexical crosscutting taken from [LLM99] this kind of selection is lexical, because the type identifier `Number` appears in the join point selection and this type identifier also appears in the base application (or more precisely in the class libraries provided by Java). However, the quality of the selection obviously differs from the previous selections because the join points are selected based on some characteristics of the underlying type system: The identifier in the selection does not correspond to the identifier of the return type of the selected join point. Instead, the type name corresponds to a different type name.



```
class Point {
  ...
  public Integer getX() {...}
  public Integer getY()) {...}
  ...
}
```

```
aspect MyAspect {
  pointcut pc1(): call(java.lang.Number+ *.*(..));
}
```

Figure 5-12. Non-lexical selection of join points in AspectJ based on type correspondances.

Based on the discussion above it can be concluded that the term "lexical crosscutting" is not well defined and not precise enough to distinguish between different kinds of join point selections.

The term **semantic-based crosscutting** as introduced in [KHH+00] is also frequently used to describe a certain way a JPSL permits developers to specify certain join points. The intention of semantic-based crosscutting is to select join points not because of syntactical properties, but because of the semantics of the underlying programming language. One example for such semantic join point selection is already given in Figure 5-12: The join points are not selected due to their syntactical properties, but because of some properties of the underlying type system. Another typical example for semantic join point selection is the use of the `cflow` construct that permits to select join points according to their position in the control flow of an application.

The problem with the term semantic-based crosscutting is that the term is not precisely defined and it is not clear how it relates to lexical crosscutting. As argued above, the selection of a method call join point due to the return type is somehow semantic because the type system is not part of the base program but of the base program's language. On the other hand, the type `Number` is also addressed in a lexical way. Hence, this kind of join point selection is also in some sense a lexical crosscutting. The same is true for the `cflow` construct in AspectJ: On the one hand the join point occurring in a control flow can in general not be directly computed from the application's syntax. However, in AspectJ the `cflow` pointcut designator has a pointcut as a parameter. This pointcut can also be addressed lexically (e.g. by selecting a previous method call by the called method's name).

5.4.2 Distinction of Join Point Encoding and Join Point Addressing

The terms lexical crosscutting and semantic-based crosscutting are confusing because it is not clear what characteristics of the JPSL they address.

In fact, there are many different ways a JPSL possibly permits to select certain join points. When designing a JPSL for a given join point model (and a given underlying system), there are two main questions that need to be answered:

- What properties should the JPSL provide for each join point?
- How can these properties be addressed and combined using the JPSL?

The first question simply addresses the problem of what distinguishing characteristics belong to a join point. For example, it is possible to design an aspect-oriented system in a way that the only available property of a method definition join point is the method's name (like the Hyper/J approach). Another approach (similar to the AspectJ approach) is to provide also a parameter type list property that permits to select methods because of their declared parameter types.

In general, the more properties are available for each join point the more possibilities developers have in order to select exactly the required join points. If a method is only selectable because of its method name, there may be the problem that developers are forced either to select all methods having the same method name or to select none. On the other hand, the more properties are available for a join point the effort of selecting a join point potentially increases. If the developer has to specify the return type, the method name, and the list of parameter types for selecting a single method the effort for specifying such a selection is obviously higher than specifying only the method name.

The second question addresses the language constructs provided by the JPSL that permit the developer to select certain join points. Similar to the questions of what properties are available, there are many different possibilities of what language constructs can be provided. For example, a simple way of selecting a method definition (whereby the underlying system provides method names as join point properties) is to specify the method name (this kind of selection corresponds to the previous example of lexical crosscutting in Figure 5-10). However, as illustrated in Figure 5-12 it is also possible that the join point selection language permits to specify not only a value of a join point property like the name, but also how the property is related to other properties. In Figure 5-12, join points are selected due to some type relationships of type `Number`.

Obviously, the design decision underlying what join point properties are available for each join point and the decision of how join point properties are used by the join point selection language are independent of each other. For example, the previous paragraphs discussed the use of a method name as a join point property for the method definition join point class and discussed different ways of addressing such a property. Furthermore, some examples show that the join point selection language possibly evolves by extending the join point properties of certain join point classes, or by extending the means to address join point properties, or both. For example, in [NCT04] a join point selection language is proposed that includes AspectJ's JPSL, but provides an additional join point property that permits to select join points based on the host in which they occur.

Consequently, this thesis distinguishes between characteristics of join point properties and means to address such properties. Both are discussed in the following in separate sections.

5.4.3 Join Point Properties

Different aspect-oriented systems provide different properties for join points. For example, in AspectJ method call join points can be selected due to the called methods' signature consisting of the method's name, return type, and parameter types. Hence, *signature* is a property of method call join points in AspectJ, and a specific signature like `void m()` represents the value of the signature property for a certain join point. In Hyper/J, a method definition join point has an *operation name* property which can be used by the join point selection language in order to select the corresponding join points. Consequently, operation name is a join point property and a concrete method name is the property's value.

Furthermore, join point properties are closely related to the underlying base language. For example, properties like return types or parameter types cannot be join point properties if the underlying language is untyped. Smalltalk is an example for such a language. Furthermore, method names in languages like C++ and Java substantially differ from selectors in Smalltalk: A selector in Smalltalk is structured in the sense that the number of parameters of a certain method can be directly derived from the selector. Consequently, a selector property for a Smalltalk based system contains more information about a method than a method name in a Java or C++ based system.

Since the properties potentially available for each join point (and also the underlying join point classes) differ in most aspect-oriented systems, it is difficult to compare them because of the available properties for each join point class. For the design of new aspect-oriented systems it is problematic that join point properties are closely related to the underlying base language – in case a certain element is not available in a based system (like static types) such properties cannot be provided.

However, it is observable that different join point properties have a number of different characteristics. For example, pointcut designators `this`, `target` and `args` in AspectJ permit to select join points based on dynamic types, while pointcut designators like `call` and `execution` permit to select join points based on static types, hence static as well as dynamic types represent join point properties in AspectJ.

Furthermore, the `cflow` construct in AspectJ permits to select join points based on some join points that are already reached in the current control flow. Also, [MaKa03] introduces the selection of join points based on some characteristics of the dataflow. Consequently, some join point properties that refer to the dataflow need to be available at certain join points.

Obviously, the described join point properties have different characteristics. Hence, similar to the previous sections, this thesis identifies a number of design dimensions of join point properties. Instead of describing a number of known join point properties for a given join point class, the design dimensions describe the general characteristics of these join point properties.

5.4.3.1 Static and dynamic properties

A fundamental difference between join point properties is their relationship to runtime data in the base system. For example, in Hyper/J method definition join points have names that can be used for selecting and adapting join points. Sally (see section 3.4) permits to access return types, method names and parameter types for method definition join points.

In AspectJ, there are also join point properties whose values appear just at runtime. For example, pointcut designators like `this`, `target` and `args` refer to dynamic types of objects that participate in the corresponding join points. In the same way, pointcut designators like `cflow` in AspectJ as well as the `dflow` pointcut designator as proposed in [MaKa03] refer to runtime specific data in the base application.

One difference between join point properties is their occurrence with respect to their dynamicity: A property like `this` in AspectJ is based on dynamic system information (or *some portion of the program state* [WKD02]), a property like a parameter of `declaredMethod` in Sally is based on static system information (i.e. information that is available because of a static analysis of the base application). Hence, join point properties can be distinguished with respect to their **dynamicity**. This thesis distinguishes between **static properties** and **dynamic properties**.

Static join point property: A join point property is static if the property can be directly derived from the base system's code.

Dynamic join point property: A join point property is dynamic if runtime-specific information is needed in order to provide the corresponding property.

An example of a static join point property is the method name of a (static or dynamic) method call join point: The method name can be directly derived by analyzing the source code (i.e. the node in the syntax tree representing the method call). In the same way static type information (as known from programming languages like Java), e.g. the static type of a method call join point or the declared return type of a method definition join point, represent static join point properties.

Section 5.3.1 distinguished between static and dynamic join points. Hence, it seems obvious that join point properties can be distinguished in the same way. Nevertheless, there is a difference between the dynamicity of join point models and the dynamicity of join point properties: A join point describes a base system's element that can be selected and adapted by the language constructs provided by an aspect-oriented system. A join point property describes a join point's characteristic that a JPSL's construct refers to in order to select the desired join point. Although a join point is dynamic, it still can have some static properties. For example, dynamic method call join points in AspectJ have static join point properties like the method's name (the method name can be directly derived from the underlying code).

5.4.3.2 Direct and abstract correspondence

The characteristics of static join point properties is that they can be directly derived from the base system's code. Obviously, a declared return type of a method declaration join point represents a static property because the return type (or the name of the return type) directly appears in the source code. Such properties are prerequisites for the term **lexical crosscutting** as introduced in [LLM99] which says that *the join points [...] consist of names that appear in the implementation [...].*

A property extracted from the base system's sources does not necessarily need to have a direct correspondence in the underlying source. Figure 5-13 illustrates some static join point properties of a method definition join point in a hypothetical aspect-oriented system. The properties `name` (which represents the name of the method), `numParams` (which represents the number of parameters), and `retType` (which represents the return type) can be directly derived from the application's syntax.

However, there is a difference between `name`, `retType` and `numParams`: The values of `name` and `retType` have a direct correspondence in the application's syntax, i.e., there are nodes in the application's syntax tree having the same value as the corresponding properties. This is not true for the value of property `numParams`: The value represents a statement over a part of the syntax tree: The aspect-oriented system needs to do some computation in order to provide such a property.

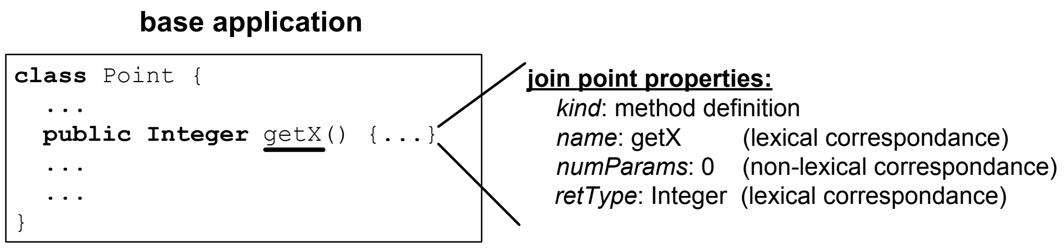

base application
class Point {
...
public Integer getX() {...}
...
...
}
join point properties:
kind: method definition
name: getX (lexical correspondance)
numParams: 0 (non-lexical correspondance)
retType: Integer (lexical correspondance)

Figure 5-13. Static join point properties with and without direct correspondence.

In order to distinguish between those different kinds of properties this thesis identifies the dimension of **directness of property correspondence** and distinguishes between **direct** and **abstract correspondence**.

Direct property correspondence: A property has a direct correspondance if the property can be directly derived from the data available at the corresponding join point.

Abstract property correspondence: A property has an abstract correspondance if the property does not directly represent some data available at the corresponding join point but needs some additional computation by the aspect-oriented system in order to provide the corresponding property.

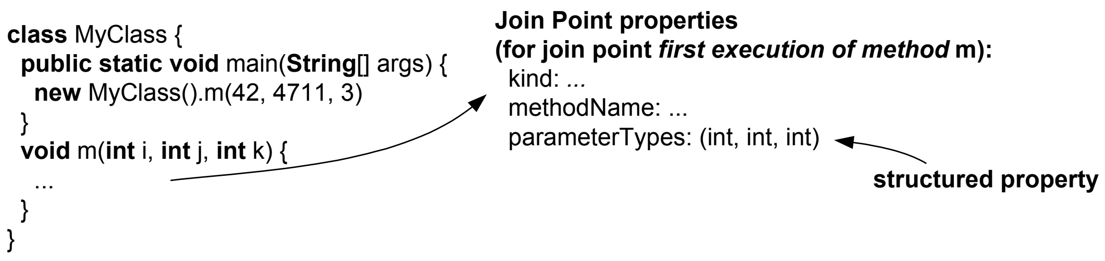
Examples of direct property correspondence are the previously discussed properties `retType` as well as the `name`. They are nodes in the base program's syntax that directly represent the value. On the other hand, the property `numParams` has no lexical value correspondence, i.e. it has an abstract value correspondence[56].

The directness of property correspondence is not limited to static join point properties. For a language that supports a variable number of parameters[57] or optional parameters, there is the possibility to provide a property for dynamic method call join points that simply states how many arguments are being passed.

The term abstract property correspondence simply states that the property is not directly available at the corresponding join point. However, there is a large variety of "being not directly available". The example above showed that "number of parameters" represents an abstract property because the aspect-oriented system needs to perform some trivial computation on the syntax tree. However, the computation performed by the aspect-oriented system might be even more complex. For example, the system could provide information about method definition join points that state whether the execution of a method potentially changes the behavior of the corresponding object. Such a property might be helpful for handling implementations of the observer pattern [GHJV95] using aspect-oriented techniques.

5.4.3.3 Atomic and structured properties

Properties like method names for method call join points are quite easily to understand because the property's value is atomic: The value itself is not structured and addressing the value is quite easy. Such kinds of properties can be found in almost all aspect-oriented system.



```
class MyClass {
  public static void main(String[] args) {
    new MyClass().m(42, 4711, 3)
  }
  void m(int i, int j, int k) {
    ...
  }
}
```

Join Point properties
(for join point *first execution of method* m):
kind: ...
methodName: ...
parameterTypes: (int, int, int)
structured property

Figure 5-14. Structured Property parameterTypes having an ordered list as the underlying data structure.

However, it is also possible that a single property is structured in a way that it represents complex data. For example, in AspectJ the formal parameters of method call join points or method execution join points are represented by a list the JPSL can refer to. Figure 5-14 illustrates such a structured property `parameterTypes` for a method call join point where the underlying data structure is an ordered list and the order

56 Systems like AspectC++ [SpGS02] provide the number of parameters explicitly in a certain variable which can be referred to within the advice.

57 A number of languages provide such a features (like e.g. TCL [Oster94]). A similar feature is also included in Java since version 1.5 (cf. [Sun04c]).

corresponds to the order of parameters of the method call. In the same way, Sally and the system described in [GyBr03] provide parameter lists as join points. Considering this, this thesis identifies the **structuredness of join point properties** as a design dimension and distinguishes between **atomic join point properties** and **structured join point properties**.

> **Atomic join point property:** A join point property is atomic if the property's value is atomic, i.e. it consists of a single unit.
>
> **Structured join point property:** A join point property is structured if its value (potentially) represents more than one atomic value.

A typical example of an atomic join point property is the method name property of a method call join point: It simply consists of one atomic value. In contrast to that, the method call join point's parameter list property is structured because the value of this property has a list as a data structure. Another example of a structured join point property in AspectJ is the encoding of the control flow property: The control flow is a structured join point property because all previous method calls within the control flow are logically part of a stack[58].

A characteristic of a structured property is that its join points are represented by one single join point class and that the structure of the underlying property potentially varies among different join points. For example, if the base language provides an infinite number of parameters in method definitions and an aspect-oriented system wants to provide a (static) method call join point and a property that represents the parameters there is a need to have a single property that represents a parameter list of arbitrary length. In case the underlying language provides a finite length of parameters, it is theoretically possible to provide a property for each individual parameter. For example, Java permits at most 255 parameters for methods. Consequently, it would be possible in Java to provide 255 join point properties representing each a parameter. In Squeak, the number of parameters is restricted to twelve. Hence, it would be possible to provide twelve properties, each representing a parameter. However, from the point of usability such a large number of properties seem to be rather undesirable because it becomes hard to handle a large amount of properties.

The distinction between atomic and structured properties becomes relevant in the context of how an aspect-oriented system permits the user to select a certain join point: If all existing join point properties provided by the system are atomic the JPSL potentially needs to provide rather simple means to address a property. Structured properties permit theoretically more advanced means to address a certain join point, because not only an atomic value is relevant for the selection. However, the distinction of the structure becomes less important, if the structure of the value of a certain property is fix. For example, a signature pattern in AspectJ that represents the values of call properties always has a fix structure consisting of a return type, a class identifier, and a method name. Conceptually it makes no difference if the corresponding property is

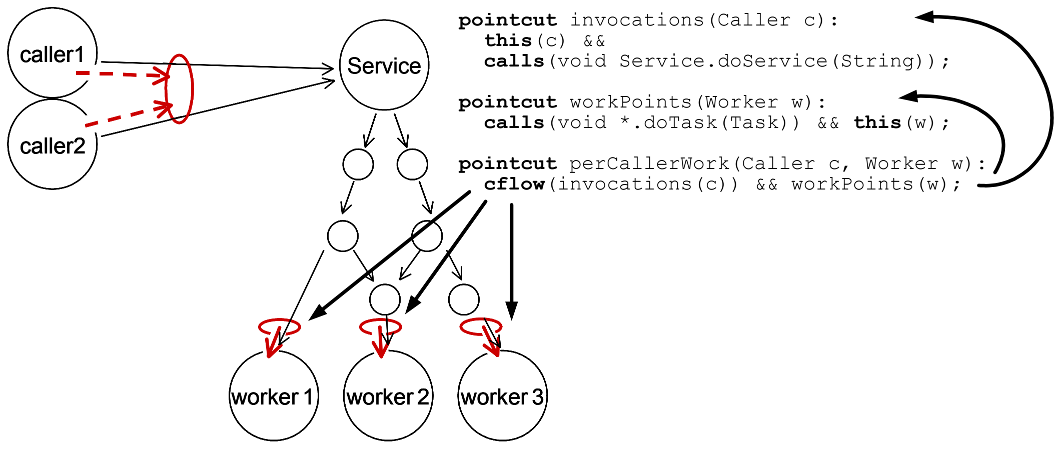
[58] However, the implementation of the `cflow` construct slightly differs from this conceptual point of view (cf. [HiHu04, p. 29]). Since such an implementation detail is out of the scope of this discussion it is neglected here.

structured or if the system provides three different properties. This is not true for the parameter property: The number of elements varies for different methods.

5.4.3.4 Local and non-local join point properties

Intuitively, for a given kind of join point class it seems clear what distinguishing characteristics can be provided. For example, it seems clear that a method name property can be provided for method call join points (which corresponds to all known aspect-oriented systems). A method name can be directly derived from (static and behavioral) method call join points because a method name is specified by the syntactic elements that represent the method call: A method call as defined in the syntax of most base languages includes the name (or the selector in Smalltalk) of the method being invoked. For the same reason it seems intuitively clear that formal parameter types can be provided for method definition join points.

Also it seems intuitively clear that a dynamic method call join point can be easily provided with the elements belonging to the corresponding method call. The corresponding call consists of a calling object, a called object, a message name, and a number of actual parameters. All these elements are being used by the underlying runtime system in order to compute the target method being invoked[59]. Hence, it seems intuitively clear that such elements represent reasonable join point properties. Furthermore, state information of all objects participating in the method call can be potentially used as distinguishing properties for such a join point. For example, if it is desirable to provide means to selected method call join points only in cases where one of the parameters is `null`, then corresponding join point properties can be provided for such calls.



```
pointcut invocations(Caller c):
  this(c) &&
  calls(void Service.doService(String));

pointcut workPoints(Worker w):
  calls(void *.doTask(Task)) && this(w);

pointcut perCallerWork(Caller c, Worker w):
  cflow(invocations(c)) && workPoints(w);
```

Figure 5-15. Wormhole pattern in AspectJ (taken from [HKG+01] with minor modifications).

On the other hand, providing the potential caller types for a static method definition join point is not that intuitive and systems like Hyper/J do not provide a corresponding

[59] This is slightly simplified. In systems like Java or C++, that provide late binding and static multi-dispatching, the computation of the method being executed also depends on the static types of the parameters being passed.

property: In Hyper/J only the classes of callers can be specified within a bracket relationship. Also, a property that provides information about all classes referring to a static class definition join point is unusual and is not provided by systems like AspectJ or Hyper/J.

The reason for this lies in what information is available at each join point. The base language defines some kind of **local context** for each join point class[60]. For example, the source code of a method call has a number of elements (like fields as targets and parameters) in its syntax tree representation that can be directly used as join point properties. Such properties are local to the method call[61] because they are closely located to (or even direct subnodes of) the node representing the method call. In the same way, the objects participating in a dynamic method call represent some local context because such objects are being used by the underlying runtime system in order to invoke the corresponding method.

Aspect-oriented systems potentially provide join point properties that cannot be directly derived from the join point's local context. One example of such a non-local deducible property (for a dynamic join point) is the occurrence of an object of a certain type in the control flow before a certain method call is reached. This kind of selection criterion is the foundation of the **wormhole pattern** in AspectJ as explained in [Labb03, pp. 256]: *The wormhole makes the object in the caller place available to the methods in the called place without passing the object through the call stack.*

Figure 5-15 illustrates the wormhole pattern in AspectJ: Instances of `Caller` invoke the method `doService()` of an instance of `Service` which itself invokes (indirectly) the messages `doTask()` of a `Worker` instance. The selection of a `doTask()` join point (the dynamic method call) which occurs because of a previous `doService()` invocation is specified in the pointcut `perCallerWork`. Since the information about the calling object `c` used within `perCallerWork` is not derivable from the local context of the join point itself, the join point selection is based on non-local deducible information. Hence, there are properties underlying the corresponding selection of method calls which are non-local.

Based on the previous discussion this thesis identifies the **locality of the join point property** as a design dimensions of join point properties and distinguishes between **local properties** and **non-local properties**.

Local property: A property of a certain join point is local if the property can be derived from the local information available at the join point.

Non-local property: A property of a certain join point is non-local if the information needed in order to provide the corresponding property is not accessible from the local context of the join point.

[60] The term context is being used here in a different way than that of for example known from [GoRo89], where the term context refers to the call stack.

[61] The phrase **textual locality** as used in [KiMe05] can be applied here, too. Since such properties can be derived from the textural representation (join point shadow) of the underlying join point, such properties are local.

Information is accessible if it is reachable from a join point's context. For example, for a given class definition join point, the members of the class are reachable since it is possible to derive class members from a class (using introspective facilities of the underlying programming language like the Reflection API in Java). For a given dynamic method execution join point, the state of the object whose method is called is reachable: Since the object is known it is possible to access the object's state directly or to send additional messages to the object in order to find out the object's state.

However, the term local context is slightly subjective. In fact, this term can be interpreted in different ways for static as well as dynamic join points.

In the beginning of this section the term local context was motivated by an example of a static method call join point: A method name and the expressions that describe the called object and the actual parameters can be interpreted as part of the local context of the syntactical expression. However, although the underlying base language might be object-oriented the language's syntax definition does not necessarily need to have an explicit representation of method calls. Furthermore, a subnode in an application's syntax tree does not necessarily represents "local" information.

Figure 5-16 illustrates production rules of a context-free grammar `A` for a Java-like base language. The grammar includes the production rule *<methodCall>* that explicitly represents a method call. Consequently, method calls can be directly identified in the syntax tree of a given class definition.

```
<classDeclaration>    ⟶  "class" <identifier>
                         ["extends" <identifier>]
                         "{" <fieldDeclarations> | <methodDeclarations> "}"
<identifier>          ⟶  ...
<literal>             ⟶  ....
<fieldDeclarations>   ⟶  (<identifier> <identifier> ";" )*
<methodDeclarations>  ⟶  (<methodHeader> <block>)*
<methodHeader>        ⟶  <identifier> <identifier>"("
                         [(<identifier> <identifier>) ("," <identifier> <identifier>)*]
                         ")"
<block>               ⟶  "{" <expressionList> "}"
<expressionList>      ⟶  ( (<expression> | <retExpression> | <ifExpression>) ";")*
<ifExpression>        ⟶  "if" "("<expression>")" <block> [ "else" <block> ]
<retExpression>       ⟶  "return" <expression>
<expression>          ⟶  <assignment> | <methodCall> | <varExpr> |
                         <newExpr> | <literal> | <identifier>
<varExpr>             ⟶  <expression> "." <identifier>
<newExpr>             ⟶  "new" <identifier> <paramList>
<paramList>           ⟶  "(" [<expression> ("," <expression>)*] ")"
<assignment>          ⟶  <varExpr> "=" <expression>
<methodCall>          ⟶  <expression> "." <identifier> <paramList>
```

Figure 5-16. Grammar A with explicit production rule for method calls.

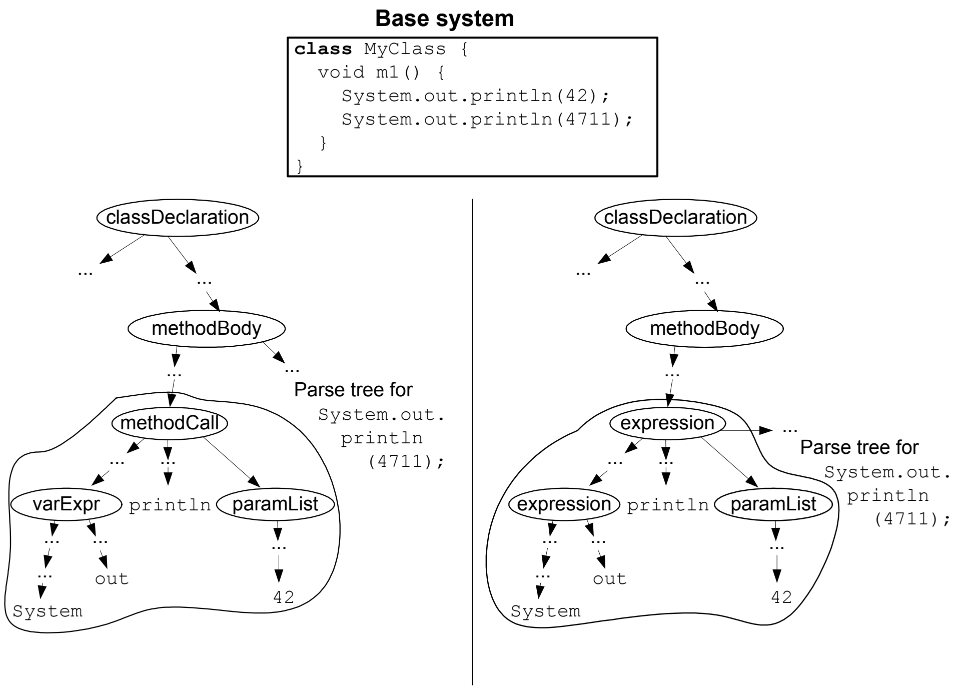
Figure 5-17 illustrates a grammar B which defines the same language as A but whose production rules slightly vary. Grammar B does not have an explicit production rule for method calls. Furthermore, the enumeration of statements (separated by the token "`;`") is defined in a different way: Instead of having a common node that enumerates all statements (which corresponds to the production rule *<expressionList>* in grammar A) each production rule representing expressions defines its following expression.

```
<classDeclaration>  ──►  ...
<...>               ──►  ...

<expressionList>    ──►  <expression> ";" | <retExpression> | <ifExpression>

<ifExpression>      ──►  "if" "("<expression>")" <block> [ "else" <block> ] ";"
                         [ <expressionList> | ε ]

<retExpression>     ──►  "return" <expression> ";" [ <expressionList> | ε ]

<expression>        ──►  (
                           <expression> "." <identifier> [ <paramList> ] |
                           "new" <identifier> <paramList> |
                           <varExpr> "=" <expression> |
                           <literal> |
                           <identifier>
                         ) [";" <expressionList> ]
```

Figure 5-17. Grammar B without explicit method call production.

Although both grammars define the same language, the syntax trees for a given application differ substantially. Figure 5-18 illustrates two derivation trees for a simple class definition that can be derived from the previous grammar definitions. Considering the first executable statement of method `m` for grammar A, the method call consists of the expression that represents the target expression `System.out`, the method name `println`, and the parameter list consisting of the integer literal `42`. In this example, the term local context can be interpreted as the whole sub-tree having the node representing the method call as its root. Consequently, it seems intuitively clear to interpret the class `System`, its field `out`, the method name `println` and the parameter `42` (which is an integer literal) as potential candidates for local join point properties. The situation is different for grammar B's derivation tree. First, the derivation tree does not contain a direct representation of method calls, because there is no explicit production rule for method calls. Second, the sub-tree that represents the method call also includes the parse-tree representation for the second statement (`System.out.println(4711)`). Obviously, it seems rather unintuitive to regard the second statements as part of the first method call join point's local context. Consequently, simply stating that the local context of a static join point can be directly derived from the syntax tree by its subnode relationships is not correct because a programming language's syntax can be defined by different equivalent grammars.

The term local context is slightly problematic for dynamic join points, too. For example, it seems to be intuitively clear to consider the objects that participate in a method call to be part of the method call join point's local context. However, a close look at a specific language reveals even more potential local join point properties. For example, in Java it seems clear to consider the called object as well as the actual parameters as part of the local context. Furthermore, it seems clear to consider information about previous methods from the call stack as being not part of the dynamic method call's local context, because Java does not directly provide a runtime representation of the call stack. However, in the dynamic execution of a Java application the stack trace is also available: It is possible to create an instance of `Throwable` (which represents a caught or uncaught exception) at runtime. By sending the message `printStackTrace` to such an instance call stack information are made explicit: All messages on the call stack are printed to a given stream. This call stack information has a pure textual representation. Hence, it is possible to determine whether a certain method is on the class stack, but it is not possible to determine what objects performed method invocations that lead to the current call stack's state.



Base system

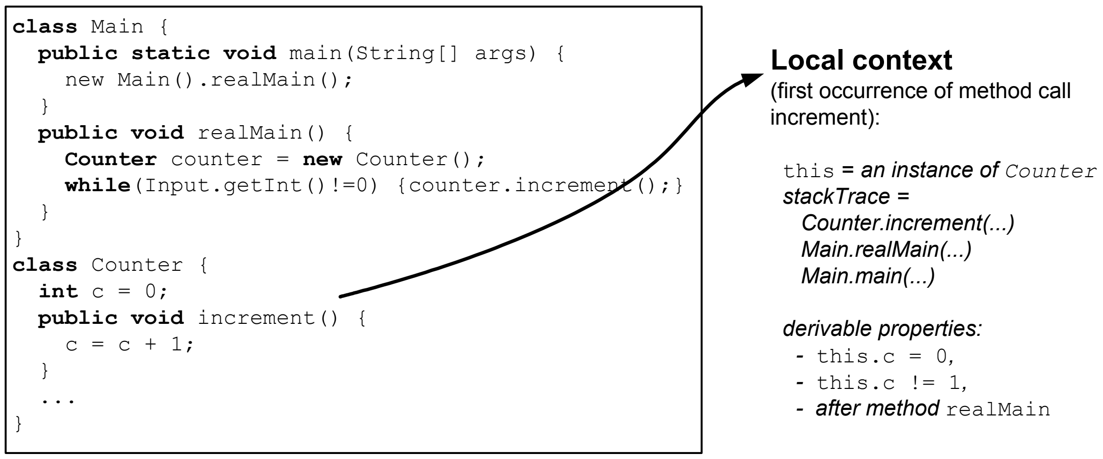
```
class MyClass {
  void m1() {
    System.out.println(42);
    System.out.println(4711);
  }
}
```

Figure 5-18. Derivation trees for a simple class definition for grammar A and B.

Figure 5-19 illustrates the local context from within a method `increment` inside a `Counter` object being executed for the first time in the application: From within the method `increment` it is possible to refer to the corresponding `Counter` object and to the stack trace leading to this method call. Since the method is executed, the stack trace consists of the method `increment`, `realMain` and `main`. Consequently, when the method `increment` is being invoked all elements of the stack trace are local deducible and represent candidates for join point properties. The object that belongs to the method calls (i.e., the instance of `Main`) cannot be deduced at runtime from the method call. Hence, the object does not belong to the join point's local context.

In systems such as Smalltalk, the local context at method calls differs widely from the local context at method calls in Java. In Smalltalk exists the special variable `thisContext` which makes the current call stack accessible. In contrast to Java, `thisContext` does not only make signatures on the call stack available but also the corresponding objects. Consequently, for a dynamic method execution join points such objects represent local deducible properties and properties provided by a Smalltalk-based aspect-oriented system that are related to such control-flow specific objects represent local join point properties.

Furthermore, Smalltalk dialects like Squeak or VisualWorks permit to access all instances of a class in the system. In Squeak, the class `Behavior` (which is a superclass of all `Class` instances) provides a method `allSubInstances` which returns all objects of the corresponding class and all subclasses. Hence, if such a message is sent to the class `Object` (which represents the superclass for all objects in the system), all objects are returned[62]. Since this method is available for all classes, and since all classes can be computed from the current image, all existing objects in the image including their state represent derivable information from a method call's local context[63].



```
class Main {
  public static void main(String[] args) {
    new Main().realMain();
  }
  public void realMain() {
    Counter counter = new Counter();
    while(Input.getInt()!=0) {counter.increment();}
  }
}
class Counter {
  int c = 0;
  public void increment() {
    c = c + 1;
  }
  ...
}
```

Local context
(first occurrence of method call increment):

`this` *= an instance of* `Counter`
stackTrace =
Counter.increment(...)
Main.realMain(...)
Main.main(...)

derivable properties:
- `this.c = 0,`
- `this.c != 1,`
- *after method* `realMain`

Figure 5-19. Local context of a method `increment` in `Counter` executed for the first time in the programming language Java.

The problem with such different views on locality is that they make it difficult to compare different aspect-oriented systems independently of the underlying base language. As argued above, even giving a precise definition of the term local context for the same base language for properties that can be derived from the syntax tree is hard because different grammars (with different production rules) potentially describe the same language although the resulting syntax trees differ widely. For aspect-oriented systems having different base languages, the term locality is problematic because

62 In VisualWorks the corresponding method is `allGeneralInstances`.

63 It is obvious that accessing all objects at a certain point in time is a highly time-consuming task and should be avoided whenever possible for reasons of performance. But since this section reflects on the characteristics of properties being provided by an aspect-oriented system, considerations of system performance are neglected.

different languages have different notions of locality. For languages that do not provide any call stack information at all, it is clear that information about previous messages are not local.

In general, there are two different goals for classifying join point properties with respect to their locality:

- First, it is desirable to determine for aspect-oriented systems in a language independent way whether they are able to handle a certain crosscutting problem. A typical question is for example whether a method call join point can be selected because the called object is referenced by another certain object. For this purpose it is desirable to have a common understanding of the term local context for different base languages. Consequently, the previously described differences between languages like Java or Smalltalk should not influence the understanding of the term local context.
- Second, it is desirable to determine for an aspect-oriented system what properties a join point provides in addition to the features of the underlying base language. This information is valuable if developers want to estimate the effort for providing a certain attribute. For this purpose, it is desirable to have a language-dependent understanding of the term local context.

In order to overcome the first problem, this thesis proposes to use a common meta model for language features like for example the UML meta model [OMG01]. Such a model represents a common abstraction for a number of languages and language constructs like class definitions or method calls. In order to overcome the second problem a join point property should additionally contain a description of whether the property is already provided by the underlying base language.

In principle, it is possible to provide a more fine-grained distinction of the join point property's locality. For example, it would be possible to define a distance between a (non-local) join point property and the property itself. Such distance could be useful for example for describing how many nodes exist between an element representing the join point and a node representing the join point property. However, this thesis considers the distinction between local and non-local join point properties to be sufficient.

5.4.3.5 Join point identity and shadow identity

A property can be used not only to decide whether or not a certain join point is to be selected. It can be used to compare different join points and to decide upon such a comparison whether or not a join point is to be selected. However, even the existence of a large number of join point properties does not guarantee that the appropriate join points are always selected. It turns out that there is often the requirement that static join points provide properties that are unique for such join points – properties that represent some identity for the corresponding join points.

Figure 5-20 illustrates an example of a class `PrettyPrint` in a Hyper/J-like system. The intention of `PrettyPrint` instances is to print a number of parameters in a pretty way to some output stream (the output stream is not shown in the example). Therefore, the class provides a number of methods `printout` which receive a number of parameters (at least one). According to [KPRS00] it is reasonable to use an aspect-oriented system to specify a number of constraints like pre or post conditions. Hence, it is reasonable to modularize code that checks the parameters for `null`. Since `null` is not a subtype of a primitive type like `int`, `char` or `boolean` in languages like Java, it is desirable to select only those methods whose parameters are not primitive. Figure 5-20 illustrates two methods where one has a primitive parameter and the other one has a non-primitive. However, since methods are only identified by their name (and not by their parameter types) it is not possible to select only the method with the `String` parameter.

The problem of not being able to address the desired static join points (not even by enumerating them) can be reduced to the problem of not being able to have identities for join points, i.e. properties that identify a static join point in a unique way. In case the static join point could be identified by a corresponding **identity property** it would be possible to enumerate exactly the desired join point. The idea of join point identities corresponds to the concept of **object identity** known from object-oriented programming (cf. e.g. [KhCo86]) or the idea of primary keys in the relational calculus (cf. [Codd70]) where the primary key is just a single attribute.

Although the previous discussion was related to static join points, it is possible to transfer the same discussion to dynamic join points. In contrast to static join points (that can be extracted by a static analysis of the base system) each dynamic join point has conceptually its own identity (see section 5.3.1). For example, each method call at runtime represents a unique join point although it might be represented by the same join point shadow. However, the idea of identity properties can be directly applied to join point shadows.

Shadow Identity property: A property is an identity property, if the property uniquely identifies a corresponding static join point, or a dynamic join point's shadow.

In principle, the idea of an identifying property represents its own dimensions and join point classes can be distinguished according to whether or not they provide a shadow identity property.

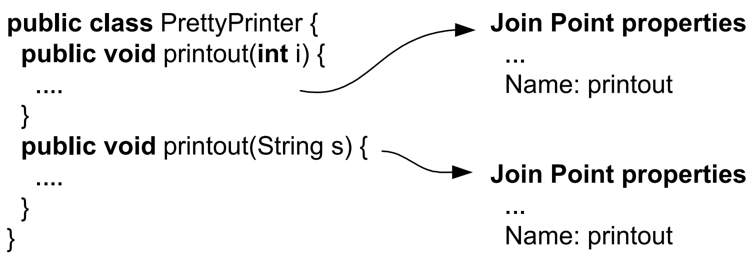

```
public class PrettyPrinter {
  public void printout(int i) {
    ....
  }
  public void printout(String s) {
    ....
  }
}
```

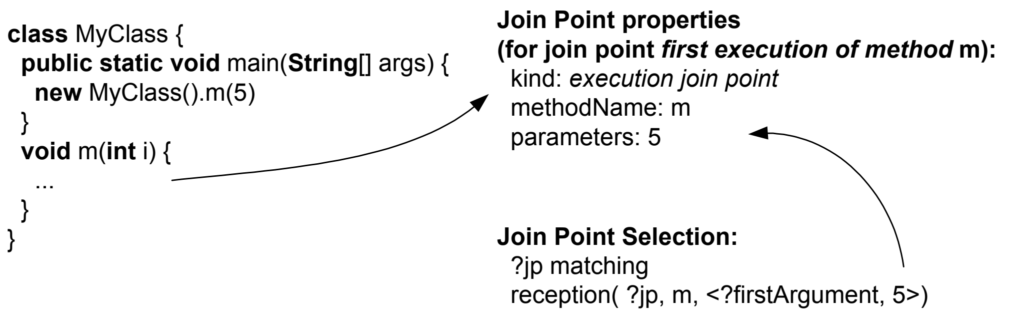
Figure 5-20. Join Point properties of methods `printout` in a Hyper/J-like system (no distinguishing characteristics for printout methods).

However, the idea of identifying a join point is not only restricted to a single property. Possibly a conglomerate of properties permits to identify a join point in a unique way (which corresponds to *compound keys* known from relational database systems, cf. [Codd70]). For example, in Java all methods within a class definition have a unique signature. If an aspect-oriented system provides for a method definition join point a number of (static) properties representing a signature and also a property representing the defining class then these properties altogether represent an identity for the method definition. Consequently, systems like AspectJ which provide such properties for method execution join points provide an identity for the corresponding join point shadows (although there is no single property representing that identity).

Although a base language might already provide from its syntax and semantics a possible set of identifying properties, this is not (in general) true for a large number of join point classes. For example, static method call join points in a Java-based aspect-oriented system cannot be directly identified by a set of lexical constructs: A method definition possibly contains an arbitrary number of method calls that refer to the same method definition. Consequently, properties for static method call join points representing the method containing the call as well properties representing the called method are not sufficient to identify the method call join point. Hence, systems like AspectJ which provide only such properties for method calls do not provide (neither an atomic nor a compound) a shadow identity for the corresponding dynamic join points.

5.4.3.6 Dynamic properties: Data and metadata properties

The distinction of static and dynamic properties only shows that there are properties which rely on runtime information. However, it turns out that different systems provide different kinds of dynamic properties.



```
class MyClass {
  public static void main(String[] args) {
    new MyClass().m(5)
  }
  void m(int i) {
    ...
  }
}
```

```
Join Point properties
(for join point first execution of method m):
  kind: execution join point
  methodName: m
  parameters: 5

Join Point Selection:
  ?jp matching
  reception( ?jp, m, <?firstArgument, 5>)
```

Figure 5-21. Join Point Selection based on variable values (taken from [GyBr03] with modifications).

For example, in AspectJ (up to version 1.06) the only available dynamic properties for join points were dynamic types of objects participating in a join point. For example, a method call could be selected due to the dynamic type of the calling object, the called object, or the arguments. However, it was not possible to select join points because of values associated with variables. For example, it was not possible to select a method call join point because some actual parameters were `null`.[64].

Systems like the one proposed in [GyBr03] permit to select join points not only because of runtime types but also because of values of variables and parameters. Figure 5-21 illustrates a join point selection taken from [GyBr03] with some modifications[65]. The JPSL selects all reception join points of method calls where the corresponding

[64] Just since version 1.06 AspectJ provides the `if`-pointcut designator which permits to select join points based on the result of a boolean expression.

[65] In the example, the underlying base language is assumed to be Java. In [GyBr03] the underlying language is a Smalltalk dialect.

method has the name `m` and the first passed parameter has the value `5`. In the example, method `m` defined in class `MyClass` is invoked only once (from the `main` method) and the passed actual parameter is an integer with the value `5`. The join point selection selects all reception join points (which corresponds to execution join points in AspectJ) where a method `m` is invoked (i.e. the method name property is addressed) and the first passed parameter has the value `5`. In contrast to AspectJ, it is possible to select the method execution based on the value `5` and not only because the passed parameter has the dynamic type `int`.

Hence, different aspect-oriented systems provide different kinds of dynamic properties with respect to what kind of runtime data they represent. This thesis calls the underlying design dimension **data representation** and distinguishes between **meta-data properties** and **data properties**.

> **Metadata join point property:** A dynamic property is a metadata join point property if it represents some metadata on runtime data (like for example runtime types) for a given join point.

> **Data join point property:** A property is a data join point property if it represents runtime data (and no metadata) for a given join point (like values of variables).

Although the distinction between metadata and data join point properties seems to be obvious, there are still a number of situations where such distinction is not that clear. In introspective languages like for example Java it is possible to gain some metadata from objects (cf. for example [Chib00]). For example, it is possible to get the class of an object or to retrieve the defined members of a class. Consequently, a join point that provides no metadata but data indirectly provides some metadata in such languages. However, even in introspective languages it is potentially not possible to gain the same meta-information from data than possibly provided by a metadata join point property. For example, a method call join point can be equipped with properties that provide information about the static types of the actual parameters. In Java the Reflection API does not permit to determine the static types of local (i.e. non-member) variables.

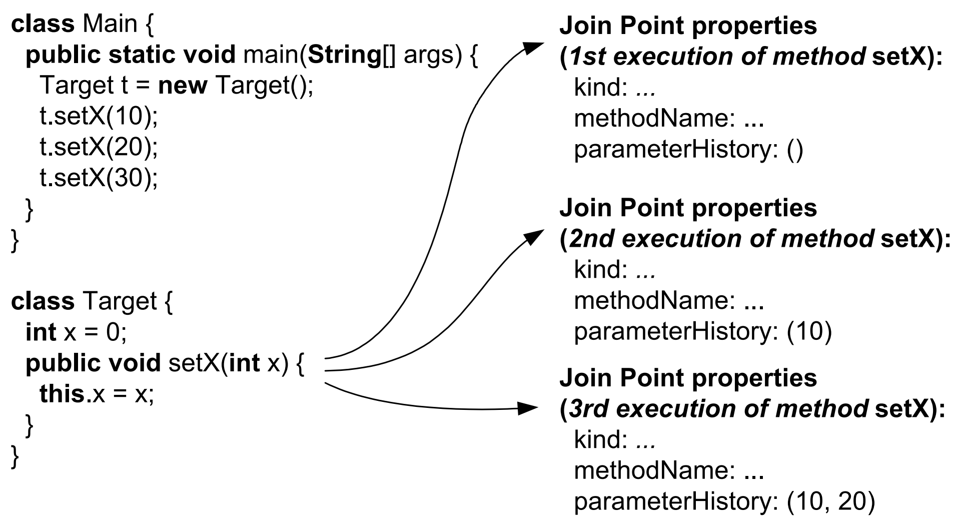
In general, the distinction between metadata and data depends on the underlying programming language. In object-oriented languages such a distinction also depends on underlying libraries. For example, in systems like Java or C++ there is a type `boolean` which has the values `true` and `false`. Consequently, if for example (boolean) parameters of a method call join point are equipped with the information that the parameters are of type boolean such information represents metadata properties. If there are properties that represent the values `true` or `false`, such properties are data properties. In Smalltalk the situation is slightly different. `Boolean` is a class and `True` and `False` are subclasses of `Boolean`. Consequently, in such a system a join point property that describes whether a parameter has the value `true` or `false` is a meta join point property.

5.4.3.7 Dynamic properties: Current, past and future properties

Dynamic properties represent data that appears at runtime. Obviously, dynamic join points can be provided with dynamic properties that represent the runtime data at the **current join point** [WKD02]. Such data reflects the system's state at the moment when the join point appears. Systems like AspectJ provide runtime properties for the current system state like the runtime type of a method call join point's parameters, for example.

However, an aspect-oriented system potentially provides not only information about the current system state but also information about past system states as well as potential future system states (cf. [HSU05a, StHa05]), i.e. properties potentially represent information about the **progress** of a system.

A typical example for such properties is the proposal of **dataflow pointcuts** as proposed in [MaKa03]. With the aid of a dataflow pointcut it is possible to state what objects influenced the state of a certain (current) object. In order to provide such a pointcut the aspect-oriented system needs to provide some properties that give some information either about some actions like method calls that occurred in the past or information about the data that has been already passed to the current object in the past. A similar approach (although not directly referring to data flow) has been proposed in [WaVi04] where the selection of join points depends on previous events that happened in the program execution.



```
class Main {
  public static void main(String[] args) {
    Target t = new Target();
    t.setX(10);
    t.setX(20);
    t.setX(30);
  }
}

class Target {
  int x = 0;
  public void setX(int x) {
    this.x = x;
  }
}
```

```
Join Point properties
(1st execution of method setX):
  kind: ...
  methodName: ...
  parameterHistory: ()

Join Point properties
(2nd execution of method setX):
  kind: ...
  methodName: ...
  parameterHistory: (10)

Join Point properties
(3rd execution of method setX):
  kind: ...
  methodName: ...
  parameterHistory: (10, 20)
```

Figure 5-22. Join Point Properties representing the history of parameters for method `setX`.

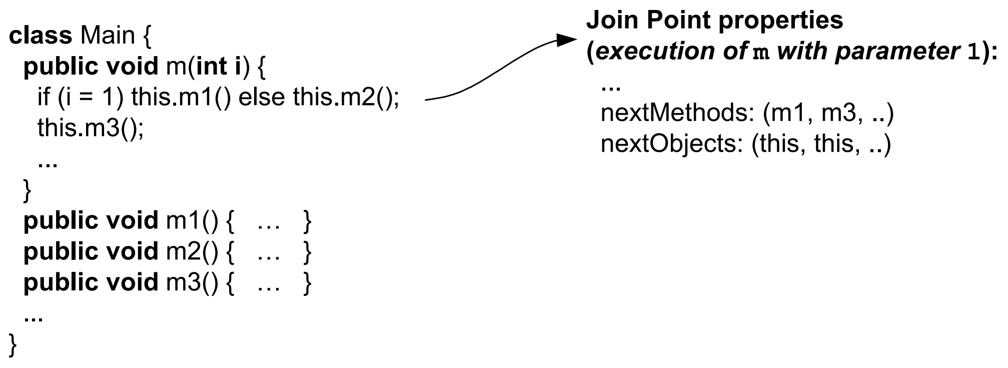
Figure 5-22 illustrates a hypothetical aspect-oriented system that provides properties that give some information about the history of the program (similar to the one proposed in [MaKa03]). The (structured) property `parameterHistory` for a method execution join point provides information about the parameters that have been already passed to the method.

Theoretically, it is also possible to provide a small number of properties that refer to information about future data or future events[66]. For example, the system might provide

[66] Of course, a system cannot provide properties for all future actions due to the **haltproblem** (cf. for example [HMU01]).

information about the statement that is about to be executed next – in case such a next statement can be computed from the information available at the current join point.

Figure 5-23 illustrates a hypothetical aspect-oriented system that provides dynamic method call join points. When method `m` is about to be executed with the parameters 1 it is possible to determine at least the next two methods that are about to be invoked.



```
class Main {
  public void m(int i) {
    if (i = 1) this.m1() else this.m2();
    this.m3();
    ...
  }
  public void m1() {  ...  }
  public void m2() {  ...  }
  public void m3() {  ...  }
  ...
}
```

```
Join Point properties
(execution of m with parameter 1):
  ...
  nextMethods: (m1, m3, ..)
  nextObjects: (this, this, ..)
```

Figure 5-23. Join Point Property representing the next methods to be called.

Based on the previous discussion there is a difference between different kinds of dynamic properties with respect to what kind of data they represent concerning the **application's progress**. This thesis distinguishes between **current**, **past,** and **future data properties**.

Current data property: A dynamic property is a current data property if it represents data that belongs to the current state at the corresponding join point.

Past data property: A dynamic property is a past data property if it represents data for the current join point from a previous system state.

Future data property: A dynamic property is a future data property if it represents data for the current join point from a future system state.

Obviously, systems based on dynamic join points provide at least some current data properties. For example, in AspectJ it is possible to select join points based on the dynamic types of parameters. The same is true for systems like Steamloom [BHMO04] or Andrew [GyBr03]. In general, in order to determine past or future data properties the system needs to perform a number of advanced analysis techniques (cf. for example [SeMo04]).

A problem with the distinction of current, past, and future data properties is that there are situations where it is not clear for a given property what kind of property it is. For example, it is not obvious how to handle control flow specific selections of join points as provided for example in AspectJ's `cflow` pointcut designator or the `cfFirstInstance` advice activator in AspectS. Both constructs permit to select join points based on some properties of the call stack in the local thread. However, it is possible to interpret the control flow specific selection in different ways.

If the construct is understood in terms of the (thread local) call stack and the underlying system provides access to the stack (which is true for AspectS, see section 5.4.3.4), then the property that represents the call stack is just a current data property since it represents data of the underlying runtime system. In case the underlying system does not provide access to the call stack (like in AspectJ), the aspect-oriented system needs to create a corresponding data structure – a structure that reflects on previous calls on the call stack. Consequently, such a data structure represents a past data property.

However, control flow specific selection can be interpreted also in a different way. Instead of having the focus on the call-stack and thread-local property it is possible to interpret it as a construct in order to select join points which are called by a certain method (this corresponds to the interpretation in [MKD03] that states that *the cflow pointcut matches join points that are created during (certain) method calls*[67]). This interpretation refers to the dependency of method calls and does not directly refer to the local thread. Consequently, a method could be selected because another method was called before. In single-threaded applications, this interpretation corresponds to the previous one because a method that invokes another method is inherently part of the same thread. In multi-threaded applications, the first interpretation refers only to method calls in the same thread. Consequently, if a method calls another one by creating a new thread the previous method call is not part of the control flow. This is not true for the second interpretation because it does not distinguish between different threads but has the focus on the dependencies between method calls.

The problem with properties for control flow specific selections is that the second interpretation considers such properties as past data properties while the first one potentially considers them as current data properties. In order to handle this conflict this thesis considers properties that reflect on the call stack as past data properties. In case the underlying system already provides access to the call stack, the property is to be annotated with this additional information.

5.4.3.8 Orthogonality of design dimensions

The previously discussed design dimensions of join point properties – dynamicity, level of correspondence, level of locality, and structuredness are orthogonal i.e. they can be applied independently of each other for the design of a certain property (see Figure 5-24)[68]. Typically, each join point is encoded with a number of properties whereby each property can be based on different design decisions. For example, one property of a (dynamic) method call join point can be a static and atomic property while another one is dynamic and structured.

A number of such properties occur more often than others. Static, local, and atomic properties with a direct property correspondence are provided by a large number of aspect-oriented systems. Such properties are for examples method names for (static of dynamic) method calls or all elements that belong to a method's signature (like return types for method definition join points) – such properties are provided by all known systems like AspectJ, Hyper/J, AspectS, etc. The reason why such properties are

[67] See [MKD03], page 56.

[68] For reasons of simplicity the identity characteristic is left out here.

provided seems obvious: Such properties can be provided by performing a very simple static analysis on the source code and by comparatively easy source code transformations (in case the aspect-oriented system is realized by such a technique).

Also, dynamic, structured, and local properties with a direct property correspondence are quite often provided by aspect-oriented systems that provide dynamic join points – examples for such properties are target objects of method calls as well as runtime types of objects participating in method calls. AspectJ provides such properties.

Only few systems provide properties that are non-local or which have an abstract property correspondence. Non-local dynamic properties are almost exclusively restricted to control-flow specific properties. AspectJ first introduced such a property. Systems like Steamloom whose design is closely related to AspectJ as well as AspectS provide such properties, too (whereby the latter one provides them already on the base language level, see section 5.4.3.4). Also, non-local static properties are quite often very restricted. For example, AspectJ provides static type information for method call join points as non-local join point properties.

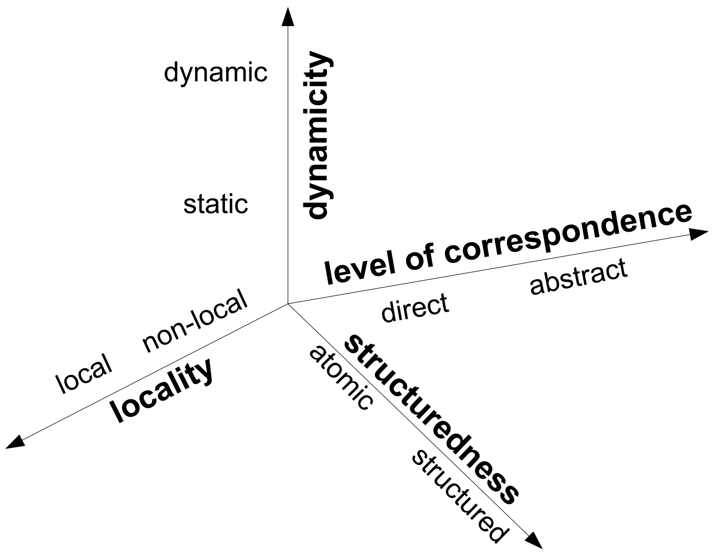


Figure 5-24. Design dimensions of join point properties.

Similar to the previous discussion some kinds of dynamic properties occur more often than others. For example, systems that provide dynamic join points (and which are closely related to AspectJ) typically provide data and metadata properties and current join point properties. There are relatively few systems that provide past join point properties (except control flow specific properties, see section 5.4.3.7). Even less systems that provide future join point properties. In fact, the only system known to the author of this thesis that provides future join point properties is the one introduced in [ÅLSM03].

A special situation is the combination of the application's progress with the locality characteristic of join points. Past as well as future join point properties represent information that is inherently not available at the corresponding join points; otherwise such a property would be a current join point property. Consequently, past and future join point properties cannot be local. According to this, the dimension of the application's progress is not orthogonal to the locality characteristic of join point properties (see Figure 5-25).

5.4.4 Join Point Addressing

Join point properties express the information available for each join point. Based on these properties join points are selected by the developer by using language constructs provided by the JPSL.

In general, the language constructs for addressing join points via their properties represent an ordinary query language. Hence, characteristics of query languages like **descriptiveness**, **adequacy** [HeSc91], or **computational completeness** [ABD+89] could be used to describe characteristics of the JPSL. However, current aspect-oriented systems are much more restrictive than known query languages (for example, the language constructs in AspectJ are not computational complete[69]). Therefore, it is necessary to describe their characteristics on a finer grained level.

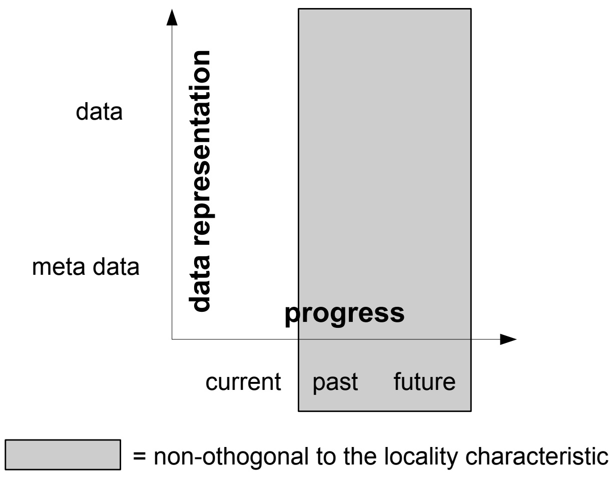


Figure 5-25. Design dimensions of dynamic join point properties.

The easiest way to select a join point is to specify the value of a certain property. This approach can be found in all known aspect-oriented systems. For example, in AspectJ a static class definition join point used by an introduction is selected by specifying the name of the class. In the same way, an operation in Hyper/J is selected by specifying the operation name. Consequently, language constructs are provided to enumerate properties and values of join points to be selected.

However, a JPSL possibly provides more complex means to specify a selection. Section 5.4.1 discussed the ability to select a join point based on a structural correspondences or the selection of a join point based on some type correspondence in AspectJ. Likewise, a number of systems (like AspectJ and Hyper/J) permit to use wildcards to specify a join point selection. That means, there are some common

[69] With the existence of the if-pointcut AspectJ's pointcut language is turing-complete.

characteristics between aspect-oriented systems as well as significant differences with respect to how join points properties can be addressed in join point queries.

Similar to the previous sections, this section discusses in the following design dimensions of selection constructs.

5.4.4.1 Lexical and indirect value addressing

In the most trivial case the JPSL provides constructs that permit developers to specify the value of a certain property. This corresponds for example to the specification of a class name within a type pattern in AspectJ. This class name is lexically compared with all class names in the application. Also, in Hyper/J an operation is specified inside the concern mapping by its name. Hence, all methods matching the specified name are selected. In fact, this kinds of join point selection is the most straight forward way and all known aspect-oriented system provide this. **Lexical crosscutting** [LLM99] as well as **enumeration-based crosscutting** [GyBr03] are based on such a join point selection.

However, for example in AspectJ, it is also possible to specify a join point selection without referring to a join point property in a lexical way, e.g. by means of the + operator (as already shown in Figure 5-12). By using such a construct, it is possible to select a join point based on a type relationship between a join point property and a certain type. Consequently, there is no lexical correspondence between the specified selection and the corresponding join point property.

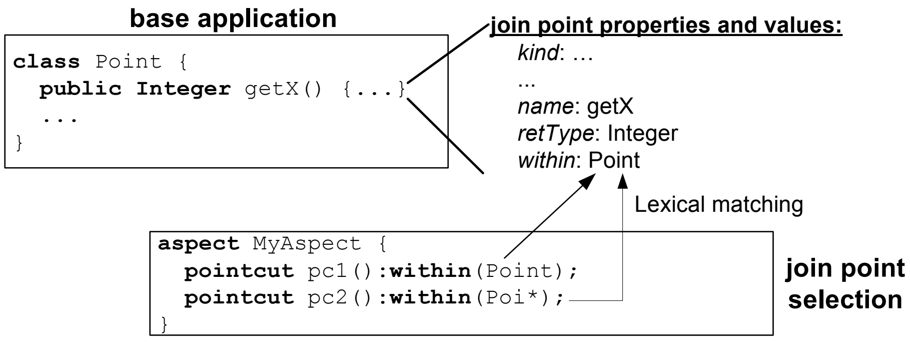


Figure 5-26. Lexical value specification in AspectJ.

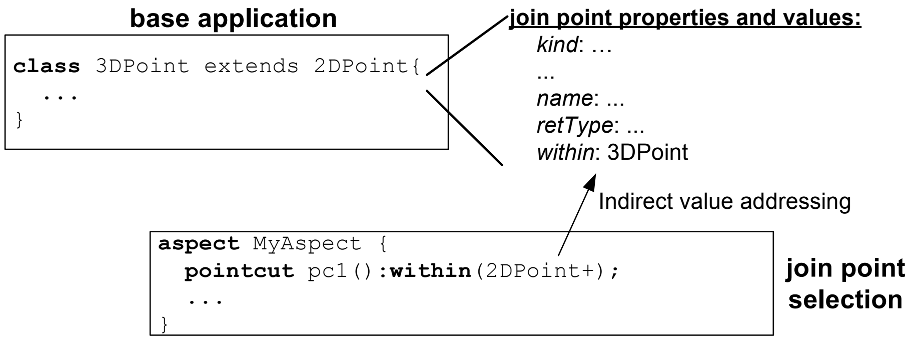
Both kinds of selection constructs differ significantly. The first construct addresses a property's value directly by specifying the same value in the selection construct. The latter one permits a more abstract way of addressing a certain property because the selection construct does no refer to the join point property in a lexical way. Consequently, this thesis identifies the **directness of value addressing** as a design dimension of selection language constructs and distinguishes between **lexical value addressing** and **indirect value addressing**.

Lexical value addressing: Lexical value addressing is provided by a language construct, if the developer needs to specify the lexical characteristics of a property's value.

Indirect value addressing: Indirect value addressing describes the facility of a join point selection language construct to permit the specification of characteristics of the property's value in a non-lexical way.

The difference between both kinds of directness of value addressing is whether within the selection the developer specifies characteristics of the property that are compared directly to the property's value. Examples of lexical value addressing can be found in all known aspect-oriented systems. In AspectJ a type pattern which is used by a number of pointcut designators like `within` or `this` permits the user to specify the selection on a lexical basis: A selection of a return type can be achieved by specifying the type's name. Also, using the `*` operator in AspectJ is a lexical value addressing because the selection criterion contains a string that is lexically compared with the corresponding property's value.

Figure 5-26 illustrates two different ways of lexical value specifications in AspectJ. The pointcut `pc1` addresses the `within` property of all join points in the program and specifies the value `Point`. The second pointcut also addresses the value of `within` in a lexical manner. Here, the developer does not completely specify the value of the property, but combines a lexical specification (the corresponding class begins with the characters `Poi`) with the wildcard `*`, i.e. the developer abstracts from an arbitrary number of characters at the end of the type name.



```
class 3DPoint extends 2DPoint{
  ...
}
```

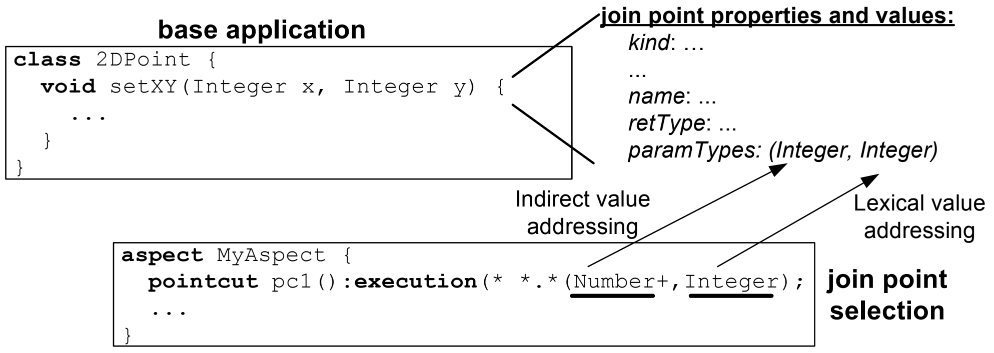
```
aspect MyAspect {
  pointcut pc1():within(2DPoint+);
  ...
}
```

Figure 5-27. Indirect value specification in AspectJ.

Figure 5-27 illustrates indirect value addressing in AspectJ using the + operator. The `within` property of all join points that appear inside of the class `3DPoint` is addressed by specifying the superclass `2DPoint` in combination with the operator +. As a consequence, there is no lexical dependency between the property value being addressed and the specification in the join point selection because the selection does not directly contain any lexical element of the corresponding property `within`. However, it must be mentioned that `2DPoint` is lexically addressed (because there is a type named `2Dpoint` in the application). But this addressing is a lexical one of a different property (the `name` property of a class definition join point) and not of the property whose value is addressed here.

An indirect value addressing is a more advanced way of addressing a certain property's value. Instead of comparing a specified string with a property's value on a lexical basis, an indirect value addressing permits to address a value's property on a more abstract level. An indirect value addressing assumes the existence of language constructs that permit to specify some non-lexical constraints on the value. For example, if a structured property describes a method definition's parameter list, a language construct that permits to address such methods because of the number of parameters is an indirect value addressing. Another example of an indirect value addressing is the + operator in AspectJ that can be applied to a type pattern: Applying the postfix operator + to a single type descriptor specifies all subtypes of the underlying type. Hence, there is no lexical dependency between the specification and the property's value. Instead, there is rather a **semantic dependency** between a specified value (a super type's name) and the value of the property (since the + operator refers to subtype relationships).

Although both kinds of addressing are conceptually different, an aspect-oriented system possibly provides language constructs for both kinds of addressing. In AspectJ it is possible to specify a selection of a property based on a lexical as well as on an indirect basis.



```
class 2DPoint {
  void setXY(Integer x, Integer y) {
    ...
  }
}
```

```
aspect MyAspect {
  pointcut pc1():execution(* *.*(Number+,Integer);
  ...
}
```

Figure 5-28. Combined indirect and lexical value specification in AspectJ.

Figure 5-28 illustrates such a combination of both kinds of value addressing. The first parameter is indirectly addressed because the + operator is used in combination with a lexical specification of the superclass. The second parameter of the structured value of the property `paramTypes` is lexically addressed because the type `Integer` is specified in the join point selection, which corresponds to the second parameter's value. Hence, the value of the join point property `paramTypes` is addressed lexically as well as indirectly.

5.4.4.2 Closed and open value addressing

In a number of aspect-oriented systems it is possible to specify join point selections in a way even those join points are potentially addressed that are not yet contained in the application. This property is essential if a number of characteristics of join points for a certain aspect are known but it is not clear how often (or whether at all) they appear in the underlying application or future evolutions of the application. For example, in AspectJ and Hyper/J it is possible to use wildcards for addressing join point properties. Such wildcards permit to specify the value of a certain join point property only partially. Such kind of addressing is interesting in situations where the base application is based on a number of **naming conventions**[70].

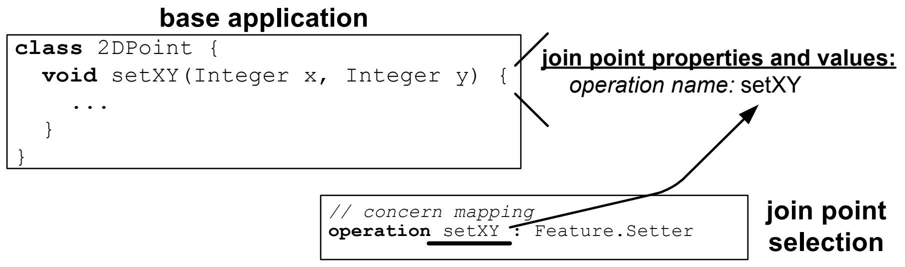
[70] One example of a common naming convention is the naming of field accessing methods using the prefixes `get` and `set`. In Java, architectures like **Java Beans** [Sun04a] or **Enterprise JavaBeans** (EJB, [Sun04b]) make use of such conventions.

The characteristic of such addressing in comparison to a pure enumeration of properties is that it refers to an infinite set of possibly matching join point property values: Join points that are added to the system due to an evolution of the system can be considered in that way.

Consequently, this thesis distinguishes between these kinds of addressing and calls the underlying design dimensions the **openness of value addressing** and distinguishes between **closed value addressing** and **open value addressing**.

> **Closed value addressing**: An aspect-oriented system provides a closed value addressing if it provides means to specify a precise set of values for a given property, i.e. at specification time it is known how many values of join point properties match the specification.
>
> **Open value addressing**: A join point selection language permits an open value specification if it permits to specify an imprecise set of values for a given property, i.e. at specification time is it not known how many values possibly match the specification.

Most of the previous examples already contained an example of a closed value addressing. For example, the pointcut `pc1` in Figure 5-27 addresses the value of the `within` property completely (by specifying the value 2D`Point`), i.e. there is only one possible value matching this specification. Figure 5-29 illustrates another example of a complete value specification in Hyper/J: The value of the property `operation name` is addressed by specifying its complete value within the concern mapping file. The term **enumeration-based crosscutting** [GyBr03] is closely related to this way of value specification: A developer enumerates all possible values for a given property and determines in this way a finite number of possible values matching the selection criterion.



```
class 2DPoint {
  void setXY(Integer x, Integer y) {
    ...
  }
}
```

```
// concern mapping
operation setXY : Feature.Setter
```

Figure 5-29. Closed (and lexical) value addressing in a Hyper/J concern mapping file.

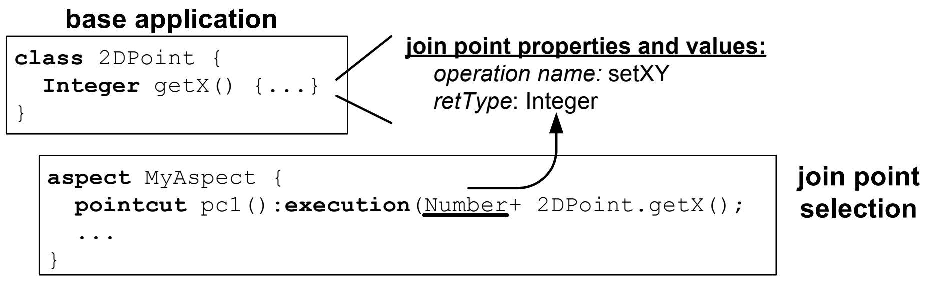
An open value addressing on the other hand specifies an infinite number of possibly matching values. Consequently, a maximum number of values that fulfill this specification cannot be determined. Examples of open value specifications are the wildcards within a pointcut specification in AspectJ as well as in hypermodules[71] in

[71] In Hyper/J wildcards can only be used within some composition rules and not for example within concern mapping files (see section 2.3).

Hyper/J. For example, the pointcut pc2 in Figure 5-26 is an open value specification: All class names starting with the identifiers Poi correspond to this specification. For the same reason, the + operator used within type patterns in AspectJ (as illustrated in Figure 5-28) is an example for an open value addressing: In case the type structure evolves, all new classes that subtype Number are addressed.

The main intention for distinguishing between closed and open value addressing pertains to their impact on the evolution of base applications: A closed value addressing does not consider any possible property values that may appear in an evolved version of the base system[72]. If the base application evolves and grows, a number of new join points appear in the system (likewise previous join points may disappear). Since the maximum number of matching values in a closed value specification is fix, additional property value pairs in the evolved base program are not considered. Not even those values that might be important to guarantee the aspect's consistency. As a consequence, evolving base applications may lead to some inconsistencies caused by aspects[73] when the join points are addressed by a closed value specification.

Note that the term open value addressing does not determine in what way the open value specification has to be done. Obviously, the straightforward implementations of wildcards in AspectJ as well as in Hyper/J are examples for open addressing. However, systems that permit to specify a value via a **regular expression** (like proposed in [PGA02]) by using a **Kleene star** would be an example of an open value addressing. In principle, all kinds of **formal languages**[74] that permit to describe languages having an infinite number of words can be used to specify values in an open way.



```
class 2DPoint {
  Integer getX() {...}
}
```

```
aspect MyAspect {
  pointcut pc1():execution(Number+ 2DPoint.getX();
  ...
}
```

Figure 5-30. Open value specification without multiple matching in an AspectJ-like system with (static) join points.

It needs to be pointed out that the openness of value addressing does not necessarily mean that the number of join points to be selected are infinite. This is because join points are selected because of all properties and values being specified in combination with the underlying join point encoding and the underlying semantics of the base language – and because of a single property that is potentially addressed in an open way.

[72] See [KRH04] for a more detailed discussion on the problem of evolving base applications in aspect-oriented systems.

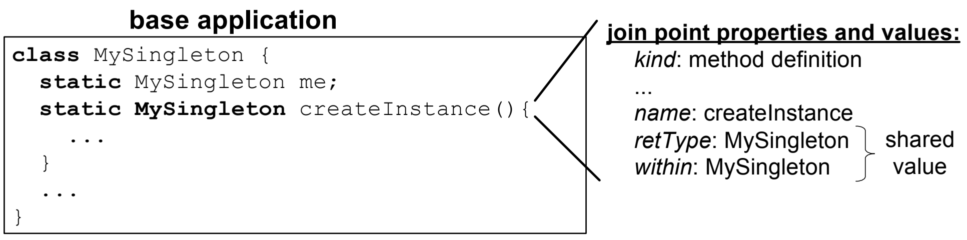
[73] In fact, a similar problem was originally addressed by **Adaptive Programming** [Lieb96], which permits to specify functionality independent of the underlying class structure.

[74] See [ASU86] for example for an exhaustive discussion on the specification of formal languages.

Figure 5-30 illustrates a join point selection where one value (the return type) is addressed in an open manner (assuming an AspectJ-like system with pure static join points). The other values (method name, defining class, parameter types) are addressed in a closed manner. The open addressed value refers to the return type of the corresponding method in the base application. Hence, the corresponding method in the base application is selected. However, if the application evolves **incrementally** (see [WeZd88] for the discussion on incremental modification in object-oriented programming) by adding subtypes to class `2DPoint` the number of selected join points is the same. In fact, the number of static join points selected by `pc1()` is at most one, because the underlying programming language does not provide the declaration of covariant return types[75].

5.4.4.3 Stand-alone and sharing property addressing

The previous design dimensions are related to the addressing of values for a certain property. However, a more interesting issue is the specification of values that are shared between different properties, i.e. whether it is possible to specify the same set of criteria for a number of different properties. In relational databases, **join operations** [Codd70] are such value sharing operations where the resulting relation depends on values shared between different relations and possibly different attributes.



```
class MySingleton {
  static MySingleton me;
  static MySingleton createInstance(){
    ...
  }
  ...
}
```

Figure 5-31. Method definition join point in a singleton class with a shared property value.

The need for the ability to have a shared value specification can be directly derived from a simple implementation of the **Singleton** design pattern [GHJV95] (see also section 3.3.1). A singleton is a class that is itself responsible for creating and maintaining its instances. Figure 5-31 illustrates a straightforward implementation of the singleton design pattern for a class `MySingleton` in Java[76]. A typical characteristic of such an implementation is that the signatures of the class method have some common characteristics: The value of the name property is often the same (`createInstance` in the example) and the value of the return type property as well as the name of the defining class is the same (`MySingleton` in the example). Figure 5-31 also contains

[75] Covariant overriding is not possible in the Java langauge up to version 1.4 (althouth the virtual machine permits this). By adding genericity to the language [BOSW98], covariant overriding becomes possible within generic types.

[76] Here, the discussion of topics like garbage collection or polymorphism in the implementation of the singleton design pattern in Java is avoided. For a more comprehensive discussion this thesis refers to [Gran98], pp. 127-133.

characteristic join point properties for the (static) method definition join point similar to AspectJ.

For selecting all methods that return the instances in singletons, it is desirable to specify the common characteristics of those methods. If the class name is not known at selection time, i.e. if the selection should be specified independent of the base application, there is a need to define the characteristic that the class method's return type corresponds to the class where the method is defined in. I.e. it needs to be specified that the value of the return type corresponds to the value of the defining class.

Such a specification on the other hand assumes that the JPSL permits to specify such a selection where a number of different properties have the same value. If the developer wants to specify a selection independent of the underlying class, there is a need to specify the selection in a way that both return type and class definition type are equal without directly referring to a certain class.

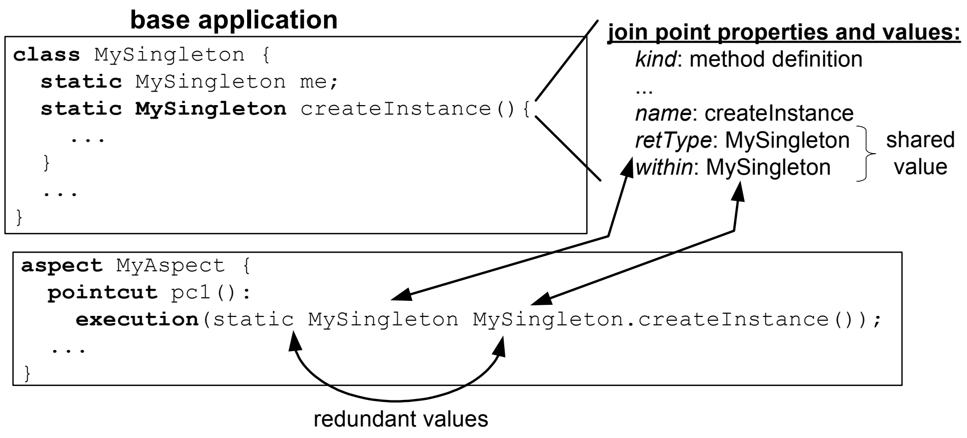


Figure 5-32. Redundant value specification caused by the stand-alone value specification in AspectJ.

Although from the singleton example it seems natural to permit the ability to share values among different join point selections, it turns out that such a features is not necessarily provided by aspect-oriented systems. Consequently, this thesis distinguishes between aspect-oriented systems according to whether their join point selection languages permit to specify shared common characteristics of property values for a number of join point properties. This thesis calls the corresponding dimension **level of value sharing** and distinguishes between **stand-alone** and **shared property addressing**.

Stand-alone property addressing: The join point selection language provides a stand-alone property addressing if it provides language constructs for specifying only characteristics for values of a single join point property.

Shared property addressing: The join point selection language provides a shared property addressing if it permits to share values across a number of properties.

A stand-alone property addressing is the easiest and most common kind of specifying a selection in aspect-oriented systems like for example Hyper/J and AspectJ:

All values for certain join point properties need to be specified separate from each other. As a consequence, whenever one single criterion is needed for a number of different join point properties, a copy of such criterion needs to be specified. Hence, in such situations a redundant specification of criteria is necessary.

Figure 5-32 illustrates the consequence of stand-alone property addressing in AspectJ based on the previous discussion. The aspect `MyAspect` selects in its pointcut `pc1` all methods of a singleton that create (or simply return) an instance of that class based on a lexical value addressing. However, what needs to be specified is that the value of the `within` property corresponds to the value of the return type parameter. Since AspectJ does not permit to specify shared values it is necessary to specify the corresponding value twice for each property: The name of `MySingleton` occurs twice in the pointcut definition. Another consequence of providing only a stand-alone property addressing is that it is not possible to specify the essential characteristic of this singleton-specific selection without referring to the corresponding class name. I.e. it is not possible to specify "the return type of the method to be selected corresponds to the class defining the method" on an abstract level without specifying the name of the class.

Shared value specifications permit to specify a selection criterion for a number of properties within a single selection. The main benefit of shared value specifications is that redundancies that occur in certain situations in stand-alone value specification are not necessarily needed.

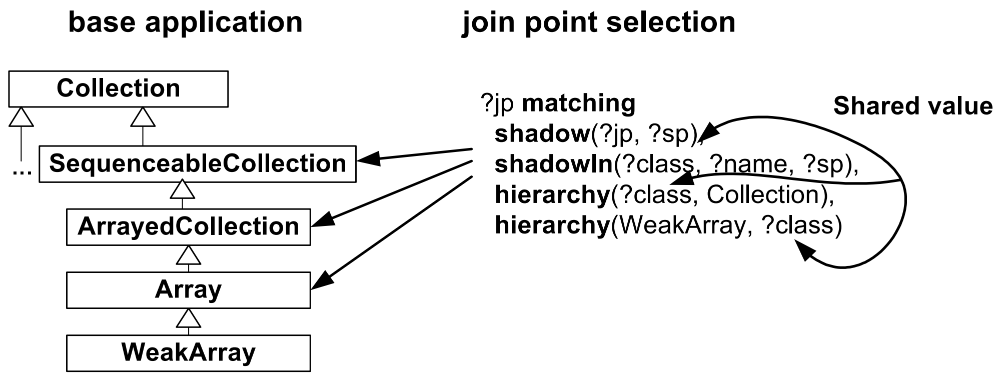


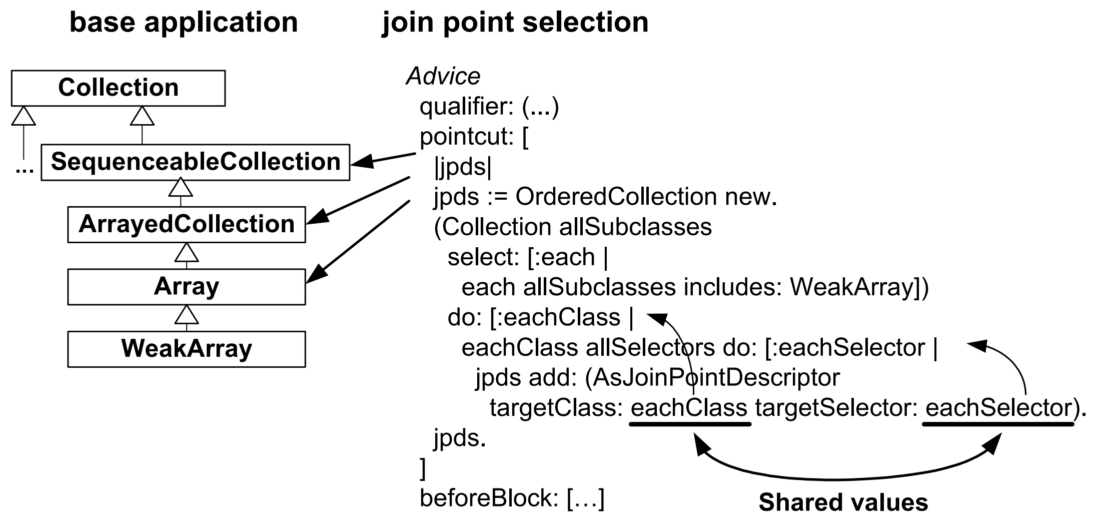
Figure 5-33. Join point selection with shared values across different join point properties.

Figure 5-33 illustrates a join point selection using a shared value specification whereby the join point selection language is based on a logic programming language (taken from [GyBr03]). The predicate `shadow` addresses all join points shadow in the base application and binds them to the variable `?sp`, the corresponding join points are bound to `?jp`[77]. The predicate `shadowIn` determines all join point shadows `?sp` within the classes `?class` and their methods `?name`. The predicate `hierarchy` determines for a given class (described in the second parameter) its subclasses, i.e. a predicate like `hierarchy(?class, Collection)` determines all subclasses of

[77] The underlying join point model for the proposed language is behavioral and dynamic (see [GyBr03]). Hence, it distinguishes between the join point itself and the corresponding join point shadow.

class `Collection` and binds them to the variable `?class`. The same variable can be applied to the same predicate. I.e. the conjunction "`hierarchy(?class, Collection), hierarchy(WeakArray, ?class)`" determines all subclasses of `Collection` that are superclasses of `WeakArray` (i.e. the variable `?class` is unified). As a consequence, the join point selection selects all join points that occur within classes that are subclasses of `Collection` and superclasses of `WeakArray`. On the left hand side of Figure 5-33 a small extraction of subclasses extending `Collection` in the collection framework in the Smalltalk dialect Squeak [IKM+97] is illustrated. According to this, the join point selection selects all join points that occur within classes `SequenceableCollection`, `ArrayCollection` and `Array`.

Another example of a shared value specification is AspectS where the computation of join points and sharing of values among join point properties is not part of the framework itself but part of the underlying language. Figure 34 shows a shared value specification in AspectS similar to the previously introduced one. The join point selection is specified within the pointcut block that determines all join point descriptors the aspect should be woven to (the example neglects the kind of advice and advice qualifier as well as the code to be executed at the corresponding advice). For constructing join point descriptor objects, the code collects all subclasses of `Collection` that are also superclasses of `WeakArray`. Then, it collects all method selectors defined in such classes. After creating a corresponding join point descriptor object, that object is stored into the collection `jpds` which represents the result of that block (see section 2.4). When the aspect is installed, it is woven to exactly those join points. The sharing of values within the join point selection is achieved by common variables whose values are used for a number of join point descriptor: The variables `eachClass` and `eachSelector` represent the shared value specification.



```
Advice
  qualifier: (...)
  pointcut: [
    |jpds|
    jpds := OrderedCollection new.
    (Collection allSubclasses
      select: [:each |
        each allSubclasses includes: WeakArray])
      do: [:eachClass |
        eachClass allSelectors do: [:eachSelector |
          jpds add: (AsJoinPointDescriptor
            targetClass: eachClass targetSelector: eachSelector).
    jpds.
  ]
  beforeBlock: [...]
```

Figure 5-34. Join point selection with shared values in AspectS.

In aspect-oriented systems that provide shared value specifications, a more advanced kind of parameterization is possible: A property's value can be specified in one place and the join point class (possibly later on) in a different place. However, there is a large variety of how such parameterization can be realized. In AspectS such parameterization can be achieved by passing a shared variable from a different context. Also, **hook methods** [Pree95] can be used in the pointcut block and defined later on in a subclass.

5.4.4.4 Monolitic and incremental selection

A more abstract view on the selection (which is not directly referred to the selection of specific join point properties) reveals that join point selections among different aspect-oriented systems also differ with respect to how selections can be composed.

One kind of composing selections originates once more from the term enumeration based crosscutting: Aspect-oriented systems typically provide mechanisms to specify a number of different values for a certain property. Since aspect-oriented systems typically provide more than one single property for each join point, all known systems permit to compose different selection criteria for different properties – like for example selecting a method definition join point because of the method's name in conjunction with the number of parameters.

However, as also known from the object-oriented world there is one essential way of specifying and composing code fragments – called **incremental modification** (cf. for example [WeZd88]) - which is the stepwise refinement of a module based on a description of how it differs from another module. In object-oriented programming languages, such incremental modifications are achieved via different kinds of inheritance (cf. for example [Taiv96]) where a subclass defines how it differs from its superclass.

For aspect-oriented systems such an incremental modification seems to be essential and corresponds to the idea of specifying aspects that can be woven to a number of different applications (cf. for example [KRH04, HaUn03a] for further discussions). Since all applications come with their own set of join points it is desirable to specify the aspect's selection criteria only partly and to refine the aspect specification later on in order to match a specific application's needs.

```
public aspect AbstractAspect {
  abstract pointcut hook():
  pointcut myTargetCall(TargetClass t): this(t) && hook();
  before(TargetClass t): myTargetCall(t) {
    // join point adaptation
  }
}
public aspect ConcreteAspect extends AbstractAspect {
  pointcut hook():
    call(void MyTargetClass.foo());
}
```

Figure 35: Example application of the Abstract Aspect idiom.

In AspectJ as well as in Sally such an incremental modification of join point selections is achieved by a mechanism similar to inheritance: An abstract aspect can provide a number of (possibly abstract) pointcuts that are later on defined in a subaspect. This kind of refined join point selection is used by a number of AspectJ idioms (cf. [HaCo03, HSU03]), like for example **Abstract Pointcut**. In this idiom an abstract aspect specifies the behavior that needs to be executed at certain join points but leaves out the join point selection to be defined later on for application specific purposes. Figure 35 illustrates a code example for Abstract Pointcut. `AbstractAspect` specifies the adaptation of certain join points within its advice and also specifies within its pointcut `MyTargetCall` that the objects at which such join point occur must be of type `TargetClass`. However, the pointcut also refers to a pointcut `hook()` that is to be defined in a subaspect. In order to connect the aspect to a specific application the developer needs to define the pointcut `hook()`. In the example, `ConcreteAspect` selects the method call join points to method `foo()` in class `MyTargetClass`.

Although it seems quite essential to have such an incremental definition of join point selection, it turns out that not every system provides corresponding mechanisms. For example in Hyper/J, it is not possible to refine concern mappings.

Based on this observation, this thesis identifies the level of **adjustability of join point selections** as a design dimension of join point selection languages and distinguished between **monolithic** and **incremental join point selections**.

> **Incremental join point selections:** An aspect-oriented system provides incremental join point selections if a join point selection can be defined in one module and refined in other modules without the need to perform invasive changes in the first one.

> **Monolithic join point selections:** An aspect-oriented system provides monolythic join point selections if a join point selection has to be defined in one single module without the ability to be modified in other modules.

5.4.4.5 Unrestricted and restricted join point selection

The intention of a join point selection is to select all join points that are "appropriate" for the aspect. Under certain circumstances it is only necessary to select join points which occur in special modules. For example, in order to specify a persistency aspect for a given class definition it is sufficient if the selection language permits only to select join points which occur in a certain class. In such a situation the underlying crosscutting is *class-specific* (cf. [HaUn02a]) and the selection language only needs to address join points that occur within this class (in case of a static join point model). However, if one aspect should be able to handle the object persistency for more than one class there is a need to specify selections that address join points in the whole application – there is no restriction of where the join points occur.

In general, crosscutting code occurs on different levels. On the one hand there are crosscutting concerns where the resulting code is restricted to a small number of modules (like different methods in a single class definition). On the other hand the resulting code is possibly distributed over a large number of modules.

Different systems have different abilities in selecting join points. Systems such as AspectJ as well as Hyper/J permit to refer to any join point in the whole application – the join point selection is not restricted to some specific modules. However, systems like **Composition Filters** [AWB+93] restrict the selection of filters to the types they are defined in. Consequently, this thesis identifies the **scope of join point selection** as a design dimension of join point selection languages and distinguishes between **unrestricted** and **restricted join point selection**.

Unrestricted join point selection: An aspect-oriented system provides an unrestricted join point selection if the join point selection constructs can refer to any join point in the whole system.

Restricted join point selection: An aspect-oriented system provides a restricted join point selection if the system restricts each join point selection to a number of modules.

5.4.4.6 Orthogonality of design dimensions

Similar to the design dimensions of join points and join point properties the identified design dimensions of join point selection constructs are orthogonal. However, the identified design dimensions are on different levels of abstraction.

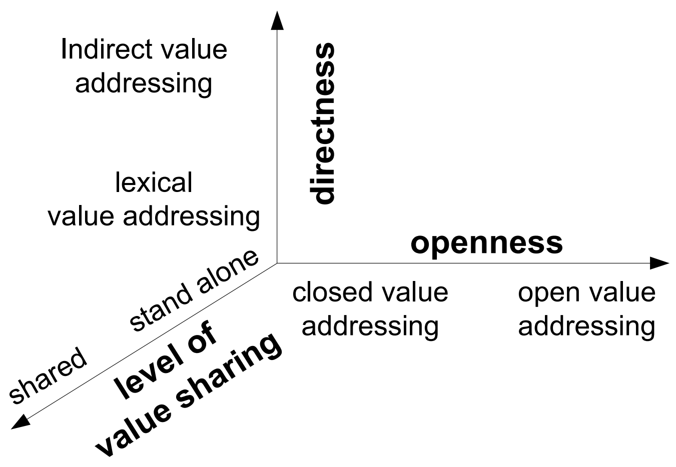


Figure 5-36. Dimensions of join point addressing.

The first three – directness of value addressing, openness and level of value sharing - are related to how the JPSL permits developers to refer to certain properties of join points. Possibly, an aspect-oriented system provides for a number of selection constructs and properties different mechanisms to select them. Correspondingly, it is reasonable to consider the design dimensions for each given construct. For example, it is possible to analyze AspectJ's call pointcut designator, execution pointcut designator, etc. how the corresponding construct maps the design dimensions directness, openness and level of value sharing.

The latter two design dimensions – the adjustability and scope of join point selections – are directed to the selection of join points themselves (and not only to the addressing of a property's value). It seems rather strange if in a given aspect-oriented system the selection constructs differ with respect to these design dimensions[78]. Hence, such design dimensions describe characteristics of the whole JPSL for a given aspect-

[78] In fact, AspectJ is already an example where the selection constructs differ for such dimensions: Class definitions join points (as used in introductions) cannot be incrementally defined, while all behavioral join points can. In respect to this, AspectJ differs from systems such as Hyper/J, Sally or AspectS where all constructs are equal in respect to such design dimensions.

oriented system and not only a characteristic of a certain language construct. For example, it does not make any sense to ask whether the execution pointcut designator in AspectJ is restricted or whether the specification of call pointcut designators can be achieved incrementally. Instead, the set of all pointcut designators can be analyzed in respect to both dimensions.

Figure 5-36 illustrates the design dimensions of value addressing. Every JPSL construct of a given aspect-oriented system can be analyzed with respect to how it matches to the corresponding design dimensions. Similar to the design dimensions of join point properties certain kinds of constructs occur more often in aspect-oriented systems.

Lexical, closed and non-shared constructs are provided by a large number of aspect-oriented systems and can be regarded as the "first generation of JPSL constructs". Examples are constructs that permit to refer to method definition join points only by specifying each method's name. This corresponds for example to the selection constructs of Hyper/J (except the bracket relationships[79]).

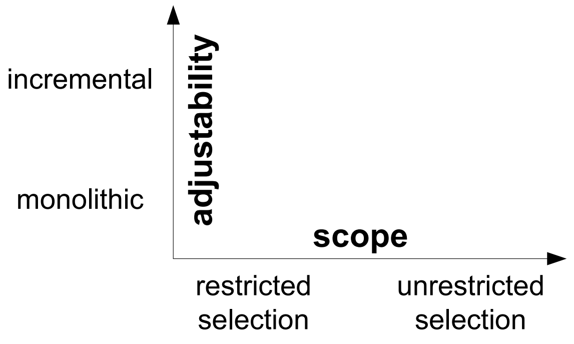


Figure 5-37. Design dimensions for join point selection languages.

An example for an indirect, open and stand-alone construct is the + operator in AspectJ. The + operator refers to join point properties without referring to the corresponding types by their name. Since all subtypes of the specified type are addressed, it is also an open value addressing. However, since it is not possible to specify the same type pattern for a number of different properties, + is a stand-alone construct.

Figure 5-37 illustrates the latter design dimensions for the join point selection. An example for an incremental and unrestricted join point selection language is AspectJ's pointcut language. Hyper/J's selection language is an example of being monolithic and unrestricted.

[79] Since bracket relationships permit to use wildcards for selecting methods this construct provides an open value addressing.

5.4.5 Relationship between Join Point Properties and Join Point Addressing

When designing an aspect-oriented system there is a trade-off between the design of join point properties and the design of join point addressing. In general, it is possible to reduce the number of properties for a given join point and to extend the way how properties can be addressed without reducing the expressiveness of the underlying system's selection language.

For example, AspectC++ provides a property for method call join points that states how many parameters a method has – a static and local property with an abstract correspondence. Such a property is for example not provided by AspectJ for method call join points. However, AspectJ provides a static, local and structured property with a direct correspondence that represents the types declared in a method's signature. Although both join point encodings differ, it is possible to select method call join points because of the number of parameters. In AspectJ this can be achieved by the `..` operator which abstracts over the number of parameters as well as with the * operator which abstracts over a property's value in a lexical way.

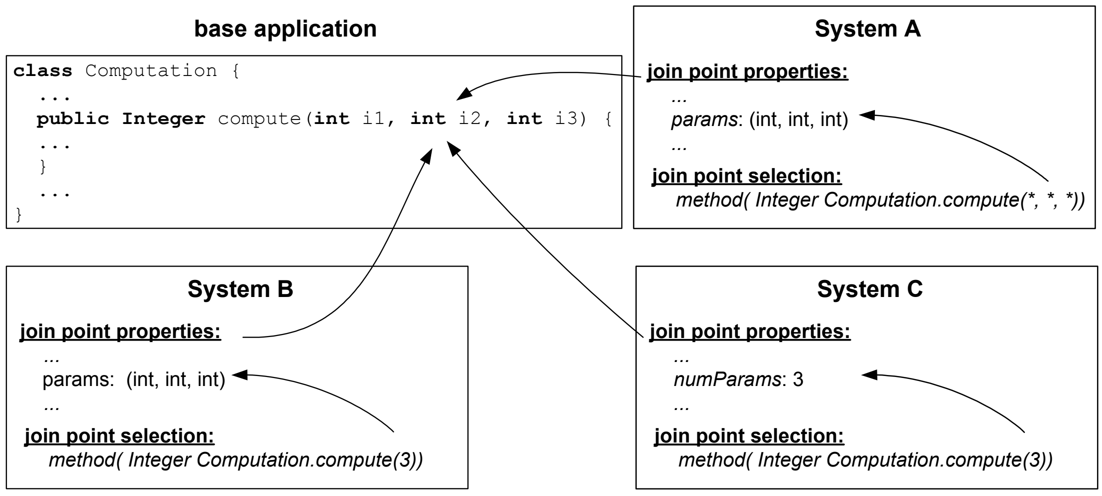


Figure 5-38. Join Point Selection with direct and indirect join point value addressing and lexical and abstract join point properties.

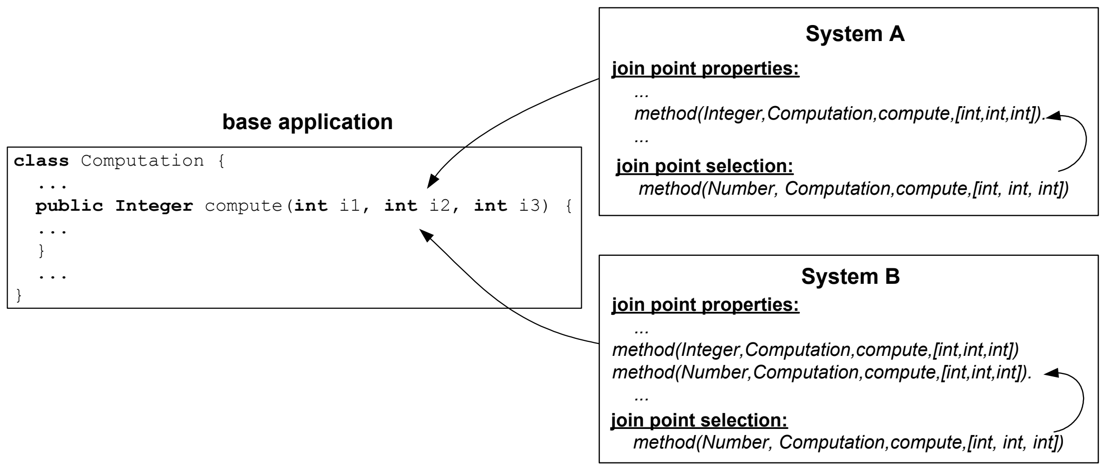
Figure 5-38 illustrates three different systems where each one differs with respect to the provided join point properties as well as with respect to the selection language constructs whereby in all these cases the resulting join point selection (selecting a certain method definition because the number of parameters is 3) is the same.

- System A is an AspectJ-like system that provides a static, local, and structured property with a direct correspondence that represents the types declared in a method's signature. Furthermore, the system provides mechanism for a lexical and open property addressing.
- System B provides the same property as system A. However, the selection construct is indirect (because the value 3 reasons on the number of parameters).

- System C provides directly a property that represents the number of parameters. In order to select the appropriate join points it is sufficient to have a lexical value addressing: The token `3` can be lexically compared to the value of the `numParameters` property.

When analyzing a certain aspect-oriented system, the underlying design of the join point properties might be not transparent. Such a situations occurs where the aspect-oriented system provides only a stand-alone property addressing (which is the case for example in AspectJ or Hyper/J). The problem here is that it is not possible to request the value of a certain property but only to specify constraints on some join point characteristics. In the previous example, system B permitted to specify an indirect selection criterion. However, if the value of the underlying property cannot be requested it is not clear how the underlying property is represented.

Although AspectJ provides only a stand-alone property addressing, it can be derived how the underlying property is represented due to the reflective capabilities of AspectJ: The keywords `thisJoinPoint` and `thisStaticJoinPoint` provide access to each join point and its properties can be retrieved from within the join point adaptation (but not from within the join point selection). Consequently, a number of properties can be derived from this join point representation. However, there are also properties whose representation cannot be derived from such reflective capabilities: The underlying data that is being used in order to permit the selection of elements from the call stack is hidden from the developer[80]. Consequently, the design dimensions of join point properties can be understood in such situations rather as a conceptual framework for the data used to determine whether a join point characteristic holds.


base application
System A
join point properties:
...
method(Integer,Computation,compute,[int,int,int]).
...
join point selection:
method(Number, Computation,compute,[int, int, int])
System B
join point properties:
...
method(Integer,Computation,compute,[int,int,int])
method(Number,Computation,compute,[int,int,int]).
...
join point selection:
method(Number, Computation,compute,[int, int, int])

```
class Computation {
  ...
  public Integer compute(int i1, int i2, int i3) {
  ...
  }
  ...
}
```

Figure 5-39. Join Point Selection based on indirect and direct join point selection.

There are some systems that come with a logical join point language in combination with a number of predicates for join points. Examples for such systems are Andrew

[80] In fact, AspectJ does not provide the whole call stack information in a property but generates some additional runtime checks that determine whether the declared call stack characteristics for a given dynamic join point hold.

[GyBr03] or Sally. In general, such approaches provide for each join point a number of facts as well as a number of rules operating on such fact. In case the underlying system does not provide constructs that distinguish between the facts and the rules applied to such facts, it is not possible to determine directly the characteristics of the underlying join point properties.

Figure 5-39 illustrates two systems that both permit developers to address the same method definition based on a subtype relationship of the method's return type: The methods are selected because the return type is of type `Number` (which corresponds to the method because `Integer` is a subtype of `Number` in Java). System A provides such a selection because the underlying predicate specifies the corresponding subtype relationship. Hence, the selection construct provides an indirect join point addressing. System B directly provides a fact that represents the subtype relationship. From the developer's perspective both join point selections are equal because they both refer to the same join point. However, if no internals about the underlying facts are known, it cannot be determined whether the underlying property has a direct correspondence (and the selection construct provides an indirect addressing) of whether the underlying property has an indirect correspondence (and the selection construct provides a lexical value addressing).

It can be concluded from the previous discussion that there is some interplay between design decisions of join point properties and property addressing: It is possible to replace a design decision of a join point property with a design decision of the join point addressing without changing the system's functionality from the developer's perspective.

5.5 Join Point Adaptation

Once join points are selected, aspect-oriented systems provide language constructs for adapting such join points. However, there are many possibilities that could be done with the chosen join points. This does not only depend on the design decisions of the aspect-oriented systems but also on the semantics of the base language and the chosen join point model. For example, if a selected static join point is a method definition an aspect-oriented system potentially may replace the body of the method or add additional statements to the method body.

Join Point Adaptation Language: The join point adaptation language specifies language constructs for adapting those join points selected by a join point selection.

With respect to the constructs that specify how a join point is to be adapted, different aspect-oriented systems already differ substantially on the first glimpse:

- AspectJ (and closely related systems like AspectS or PROSE) provides at least **advice**, i.e. modules that permit to change the behavior of an application at behavioral join points. Furthermore, AspectJ provides a mechanism of **introduction** that permits to extend class definition join points by adding new members or interfaces.
- Hyper/J provides on the one hand **relationships among hyperslices** that permit to add new structural elements to classes within a hyperslice. Furthermore,

the **bracket relationships** in Hyper/J permit to change the behavior at certain behavioral join points by adding new method invocations.

By studying such mechanisms in more detail it turns out that there are number of similarities among them. First, most approaches have a mechanism that is similar to introductions in AspectJ. Second, most approaches have an advice-like mechanism.

5.5.1 Structural and Behavioral Join Point Adaptation

AspectJ introduces the term introduction (which has been exhaustively discussed in Chapter 3). It implements a mechanism for adding fields, methods, and interfaces to classes. This is similar to **open classes** as described in [Cann82] and [CLCM00]. Introductions have been motivated by the observation that different concerns have a direct impact on the type structure of object-oriented applications. As a consequence, modularization is compromised since some elements in the type structure, like certain fields and methods, come from different concerns. In Hyper/J, AspectJ-like introductions can be implemented by defining classes that contain the members to be introduced and by specifying within the hypermodule how to weave the participating classes.

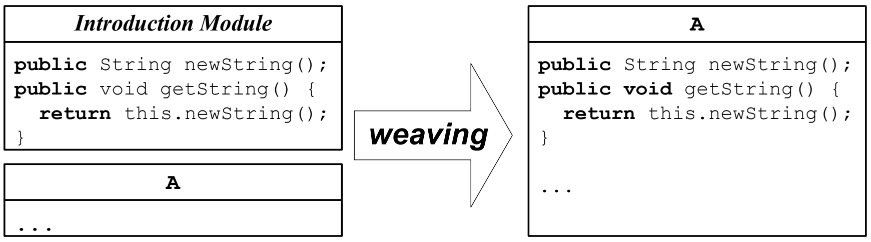

Introduction Module
public String newString();
public void getString() {
return this.newString();
}
A
...
weaving
A
public String newString();
public void getString() {
return this.newString();
}
...

Figure 5-40. Introducing methods newString and getString to a target class A [HaUn03a].

In general, both systems contain mechanisms that permit to achieve a similar result: Introducing members into a class definition join point. Figure 5-40 illustrates an introduction without referring to a concrete aspect-oriented system. A special introduction module defines new members (method `newString` and `getString`) that are to be introduced into a target class `A`. The aspect-oriented system takes the introduction module and the target class and weaves them together. Thus, all elements of the introduction module become members of the target class `A`. An introduction is a strictly type increasing operation on types since it adds new features to types but does not permit to remove anything. The intention of applying introductions is to change the way that the system possibly interacts with certain types. Typically, introductions are performed by aspects that later on themselves access the introduced members.

On a more abstract level, an introduction is a composition mechanism that refers to a structural join point and that adapts the target join point in a way that only its structure changes. Introductions do not change the way how the application is executed[81], hence the semantics of the resulting (woven) application stays the same[82].

However, aspect-oriented systems also potentially provide an advice-like mechanism that changes the behavior of the base application without changing the structure of the application. Figure 5-41 illustrates the specification of an advice before and after weaving in AspectJ. The around advice changes the base application at the specified join point but it does not have any impact on the structure of the join point: The structural appearance of the method stays the same[83]. Similar results can be achieved (but just for static join points) in Hyper/J by using bracket relationships.

Obviously the means of how join point are adapted using advice and introduction substantially differ. The main difference is that an introduction is a structural adaptation while an advice is a behavior adaptation. Based on this observation, this thesis identifies the design dimension **level of adaptation** and distinguishes between **structural join point adaptation** and **behavioral join point adaptation**.

Structural join point adaptation: An aspect-oriented system provides strucural join point adaptations if there are constructs that permit to change the structure of the join points.

Behavioral join point adaptation: An aspect-oriented system provides behavioral join point adaptations if there are constructs that permit to change the behavior of the join points (without changing the join points' structural appearance).

In the easiest case, a structural adaptation is the attachment of members to class definitions without interfering the lookup process of the objects of this class[84]. Under the same restriction, adding a new method to an object in a prototypical language represents a structural join point adaptation. Both adaptations have in common that the corresponding join point is a structural join point.

Although the (static) structural join points described in the examples above are class definitions, there is (on the conceptual level) no reason why other kinds of structural join points should not also be used for the same kind of composition. For example, an introduction-like mechanism for method definition join points could be the attachment of optional parameters[85]. Obviously, attaching optional parameters to a method definition join point does not change the behavior of the application as long as the

81 In fact even introductions in AspectJ possibly change the behavior of the woven application. However, this is rather regarded as an implementation failure (see [Stör03] for further discussion).

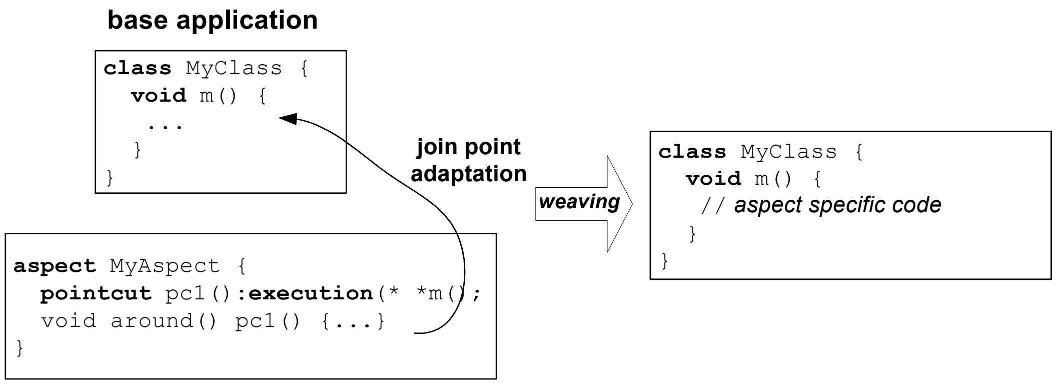
82 Of course, this assumes that the underlying application does not depend on **structural reflection** [Maes87], i.e. the way how the application is executed does not depend on the structure of the application itself.

83 This view on the woven code in AspectJ is quite simplified since on the bytecode level there are a number of structural changes on the class the target method is defined in (see [HiHu04]).

84 If a method would be added to a class and overrides a method defined in a superclass, this potentially changes the behavior of the application because the overriding method would be invoked instead of the overridden one.

85 However, programming languages like C++ and Java currently do not provide optional parameters while this is common for a large number of languages like LISP or TCL.

lookup mechanism is not interfered and as long as there is no code accessing such optional parameters. In the same way, a method definition join point in systems like Java could be adapted in a way that declared exceptions are to be deleted from the method definition. It is also imaginable that structural join point adaptations are applied to behavioral join points like for example transforming a method call to a behavior equivalent expression. This could be achieved by providing language constructs that inline the target method directly. Although possible, this thesis considers such structural join points adaptations applied to behavioral join points as composition mechanisms that play design rather a subordinate role in aspect-oriented system [86].



```
class MyClass {
  void m() {
   ...
  }
}
```

```
aspect MyAspect {
  pointcut pc1():execution(* *m());
  void around() pc1() {...}
}
```

```
class MyClass {
  void m() {
    // aspect specific code
  }
}
```

Figure 5-41. Adapting the behavior of a base application by using advice in AspectJ.

Behavioral join point adaptations are in the easiest case adaptations of behavioral join points (like advice in AspectJ and AspectS or bracket relationships in Hyper/J). For example, a method call that is replaced by a different one represents a behavioral join point adaptation. Of course, such a replacement could lead - after weaving - to an application that is still behavior equivalent to the unwoven application, if the replacing method is behavior equivalent to the original method call. However, behavior join point adaptation just refers to the existence of language constructs for specifying join point adaptations that possibly change the application's behavior: It is not necessary that by applying a specific adaptation the resulting application behaves actually different. In contrast to structural join point adaptations, it is quite common that behavioral join point adaptations are applied to structural join points. For example, Hyper/J replaces method definition join points by corresponding override relationships.

In current aspect-oriented systems that are based on object-oriented base languages, behavioral adaptations are usually applied to method call join points or method definition join points. But in principle it is also possible that behavioral join point adaptations can be applied to any arbitrary behavioral join point like for example the application of operators.

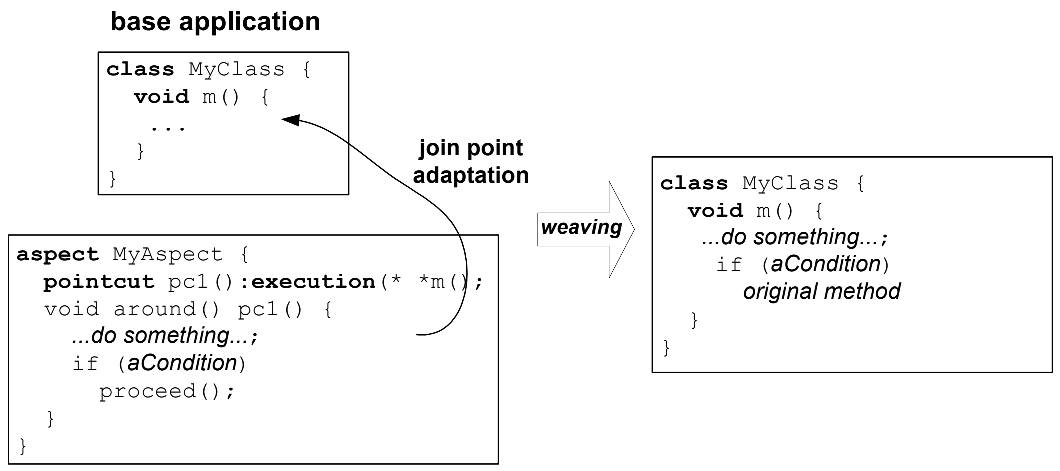
[86] In fact, altough its theoretically possible, there is no system known to the author that provides such a kind of adaptation.

5.5.2 Constructive and Destructive Join Points Adaptation

One characteristic of advice in AspectJ (and also bracket relationships in Hyper/J) is that they permit to change the behavior at behavioral join points. Such advice permit to specify that the behavior to be added potentially occurs before or after the join point. The interesting observation with these kinds of behavior adaptations is, that the original join point stays the same because just additional behavior is executed at these join points. Of course, this does not means that such additional behavior does not interfere with the behavior specified in the base application. It simply states, that the original join point is still (potentially) part of the application.

There is a parallel with the previously discussed introductions: Introductions add something to the static class definition join point while the join point after weaving still contains all elements that existed before weaving: The type being the target of the introduction is still compatible with the original type.

However, around advices that are available in AspectJ and AspectS slightly differ from that: Around advices potentially replace the original join point with the elements specified within the advice. Furthermore, AspectJ as well as AspectS provide a construct that permits the execution of the original join point. In AspectJ such construct is the `proceed` keyword; in AspectS it is realized by invoking the original method extracted from the context object. The difference between such a proceed-like construct and a before or after advice is that it is possible from within the adaptation specification to refer to the original join point. Before and after advice simply represent a form of build-in mechanism where users of the aspect-oriented system cannot specify on their own how the original join point is being used. In contrast to that, proceed-like constructs permit developers to specify on their own how and when the original join points is being used.



```
class MyClass {
  void m() {
    ...
  }
}
```

```
aspect MyAspect {
  pointcut pc1():execution(* *m();
  void around() pc1() {
    ...do something...;
    if (aCondition)
      proceed();
  }
}
```

```
class MyClass {
  void m() {
   ...do something...;
    if (aCondition)
      original method
  }
}
```

Figure 5-42. Behavioral join point adaptation that refers to the original join point.

Figure 5-42 illustrates schematically a join point adaptation before and after weaving where the adaptation specification explicitly refers to the original join point. The aspect `MyAspect` contains an around advice that refers to method `m` defined in class `MyClass`. The developer of the aspect decided to execute the original join point only if some runtime condition is fulfilled. In case this condition is not fulfilled, an execution of method m only executes the behavior that is additionally specified inside the advice, but the original behavior is not executed.

Hyper/J also provides an override relationship: Overriding one method with a different one simply replaces the original method definition. There is no possibility to refer from the overriding method to the original implementation.

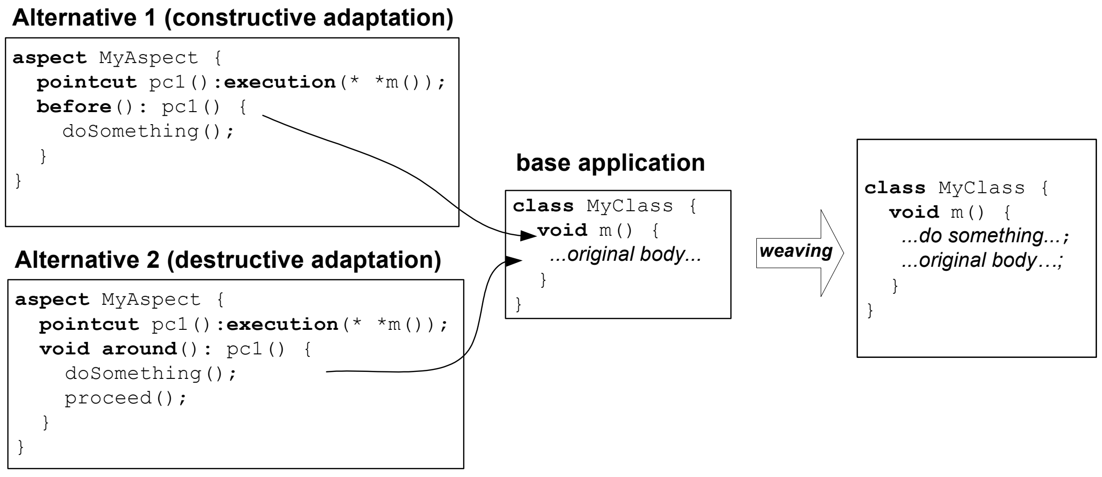
Based on the previous discussion this thesis identifies **constructiveness of join point adaptation** as a separate design dimension for join point adaptation and distinguishes between **destructive join point adaptation** and **constructive join point adaptation**.

Destructive join point adaptation: An aspect-oriented system provides destructive join point adaptations if the adaptation necessarily replaces the join points it refers to.

Constructive join point adaptation: An aspect-oriented system provides constructive join point adaptation if the adaptation does not necessarily change the join points it refers to in a way that they no longer exists after weaving.

The simplest kind of a destructive join point adaptation is the replacement of method bodies from the base application with new bodies specified in the adaptation specification. This corresponds to the override relationship in Hyper/J. The particular characteristic of destructive adaptations is that the original join point is replaced by a new one.

Constructive join point adaptations simply permit to change the join point in a way that the join point after weaving still exists. Before and after advice as well as introductions in AspectJ are typical examples of constructive join point adaptations.


Alternative 1 (constructive adaptation)
Alternative 2 (destructive adaptation)
base application
weaving

```
aspect MyAspect {
  pointcut pc1():execution(* *m());
  before(): pc1() {
    doSomething();
  }
}
```

```
aspect MyAspect {
  pointcut pc1():execution(* *m());
  void around(): pc1() {
    doSomething();
    proceed();
  }
}
```

```
class MyClass {
  void m() {
    ...original body...
  }
}
```

```
class MyClass {
  void m() {
    ...do something...;
    ...original body...;
  }
}
```

Figure 5-43. Equivalent adaptation using a constructive or destructive adaptation.

As argued above, language constructs like around advices in AspectJ also permit to refer to the original join point. Hence, on the one hand they replace the original join point, but on the other hand they still give developers the ability to refer to the replaced join point from within the join point adaptation. From the user's perspective, a system that provides destructive join point adaptations can express the same as constructive adaptations as long as the adaptation refers to the original join point.

Figure 5-43 illustrates schematically two different (behavioral) adaptations in AspectJ. The first alternative specifies the adaptation using a constructive adaptation using a before advice in AspectJ to add certain behavior. The second alternative specifies the adaptation using a destructive around advice. From the application's semantics perspective both join point adaptations lead to the same application as illustrated in Figure 5-43[87]. This means, in general a destructive join point adaptation is not less powerful in comparison to constructive adaptations: It also depends on the existence of constructs that permit to refer to the original join point that is replaced by the adaptation.

In comparison to Hyper/J such a language feature differs essentially, because the join point adaptation itself can determine whether or not the join point is to be removed. Hence, this thesis does not only distinguish between destructive and constructive adaptations but also on adaptations that are potentially constructive by allowing to decide within the adaptation whether or not the join point is to be replaced.

Conditional constructive join point adaptation: An aspect-oriented system provides a conditional constructive join point adaptation if within the adaptation module it can be determined whether the target join point is to be destructed.

Conditional constructive join point adaptations are for example around advice in AspectJ that refer to the original join point via `proceed` as well as wrappers in Sally that refer via `wrap` to the original join point.

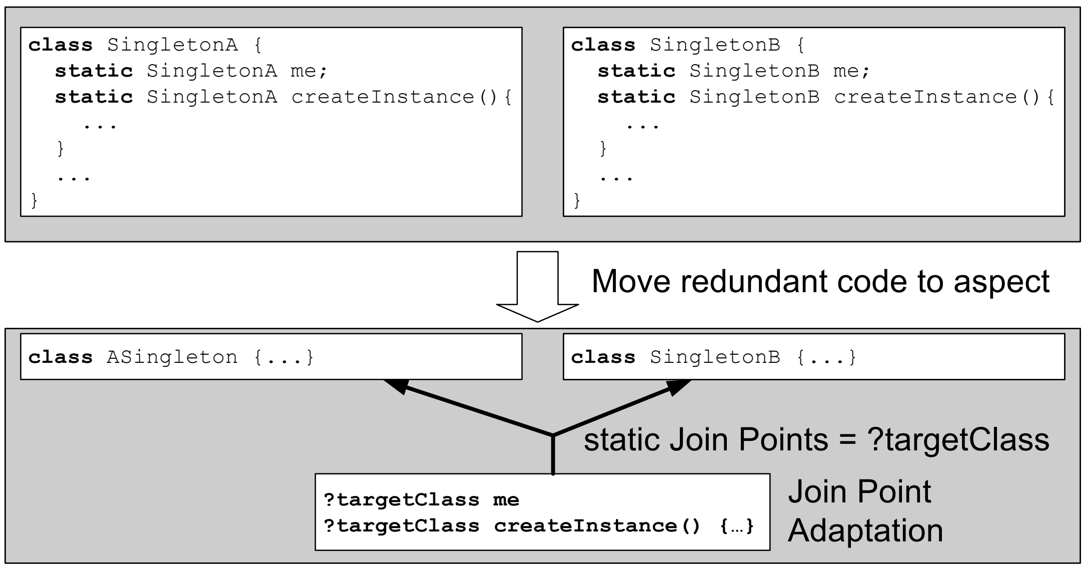
There are some parallels between the conditional constructive adaptation of dynamic join points and different kinds of **inheritance** [Taiv96] in object-oriented systems (see also [HaUn01a]). Inheritance is often regarded as a kind of **incremental modification** [WeZd88] of the superclass where method overriding is one special kind of incremental modification. Furthermore, in most object-oriented languages the newly defined method is able to refer to the method which is about to be overridden by using the `super` keyword. Such overridden methods with super references are quite similar to conditional constructive join point adaptations: The newly defined behavior potentially refers to the old definition by invoking the overridden method via a super-reference. Hence, the new behavior is also defined in terms of the old (incremented) behavior. However, obviously there are also some differences: Method overriding in object-oriented programming is achieved by creating a new class which overrides a number of methods. I.e. the "join point" is in fact a method definition for an object created from a new class (and not a class that already exists in the system). Hence, simply creating a new class and refining a method does not change anything in the base application at all as long as inside the "base application" such new classes are not used.

[87] The woven application is just an illustration of the woven application in AspectJ. In fact, AspectJ generates much more elements like additional methods, etc. (see [HiHu04] for more details).

5.5.3 Non-Variable and Parametric Adaptation

One characteristic of aspect-oriented systems like AspectJ or Hyper/J is that their adaptation language permits to specify adaptations in a way that the code of the adaptation is fix. I.e. all elements of the code are statically defined and do not contain any variabilities with respect to the code. If the base language is a statically typed language like AspectJ or Hyper/J, it is possible to assign static types to all elements in the adaptation and typechecking can be performed in a straight forward way.

However, there are some good reasons why aspect-oriented systems should be able to handle a more relaxed form of adaptation specification: Adaptations that contain also variables. One typical example that illustrates the need for variabilities in the join point adaptations can be directly derived from the singleton design pattern [GHJV95][88]. Figure 5-44 illustrates two classes both making use of the singleton design pattern. Obviously, both classes contain some kind of crosscutting: The singleton specific static field and method occur in more than one place. A first idea could be to modularize the singleton specific elements within a join point adaptation (the corresponding join point selection addresses the static class definition join point). However, in languages like AspectJ or Hyper/J it is not possible to modularize the singleton specific elements.



```
class SingletonA {
  static SingletonA me;
  static SingletonA createInstance(){
    ...
  }
  ...
}
```

```
class SingletonB {
  static SingletonB me;
  static SingletonB createInstance(){
    ...
  }
  ...
}
```

```
class ASingleton {...}
```

```
class SingletonB {...}
```

```
?targetClass me
?targetClass createInstance() {…}
```

Move redundant code to aspect
static Join Points = ?targetClass
Join Point
Adaptation

Figure 5-44. Moving code redundancies from singleton implementation to aspects by using a parametric join point adaptation.

As shown in Chapter 3, such a kind of adaptation is possible in Sally (other examples for systems that provide such a variability in the join point adaptation is LogicAJ, cf. [KRH04]). Figure 5-44 illustrates a possible syntax for a join point adaptation (closely related to Sally) where some elements (in this example the types) are variable and are replaced when the corresponding aspect is woven.

88 The ingredients of a typical Singleton implementation are already discussed in section 3.3.1 as well as in section 5.4.4.3 and are therefore not repeated here.

Since the impact of providing constructs like Sally seems to be essential (because the existence of such constructs determines whether or not certain crosscutting concerns can be modularized), it is reasonable to describe the existence of such constructs as a design dimension on its own. Hence, this thesis identified the **level of variability** as a design dimension for join point adaptation and distinguishes between **non-variable join point adaptation** and **parametric join point adaptation**.

Non-variable join point adaptation: An aspect-oriented system provides non-variable join point adaptations if the code that adapts each join point does not permit to contain any variabilities.

Parametric join point adaptation: An aspect-oriented system provides a parametric join point adaptation if there can be variabilities inside the code representing the join point adaptations.

Almost all current and popular aspect-oriented systems provide non-variable join point adaptations. In AspectJ for example, the body of an advice is very similar to the body of a method definition in Java (neglecting the new keywords `proceed`, `thisJoinPoint`, and `thisStaticJoinPoint`).

Parametric join point adaptations currently occur not that often in aspect-oriented systems. The approach of **aspect-oriented logic metaprogramming** as proposed in [DVDH99] can be regarded as a form of parametric join point adaptation[89].

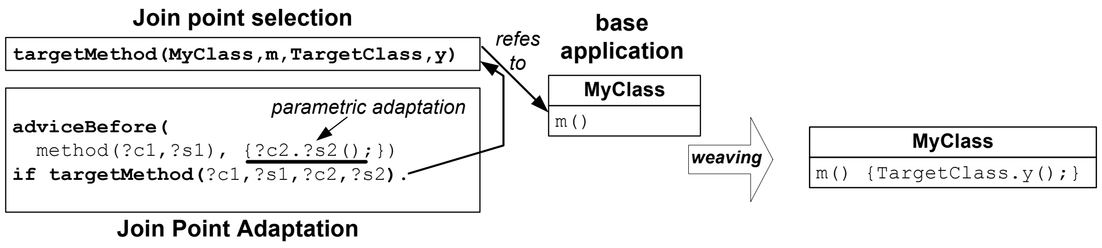


Figure 5-45. Parametric join point adaptation similar to the SOUL/Aop system as described in [BMDV02].

The approach described in [BMDV02] which makes use of the previously mentioned approach [DVDH99] can also be regarded as an approach supporting parametric join point adaptations[90]: The (behavioral) join point adaptations possibly contain variables which are "filled" with parameters from the corresponding pointcut.

Figure 5-45 illustrates a parametric join point adaptation similar to the one described in [BMDV02]. The join point selection (which is specified in a different place) refers to

[89] Although the approach as described in [DVDH99] rather resembles of a logic-based code generator than a system that matches the here proposed design dimensions.

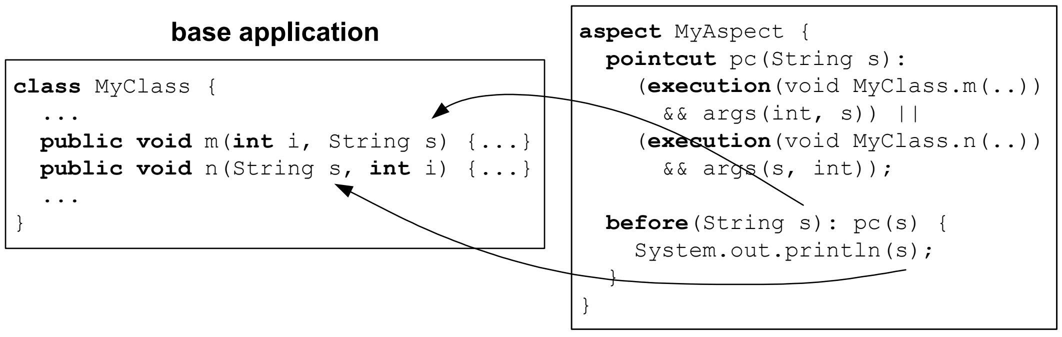
[90] In constrast to [DVDH99] it has a clear static join point model where the corresponding join points are method definitions. However, it should be noted that the intention of [BMDV02] is not to provide a single general-purpose aspect language, but to provide a framework for implementing aspect-specific languages.

method `m` in class `MyClass`. The join point adaptation refers to that method. The term `?c2.?s2()` delimited by the curly braces in the construct similar to advice in AspectJ represents the (constructive) join point adaptation. In the example, the term consists of two variables `?c2` (for a class) and `?s2` (for a selector, i.e. a method name) which are specified within the join point selection (and they are bound to the class `TargetClass` and method `y`). The term represents a method call[91]. However, the join point adaptation does not determine in what class what method will be invoked: This depends on the binding of the variables.

Another example of an aspect-oriented systems providing parametric adaptations is – of course – Sally (see Chapter 3) since the need for a variable code specification within introductions (and advice) has been the main problem addressed by Sally.

5.5.4 Variable and Fix Join Point Abstraction

Systems like AspectJ provide the ability to expose some context to the advice by specifying parameters within the advice and pointcut header: Inside a pointcut declaration, (dynamic) pointcut designators refer to these parameters and permit to specify additional constraints on them (like additional type constraints by means of `this` or `target` pointcut designators or Java-defined constraints with the help of the `if`-pointcut). Within the join point adaptation (i.e. within the advice) the code refers only to the context exposed by the selection, to local variables within the aspect or to global variables. I.e. the values to be used within the join point adaptation are defined (among others) by the corresponding join point selection. Consequently, the data available in order to adapt a certain join point possibly varies from adaptation to adaptation.



```
class MyClass {
  ...
  public void m(int i, String s) {...}
  public void n(String s, int i) {...}
  ...
}
```

```
aspect MyAspect {
  pointcut pc(String s):
    (execution(void MyClass.m(..))
      && args(int, s)) ||
    (execution(void MyClass.n(..))
      && args(s, int));

  before(String s): pc(s) {
    System.out.println(s);
  }
}
```

Figure 5-46. Abstraction of join point context in AspectJ.

The situation is different in systems like AspectS (but also for Sally). There, each join point provides a list that represents the list of parameters passed to it. Consequently, if for example a single adaptation refers to two method execution join points with two different shadow join points, and the adaptation needs to refer to the method's first

[91] The illustration was slightly adapted to fit a Java-like syntax. In the original paper, the base language used by the authors was Smalltalk. Hence, the corresponding term it the system desribed in [BMDV02] would be `?c2 ?s2`, where `?c2` describes the target object (which is potentially a class object) and `?s2` desribes the message selector.

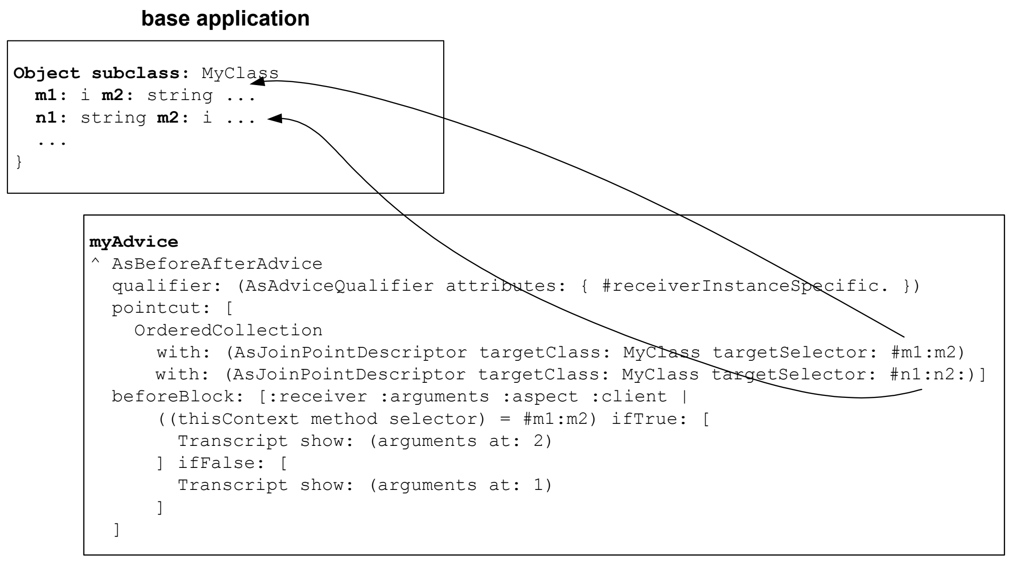
parameter and the method's second parameter, it is up to the developer to select the appropriate parameter within the adaptation.

Figures 5-46 and 5-47 illustrate the difference between the context handling in AspectJ and AspectS. In the example, both (constructive) join point adaptations simply print out the string passed to the method in class `MyClass`.

In the AspectJ example, the join point selection already refers to the parameter of interest and passes it to the join point adaptation. The join point adaptation refers to the passed parameter in order to print it out.

In the AspectS example, the join point selection refers to the same methods (written in Smalltalk). However, the join point adaptation receives for all join points the same parameter list. Hence, the developer of the adaptation needs to extract on his own the right parameter by reflecting on the current join point (using the keyword `thisContext`) in order to print out the right parameter.

Both kinds of join point adaptations have a large impact on the resulting aspect. In AspectJ it is possible to specify the adaptation first and later on a selection that refers to the appropriate join points. Hence, the system provides for the adaptation some abstraction over join points and permits in that way to reuse the adaptation in a number of different situations.



```
Object subclass: MyClass
  m1: i m2: string ...
  n1: string m2: i ...
  ...
}
```

```
myAdvice
^ AsBeforeAfterAdvice
  qualifier: (AsAdviceQualifier attributes: { #receiverInstanceSpecific. })
  pointcut: [
    OrderedCollection
      with: (AsJoinPointDescriptor targetClass: MyClass targetSelector: #m1:m2)
      with: (AsJoinPointDescriptor targetClass: MyClass targetSelector: #n1:n2:)]
  beforeBlock: [:receiver :arguments :aspect :client |
      ((thisContext method selector) = #m1:m2) ifTrue: [
        Transcript show: (arguments at: 2)
      ] ifFalse: [
        Transcript show: (arguments at: 1)
      ]
  ]
```

Figure 5-47. Selecting the appropriate context in AspectS.

In AspectS the adaptation needs to reflect on the join points itself in order to determine the right parameters. Consequently, the join point adaptation is closely related to the selected join points and it is hard to combine a once specified join point adaptation with different join point selections. For example, if execution join points of other methods should be adapted (where the 3rd or 4th parameter needs to be printed out) it is necessary to change the join point selection as well as the join point adaptation.

This thesis identified the **level of join point abstraction** as another design dimension of aspect-oriented systems and distinguishes between a **fix** and **variable join point abstraction**.

Fix join point abstraction: An aspect-oriented system provides a fix join point abstraction if the join point adaptation's context is fix for all join points of a given concrete join point class.

Variable join point abstraction: An aspect-oriented system provides a variable join point abstraction if the developer can specify on his own the join point's context the adaptation operates on.

In principle, the ability to provide a context to the join point adaptation can be considered as a more mature mechanism of aspect-oriented systems than a fix join point abstraction because mechanisms are needed that extend the adaptation context with additional runtime entities. Consequently, more recent approaches (cf. for example [NCT04]) investigate the ability to specify a larger variety of contexts for join point adaptations.

5.5.5 Orthogonality of Design Dimensions

The previously identified design dimension are orthogonal to each other and join point adaptation mechanisms can be analyzed with respect to all dimensions independently of each other (see Figure 5-48).

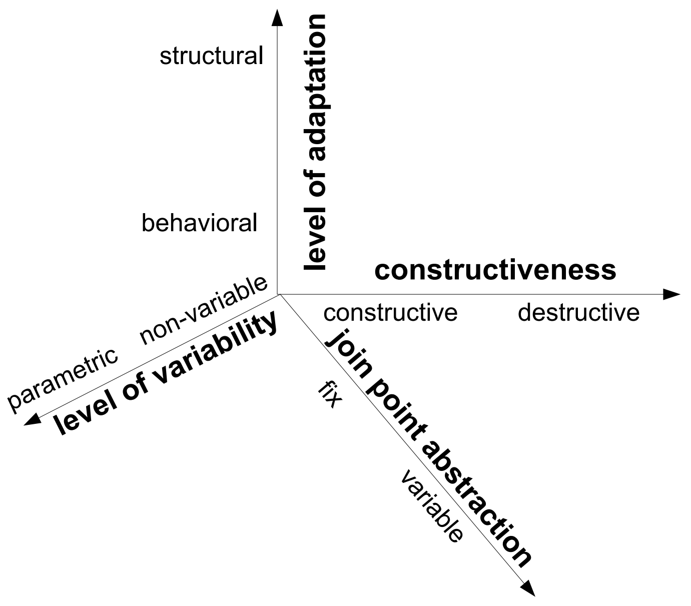


Figure 5-48. Orthogonal design dimensions of join point adaptation.

Similar to the section 5.4.4.6 the design dimensions represent the underlying design space for different kinds of join point adaptations. It is possible that an aspect-oriented system provides language constructs for different kinds of adaptations where each one is based on different design decisions. For example, AspectJ provides with introductions and advice two different kinds of adaptations where each one is based on different design decisions (introductions are structural adaptations while advice are behavioral adaptations). In Hyper/J there are even more implementations of the design dimensions due to its large number of composition rules.

A special attention should be paid to the orthogonal dimensions level of variability and join point abstraction. From the first glimpse it looks like both dimensions describe the same (or at least a similar) situation because both refer to a kind of parameterization of join point adaptations. However, the difference is that while the level of variability focuses on variable code specifications, the join point abstraction determines the runtime context in which the adaptation is executed. The orthogonality of both dimensions can be shown by examples of possible combinations. AspectS provides by its advice a fix level of variability and also a fix join point abstraction. AspectJ provides via pointcuts and advice a fix level of variability and a variable join point abstraction. Sally is an example for a parametric system with a fixed join point abstraction. Sally-like systems that also provide a kind of context exposure (for example, LogicAJ is such a system, cf. [KRH04]) provide a parametric join point adaptation with a variable join point abstraction.

It seems as if the **level of variability** does not change between different kinds of adaptations in the same aspect-oriented system. With respect to AspectJ, it seems natural that if someone constructs an AspectJ extension supporting parametric adaptation the parametric adaptations are then provided for each kind of adaptation. For example, the approach of LogicAJ [KRH04] is closely related to AspectJ and provides a parametric adaptation within introductions as well as in advice.

Similar to the design dimensions of join point properties and join point addressing, there are some combinations of design decisions that occur more often than other. For example, **behavioral join point adaptations** with a **non-variable level of variability** are provided by a large number of systems (like AspectJ, AspectS, Prose, etc.). Reasons for this might be the probably rather easy way to implement such adaptations via simple code transformations.

5.6 Weaving

Weaving describes the process of integrating aspects into the base application. However, likewise to the previous section, there is a number of different interpretations of the weaving process. While this thesis considers join points, join point selections, and join point adaptations as concepts aspect-oriented systems are based on, weaving is rather considered as an implementation detail. I.e. the semantics of the woven application should be only determined by the base application, the join point model, the join point selection, and the join point adaptation – and not by the way aspects are woven.

Nevertheless, for comparing different aspect-oriented systems it might be crucial in some situations to know what weaving technique is used, and for implementing a new system the decision of what kinds of weaving are to be provided has a large impact on what techniques are being used (like for example building a new interpreter, a new compiler, preprocessor, etc.).

It is observable that there are recurring ways of implementing weavers in aspect-oriented systems. Hence, it is consequent to describe the different design alternatives underlying weavers in the same way as for the other ingredients of aspect-oriented systems – by corresponding design dimensions.

Originally, the term weaver and weaving was introduced in [KLM+97] which states *that a weaver accepts the component and aspect programs as input, and emits a (…) program as output.*

This view on weaving is closely related to **generative programming** [CzEi00] where an output program is generated from a number of input programs. However, [KLM+97] also states that there are different alternatives to realize such an integration of aspects into the base program: *On the one hand weavers might work at runtime or at compile time*. This makes the view on weaving slightly inconsistent because it is not clear whether the underlying idea of a weaver is for example a program generator that operates on the application's syntax or a **metaobject protocol** [KDRB91], which permits to adapt metaobjects at runtime. In the same way it would be possible to regard a weaver as an **interpreter** (see for example [StSu78] for an example discussing the construction of interpreters).

However, even using the term weaving for describing the "process of integrating aspects" is not clear: The term weaving is also used for example in [KCA04] to describe the bytecode modifications at load-time in the programming language Java to provide an infrastructure for adding aspects dynamically ([KCA04] speaks about *load-time weaving for injecting hooks that enable run-time weaving*). In contrast to the original interpretation of the term, this kind of load-time weaving neither affects an application's behavior nor (or hardly) does it affect an application's structure in any way. So, this kind of weaving does not integrate an aspect in the system. Instead, hooks are inserted in each method that forward each message to another module (object or class), which decides what to do at each point[92]. This permits to integrate aspects into the base system later on.

To handled even such techniques with a common definition of weaving, this thesis proposes the following definition:

Weaving: Weaving is the process that creates a connection between the base system and the aspect-oriented system.

This definition does not emphasize the integration of aspects; the integration of concrete aspects into the base system might be either achieved by the weaver itself or by an infrastructure, which the weaver connects with the base system.

5.6.1 Static and Dynamic Weaving

The first criterion for classifying weavers can be directly extracted from [KLM+97]: The criterion of **weave time**. This criterion simply determines the point in time when a weaver performs its work. According to [PGA02, PGA03] applying this criteria divides aspect-oriented systems into two classes: Systems that perform **dynamic weaving** and **static weaving** (the same terminology is used for example in [CBE+00, Rash02]).

Probably the most intuitive example of a dynamic weaver is a language that permits to change metaobjects during runtime. An example of such a language is Smalltalk that permits to change for example compiled methods which have an object-oriented representation during runtime (see sections 2.4, especially 2.4.3). In such a situation the programming language's metaobject protocol is the weaver and weaving is the replacement or configuration of metaobjects.

[92] The mechanism for forwarding messages is an implementation of the **message redirector** design pattern (cf. [Zdun01]). A similar approach can be found in the **aspect moderator framework** [CBE00].

A number of authors (like for example [Aßma03, CBE+00, PFFT02, Rash02]) consider static weaving and **compile-time weaving** to be equal. However, the problem with this interpretation of the weaving process is that there are languages that either do not have any compile-time (which is the case for most scripting languages). Furthermore, languages like Java also provide an additional **load-time** which refers to the time when elements from the application's sources (or bytecode in the Java example) are being loaded. On the one hand load-time seems to be closer to run-time than to compile-time because the application is already running. From this point of view adapted class loaders in Java like for example the **Binary Component Adaptation** (BCA, [KeHo98]) would represent a kind of dynamic weaving. However, according to the Java Language Specification ([JSGB00], p. 248), a once loaded class can only be unloaded if a number of very restrictive conditions are fulfilled (see for example [McBa98] for a more detailed discussion). As a consequence, load-time approaches permit to modify a class when it is loaded; afterwards it is (hardly) possible to modify the once loaded class. Hence, if load-time class transformations are applied as a technique for implementing an aspect-oriented system upon a base system, the infrastructure for integrating aspects can be added but not removed. From this point of view, load-time bytecode transformation is rather a technique for static weaving because the decision whether or not a class is adapted is a final one (even if decided at runtime) and cannot be revised after the transformation is performed.

Based on the previous discussion, this thesis identifies the **dynamicity of weaving** as a design dimension of weaver design and distinguishes between **dynamic weaving** and **static weaving**.

> **Dynamic weaving:** Dynamic weaving describes the ability of an aspect-oriented system to create and remove the connection between the base system and the aspect-oriented system at an application's runtime.
>
> **Static weaving:**. Static weaving describes every kind of weaving where the decision whether or not to connect a particular part of the base system to the aspect-oriented system is a final decision and cannot be revised afterwards.

The design dimension does not explicitly refer to load-time weaving. This is because load-time is a term closely related to systems like Java and can hardly applied to other programming languages. Furthermore, since load-time approaches have the characteristic that a transformed class cannot be changed once more during runtime it is reasonable to characterize such an approach as static weaving.

5.6.2 Code Instrumentation and Interpretation

Instead of distinguishing weaving in terms of its dynamicity, it is also possible to distinguish systems with respect to the underlying technique that implements the weaver. In general, there are two different approaches.

- A weaver can transform the base system's code by inserting appropriate hooks that execute aspect-specific code by inserting the aspect-specific structures directly into the underlying code base. In fact, most of the current approaches (like AspectJ, Hyper/J, AspectS, and Sally) perform such a code instrumentation. One characteristic of such approaches is that the runtime environment of the underlying base language does not need to be changed.

- Weaving can also be achieved by an interpreter that contains the functionality for performing the connection to the aspect-oriented system. This can be done via a new interpretation of the base application that implies the same semantics of the non-adapted join points and that provides the corresponding semantics of the adapted join points. One system performing a new interpretation of the base code is for example Prose [PGA02].

From the implementation's point of view both techniques are quite different. In the first case a preprocessor is sufficient (or the transformation of some underlying metaobjects in case a corresponding meta-object protocol is available). In the latter case an adaptation of the runtime system is necessary. Consequently, this thesis identified the **level of adaptation** as a design dimension of weavers and distinguished between **code instrumentation** and **code interpretation**.

Code instrumentation: A weaver is based on code instrumentation if the underlying code base is transformed to adapt join points. I.e. the way how the base system is to be interpreted stays the same, but additional constructs written in the same language as the base system establish a connection to the aspect-oriented system.

Code interpretation: A weaver is based on code interpretation if weaving is achieved by a new interpretation of the underlying code base, i.e. the weaver is an interpreter that permits on the one hand to interpret the base system, but performs the join point adaptation by a new interpretation of the corresponding join points.

One characteristic of weavers based on code instrumentation is that the finally woven application has a different appearance than the original one – additional statements or additional structural elements like classes or methods need to be added to the base application, which are written in the same language as the base language. In the presence of dynamic behavioral join points this code potentially includes **join point checks** [HHU04] or **join point residues** [HiHu04] that determine for each adapted join point shadow whether a selection criteria is fulfilled.

5.6.3 Dynamic Weaving: User-Driven and System-Driven Weaving

Dynamic weaving can be provided by different underlying techniques. First, dynamic weaving could be a technique for improving a system's performance: The intention of morphing aspects is to adapt join points just at the moment when they are needed. Consequently, the number of join point checks performed at runtime is less than in comparison to a static weaver, since join point adaptations are performed just in time (see Chapter 4). The intention of morphing aspects is to keep the same semantics of the woven application than static weaving.

In contrast to that, AspectS also permits a completely different approach. Instead of specifying the relationships between join points upfront it is possible that weaving is performed by a user's request. For example, instead of providing a persistency aspect which stores all objects fulfilling certain criteria into a database, it is possible that the system provides some interaction with the user which determines whether or not an aspect should be woven – like a graphical user interface that explicitly permits users annotate objects to be persistent.

Technically, both kinds of weaving are the same – both are performed just at runtime. However, there is a conceptual difference: The first approach has already specified all criteria under which an aspect is to be woven. The system needs to determine whether or not a join point is reached that requires to adapt or release join points. In the latter case the application leaves it up to the user of the system to determine whether or not an aspect is to be woven: The join point selection is conceptually performed (at least partly) by the application's user. However, it is not necessary that this kind of weaving is restricted to the selection of join points – possibly the application also permits means to let the user specify how to adapt join points (like letting the user choose what data storage is to be used).

Consequently, this thesis considers the **weaving stimulus** as a design dimension of dynamic weaving and distinguished between **user-driven** and **system-driven** weaving.

User-driven weaver: A dynamic weaver is a user-driven weaver if the set of join points that are to be adapted or the way of how join point are to be adapted is explicitly determined be the user of the base application. I.e. some stimulus defined by the application's user is needed in order to determine the semantics of the application.

System-driven weaver: A dynamic weaver is a system-driven weaver if the join point selection and the join point adaptation is completely specified and not influenced by user input.

The problem with user-driven weavers is that it is hard to study the implications of join point selections. The point in time when weaving is performed is the result of a stimulus provided by the application's user. As long as the user does not provide such a stimulus, weaving (and hence the join point adaptation) is not performed. Consequently, the semantics of the application can hardly be determined because the semantics of the application depends on the application's user. Consequently, the user's behavior represents (at least partially) a selection criterion.

System-driven weavers on the other hand operate with a predefined set of join point selection criteria and adaptation mechanisms. The application's behavior is consequently defined in terms of the underlying aspect specification.

5.6.4 Orthogonality of Design Dimensions

The design dimensions of dynamicity as well as the level of adaptation can be applied independent of each other in order to distinguish weavers (see Figure 5-49).

A system might provide for example static weaving based on code instrumentation (like AspectJ), while another one may provide dynamic weaving based on code interpretation (like for example Prose). As a consequence, applying the design dimensions permits to distinguish between four different kinds of systems.

However, while dimensions 1-3 seem to be reasonable implementations for weavers, the combination of static weaving and code interpretation seems to be rather exotic. On the one hand such a weaver would perform aspect-specific behavior at certain join points within the runtime system. On the other hand such a weaver would not permit to adapt new join points (or remove an adaptation) at runtime.

The reason why such an approach seems to be less reasonable is that evaluation rules in interpreters typically do not differ for expressions of the same kind (like for example method calls). Such a system would handle expressions of the same kind in different ways: For example, only some method calls would be adapted while others are not. For the developer, such a behavior would seem to be the rather unintuitive.

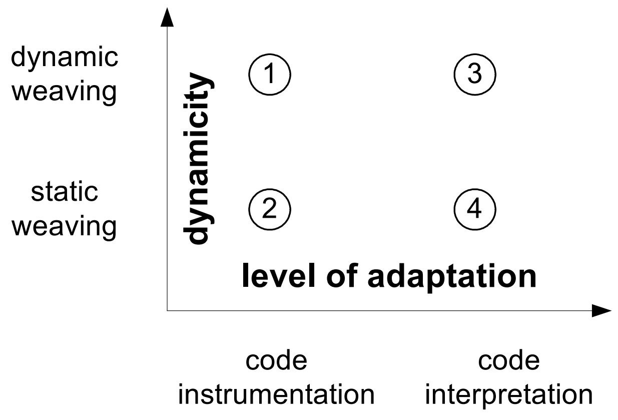


Figure 5-49. Orthogonal design dimensions for weavers.

5.7 Dependencies between Join Point Model, Selection, and Weaving

The relationships between the ingredients of an aspect-oriented system are conceptually described by Figure 5-50. Each join point has a number of properties which are addressed by join point selections. Join point selections indirectly select join points (illustrated by a dashed line) by addressing join point properties. Join point adaptations indirectly adapt join points (illustrated by a dashed line) because they refer to join point selections. The weaver implements the adaptation by transforming join points or by interpreting them in a different way.

According to the previous sections all elements of an aspect-oriented system are based on certain design dimensions that are in most cases orthogonal.

It has already been noticed in section 5.3.3 that an aspect-oriented system's concrete join point model can rely on different abstract join point models – like for example a system that provides static structural as well as dynamic behavioral join points. Each join point itself provides a number of properties – whereby different properties can rely on different design decisions with respect to the corresponding design dimensions. In the same way, an aspect-oriented system can provide different constructs for addressing join point properties – whereby different constructs possibly are based on different design decisions. Possibly a system provides different mechanisms in order to adapt join points whereby each mechanism is possibly based on different design decisions. And finally, a system may provide different weavers whereby each one is based on different design decisions.

However, until now it is not discussed whether there are any dependencies between design decisions of different ingredients of an aspect-oriented system. It needs to be discussed for example whether the underlying design decision of a certain join point property has an impact on the underlying join point model.

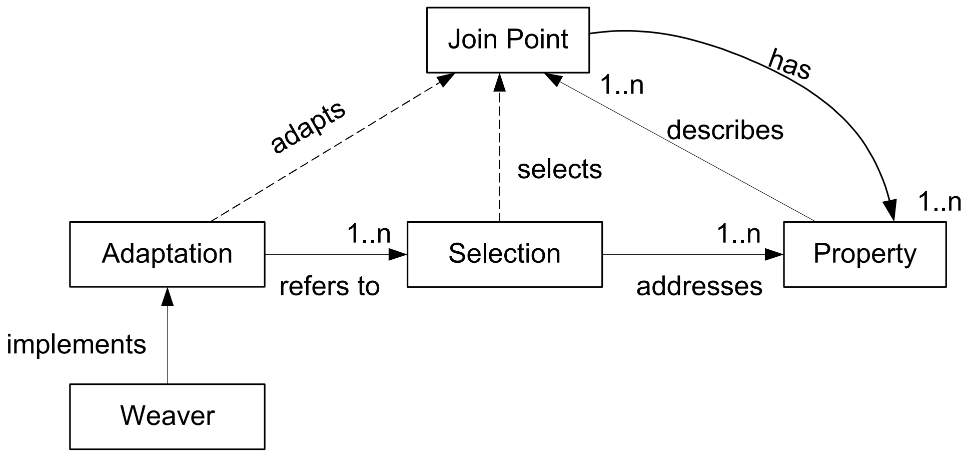


Figure 5-50. Relationships between Join Point, Join Point Property, Join Point Selection, Join Point Adaptation and Weaver.

A newly developed hypothetical aspect-oriented system can have the following characteristics. First, the developer decides to provide a dynamic, incomplete and behavioral join point model. For each concrete join point class the developer provides a number of properties; some of them are static, direct, atomic, and local, others are dynamic, direct, local, atomic and current meta-data properties. Each property can be addressed in a lexical, closed, stand-alone, monolithic, and unrestricted way. Each join point can be behaviorally, destructively, and non-variable adapted, and the weaving is performed by a static code transformer. A Java-based aspect-oriented system, which corresponds to this description, could provide method call join points identified by the method name (static, direct, local, and atomic property) and the type of the called object (dynamic, direct, local, atomic, and current meta-data property). Furthermore, the developer decides to implement the weaver as a preprocessor (i.e. a static code transformer). Obviously, such a system seems to be reasonable.

The exemplary system could be changed in some ways. First, the weaver could be implemented as an interpreter instead of a code transformer. Instead of changing the behavior of the application according to some aspect specification the weaver is just an interpreter, which determines at every join point whether a corresponding aspect refers to it. This design decision addressing the weaver does not touch the design decisions addressing the other ingredients. Second, the system could be changed in a way that new properties are to be provided – for example by providing the static types of each method call's parameters. Adding new parameters simply changes the data provided by each join point – in this specific case the other design dimensions are not touched.

However, in case the system provides no dynamic properties at all, there are no runtime specific characteristics that can be used in order to distinguish join points – all method calls can only be selected because of the static types of their parameters and the methods' names. Consequently, all join points are only distinguishable by static properties and the resulting join point model is a static one. Thus, a given concrete join point class is based on a dynamic join point model if there is at least one dynamic property available for the corresponding class – a concrete join point model directly depends on the provided properties. In other words, the join point model is partly restricted by the provided join point properties.

5.8 A Critial Discussion of the Term Design Dimension

The main intention of the design dimensions is to describe different, orthogonal views in the possible design decisions of aspect-oriented systems. From this perspective the term design *dimension* seems to be reasonable to describe the approach in this chapter.

However, the term also might be understood in the wrong way. The term dimension is also used for example in linear algebra in mathematics and seems to describe a similar thing there.

Nevertheless, the term dimension in algebra and the term dimension being used here differ noteworthy. First, not all constructs that are called dimensions in this chapter are orthogonal to other design dimensions (an example is the locality of join point properties as described in section 5.4.3.4 and the application's progress as described in 5.4.3.7). Such non-orthogonal constructs are from a mathematical point of view a contradiction with respect to the term dimension. Second, the described dimensions have a finite number of possible values: For example, this thesis identifies two different implementations of the dynamicity dimension (static and dynamic, see section 5.3.1). This is also a contraction to the term dimension as used in algebra, where each dimension has an infinite set of values.

Nevertheless, this thesis still considers the term design dimension to be appropriate in order to communicate the underlying idea because of two reasons.

First, in [Wegn87] the same term is being used to describe different kinds of object-oriented systems. Consequently, the use of the term within this thesis can be considered an adaptation of the term from the object-oriented world.

Second, the term design dimension seems to be a valid *metaphor* for describing the approach. Instead of relying on exact definitions of the term, it is rather considered to be a metaphor that suggests the right meaning and which still requires an appropriate interpretation by the reader. Although the use of metaphors in software technologies seems to be in contradiction with an often cited goal of accuracy and determinism, this thesis agrees with approaches as described for example in [Beck02] that consider the use of metaphors in software construction as a valuable contribution in order to communicate concepts and ideas.

5.9 Chapter Summary and Conclusion

This chapter proposed the main contribution of this thesis – the design dimensions of aspect-oriented system. Thereto it identified in section 5.2 the core ingredients of aspect-oriented system – the join point model, the join point selection, the join point adaptation and the weaving – and proposed a number of design dimensions for each ingredient in the following sections. The main motivation for such an analysis of design decisions underlying different aspect-oriented approaches is the observation that there are a number of differences between systems which are commonly accepted to be called aspect-oriented. Such differences have a large impact on the way of how aspect-oriented system can be understood and how they can be applied. On the one hand the underlying terminology of different systems differs, on the other hand the underlying means to select and adapt join points differs. For the developer of new systems this situation is problematic because the alternatives in providing an aspect-oriented system for a given base language are not directly obvious and need to be derived by analyzing existing systems.

In detail, this chapter introduced the following design dimensions for join point models, join point properties, property addressing, join point adaptations, and weavers.

- **Join Point Models**:

  Dynamicity (section 5.3.1) – that distinguishes between static and dynamic join points, **Level of Abstraction** (section 5.3.2) – that distinguishes between structural and behavioral join points, and **Completeness** (section 5.3.5)– that distinguishes between incomplete and complete join point models.

- **Join Point Properties**:

  Dynamicity (section 5.4.3.1) – that distinguishes between static and dynamic properties, **Directness of Property Correspondence** (section 5.4.3.2) – that distinguishes between direct and abstract property correspondence, **Structuredness** (section 5.4.3.3) – that distinguishes between atomic and structured properties, **Locality** (section 5.4.3.4) – that distinguishes between local and non-local properties, **Identity** (section 5.4.3.5) – that distinguishes between identity and non-identity properties, **Data Representation** (section 5.4.3.6) – that distinguishes between data and metadata properties, and **Application's Progress** (section 5.4.3.7) – that distinguishes between current, past and future data properties.

- **Join Point Property Addressing**:

  Directness (section 5.4.4.1) – that distinguishes between lexical and indirect value addressing, **Openness** (section 5.4.4.2) – that distinguishes between closed and open value addressing , **Level of Value Sharing** (section 5.4.4.3) – that distinguishes between stand-alone and shared property addressing, **Adjustability** (section 5.4.4.4) – that distinguishes between incremental and monolithic join point selections, and **Scope** (section 5.4.4.5) – that distinguishes between unrestricted and restricted join point selections.

- **Join Point Adaptation**:

  Level of Adaptation (section 5.5.1) – that distinguishes between structural and behavioral adaptation, **Constructiveness** (section 5.5.2) – that distinguishes between constructive and destructive adaptations, **Variability** (section 5.5.3) – that distinguishes between non-variable and parametric adaptations, and **Level of Join Point Abstraction** (section 5.5.4) – that distinguishes between fix and variable join point abstraction.

- **Weavers**:

Dynamicity (section 5.6.1) – that distinguishes between static and dynamic weaving, **Level of Adaptation** (section 5.6.2) – that distinguishes between code instrumentation and code interpretation, and **Weaving Stimulus** (section 5.6.3) – that distinguishes between user-driven weaver and system-driven weaver.

The design dimensions were on the one hand extracted based on a study of parallels and differences in the current aspect-oriented literature and on the other hand based on a comparison of parallels and differences between different aspect-oriented systems. They represent a common conceptual framework for aspect-oriented systems. On the one hand they provide a conceptual framework to understand such systems in terms of join point models, join point selections, join point adaptations, and weavers. On the other hand they provide distinguishing characteristics on a lower technical level which can be used in order to compare different systems or to build up new systems. For example, a system can be analyzed with respect to whether it provides the ability to address properties in an indirect way. Also, such a system can be analyzed in respect to whether code instrumentation is the underlying technique for achieving the connection between the aspect-oriented system and the base system.

Based on the identified core ingredients of aspect-oriented systems as (section 5.2), it is possible to express the characteristics of aspect-oriented systems by means of an equation similar to the way the term object-oriented was expressed in [Wegn87][93]:

Aspect-Oriented = Base System + Join Points
+ Join Point Selection + Join Point Adaptation

Although this (non-formal) equation rather communicates the core ingredients of aspect-oriented systems on an abstract level, it describes the characteristics of aspect-orientation on a finer-grained level than the nowadays used terms quantification and obliviousness. If a finer-grained study of a certain system is needed, the systems ingredients can be mapped to the proposed design dimensions.

The design dimensions are on different levels of abstractions and address different problem domains. For example, the design dimensions of join point models are very high-level abstractions for the (concrete) join point model underlying aspect-oriented systems. This abstraction provides a very generic overview of aspect-oriented systems and abstracts from a number of essential elements – the way of what information is available in order to select a join point and adapt join points. The design dimensions of join point properties are quite technical and represent rather a low-level abstraction because the analysis of what design dimensions are applied needs to be studied for each property independently of each other.

The design dimensions represent a common basis that can be used in order to understand and compare aspect-oriented systems. Furthermore, the design dimensions can be used to assess aspect-oriented systems with respect to their appropriateness to modularize a given crosscutting concern.

[93] In [Wegn87] the term *object-oriented* is characterized by the equation *object-oriented = objects + classes + inheritance.*

6

IMPLEMENTATIONS OF DESIGN DIMENSIONS

6.1 Introduction

The previous chapter described design dimensions for the different ingredients of aspect-oriented systems. In order to show that the design dimensions represent a reasonable framework for aspect-oriented systems, this chapter applies them to a number of systems. Furthermore, this chapter contributes to the discussion of the aspect-orientedness of **roles** by explaining the role concept in terms of the design dimensions.

Section 6.2 first discusses briefly different alternatives of aspect-oriented systems in order to implement certain design dimensions. Sections 6.3 – AspectJ, 6.4 – Hyper/J, 6.5 – AspectS, 6.6 – Sally, and 6.7 – Morphing Aspects map the aspect-oriented systems introduced in Chapters 2 to 4 to the design dimensions introduced in Chapter 5. Such mapping requires a detailed study of all features of the corresponding system. Especially all properties provided by the corresponding system (which are typically a lot) need to be analyzed with respect to how they implement the corresponding design dimension.

Section 6.8 maps the role concept (as introduced in [KrØs96, Kris96]) to the design dimensions. Section 6.9 summarizes and concludes this chapter.

6.2 Variabilities in Implementations

The design dimensions leave open a large number of different ways how they can be implemented by aspect-oriented systems. First, developers of an aspect-oriented systems can choose based on their join point model which kind of encoding they want to provide, i.e. which properties should be available as distinguishing characteristics for each join point. Each provided property might be based on different implementations of different design dimensions. Next, for the join points a number of language constructs that address the corresponding properties are required. Then, constructs need to be found that define how to adapt the join points. Finally, a technical solution for the weaver is required.

6.2.1 Join Point Model

The abstract as well as the concrete join point model depend on the one hand on the design decisions of the system's developer. On the other hand this also depends on the base language underlying the aspect-oriented system.

When a system is developed, the developer has to determine what structural elements as well as what behavioral elements should represent join points: The language constructs provided by a certain programming language implicitly define the possible kinds of join points. In principle, the easiest decision is simply to provide a complete join point model (see section 5.3.5). However, the consequence of such a decision is that the aspect-oriented system also needs to provide corresponding properties and addressing mechanisms for all join points. Furthermore, developers need to provide corresponding adaptation constructs for each kind of join point. Because of this effort, it is currently rather not the case to provide complete join point models.

In addition to the structural and behavioral elements the developer also has to determine whether the system should provide static or dynamic join points. In general, the effort of providing static join points seems (from the current perspective) easier because adaptation of static join points (based on a code transforming weaver) does not require to generate corresponding join point checks (see Chapter 4 as well as section 5.3.1 for a detailed discussion on join point checks and shadow join points).

In general, the set of potentially provided kinds of join points differs from programming language to programming language due to the different abstraction being provided by each one. For example, templates are quite typical constructs in C++ that do not have direct correspondences in languages like Smalltalk or Java. Consequently, template definitions are potential static structural join points in C++ whereby no corresponding kind of join point exists in Java or Smalltalk. Java provides declared exceptions within method headers that also potentially represent join points (which are not available in C++ or Smalltalk). In Smalltalk blocks are potential join points (which are not available in Java or C++).

Nevertheless, it seems from the current perspective that at least method definitions in object-oriented systems should represent (structural) join points (since all known aspect-oriented systems provide such join points). Consequently, the decision whether methods are to be provided as join points does not seem to be design alternative for aspect-oriented systems.

6.2.2 Join Point Properties and Join Point Addressing

With respect to join point properties as well as join point addressing, there is a large variety of possibilities how they can be implemented.

With respect to join point properties developers face two situations (see also section 5.4.3). First, the set of possible available join point properties depend on the underlying programming language. Second, even for the same programming language different developers can provide for the same kind of join point different properties: Properties which are common with respect to their design dimensions do not necessarily represent the same information for each join point. For example, a property in a Java-based aspect-oriented system representing the number of parameters of a (static) method declaration join point is a **static**, **local**, **atomic** property with an **abstract**

correspondence. In comparison to this a property that describes the number of declared exceptions for a method definition join point is also a **static**, **local**, **atomic** property with an **abstract correspondence**. Although both properties have the same characteristics and also refer to the same join point (and both are provided in aspect-oriented systems based on the same programming language), both properties obviously describe two completely different things. Hence, the selection of the corresponding join point based on the one property will be useful for developers in other circumstances than the selection of the join point based on the other property.

In general, it is up to developer to find a number of properties that seem to be useful for the developer. Furthermore, it seems desirable to provide not too many properties: The more properties are available the more complex becomes the resulting language. Furthermore, it seems clear that the provision of dynamic properties coming from the applications past require a larger overhead for the implementation that simply providing current data properties because the system needs to maintain a number of different runtime information in order to provide corresponding join point properties.

With respect to the language constructs to address join point properties the developer has even more freedom. While **lexical** and **closed** value specification as well as **stand-alone** value specification hardly leaves any freedom for the developer, there are many different ways how **indirect** value specifications, **open** value specifications and, **shared** value specifications can be implemented.

Indirect value specification just means that there are language constructs for addressing a value of a property without specifying it lexically. However, it is up to the developer to decide what values potentially can be indirectly specified and how. For example, a type might be selected because of its position in the inheritance hierarchy (like implemented in AspectJ), but maybe also because of its structure (i.e. for example selecting a type because of the types it refers to). Hence, the developer may decide to provide an indirect value specification that reasons on part-of relationships between different types.

With respect to the **open** of values specifications there are many different possibilities. Developers may want to provide open value specifications only for certain kinds of join points or for certain properties. For open value specifications developers may want to provide regular expressions or even context-free grammars, etc.

Shared value specifications give developers the freedom to choose among a large number of possible implementations. One possibility is to implement join operations like known from the relational databases like natural joins or outer joins ([Codd70]). Another alternative is to provide a Turing complete language that shares variables among selection expressions.

However, typically the design decisions for aspect-oriented systems with respect to the encoding and the value addressing are not independent of each other in a way that the developer first decides what kind of encoding and then what kind of selection he wants to provide. There are situations where it is possible to disclaim a certain feature on the encoding level and instead providing an additional feature on the addressing level (cf. for example the discussion in section 5.4.5). From the user of the aspect-oriented system's perspective the result could be the same. For example, if the developer of the system wants to give users the ability to select join points in a more sophisticated way. Selecting a static method definition join point because of the number of parameters in the method's signature can be achieved by providing additional properties, or additional addressing constructs, or both.

Incremental adjustability of join point selections also permits a large variety of different implementations. Probably the best-known approach is to provide an inheritance-like mechanism in order to share join point selections among different modules (such an approach is provided for example by AspectJ or Sally). Another alternative is to provide variables within join point selections that can be set by developers in analogy to the template instantiation in languages like C++.

6.2.3 Join Point Adaptation

Similar to the previous section, the design dimensions for join point adaptations give a general framework for the design of adaptations, but aspect-oriented system developers still have much freedom to provide a corresponding implementation.

With respect to **structural** or **behavioral** adaptation the design dimensions do not prescribe what kind of (structural of behavioral) adaptation should be applied. One language construct for performing structural adaptations could for example add additional superclasses or interfaces to a class definition (or to an object if the corresponding join points are dynamic and structural). Another construct may add exception declarations or additional parameters to a method definition. What kinds of join point adaptations are possible also depends on the underlying programming language. For example, in languages like Java that provide explicitly declared exceptions within method headers, the addition of such exceptions to static method definition join points seem to be a reasonable join point adaptation. Of course, such an adaptation is not reasonable in programming languages like Smalltalk or C++ that do not provide declared exceptions.

Also, the design dimensions of join point adaptation refer only to adaptation constructs for a given kind of join point. For example, developers might decide to provide a variable join point abstraction only for behavioral adaptations of behavioral join points, while having a fix adaptation for structural join points[94].

For **parametric join point adaptations** developers possibly decide to provide only certain kinds of parameters within the introduction. One possibility is (in correspondence to class templates in C++) only to provide types as variables within join point adaptations. Another possibility is (according to Sally) to provide any identifier or parameter list as a parameter.

In general, although the abstract join point model might be already determined there is still a large variety of how each kind of join point can be adapted.

6.2.4 Weaving

Weaving is probably the most complicated task in aspect-oriented systems, because the weaver provides the connection between base system and the aspect-oriented system.

[94] This corresponds for example to the implementation of join point adaptation in AspectJ, which will be discussed in section 6.3.3.

If the weaver performs **code instrumentation** and the underlying join point model is a dynamic one, the weaver must (when weaving is performed) determine and adapt all locations where potentially or for sure aspect-specific code needs to be executed (see further Chapter 4 as well as [HiHu04]).

Obviously, building a weaver based on code instrumentation in combination with a dynamic join point model is quite a complex task. The effort of building a weaver making use of a dynamic join point model and which is based on **code interpretation** can be substantially less complex than the one build upon code instrumentation. For example, [BHMO04] argue that the creation of a weaver by extending an open source java runtime system just took about 2000 lines of code (including an additional API).

If the join point model is a static one and all join point properties are local ones the resulting weaver is much more trivial since the join points can be directly determined by reasoning on the application's syntax and do not consider the base program's semantics. In such a situation, weaving can be easily achieved by performing pure syntactic transformations on the source code where the weaver has to guarantee that the resulting woven application is valid.

There are technical reasons for or against a certain kind of weaver. Possible reasons against the implementation of a new interpreter could be (depending on the underlying language) that some characteristics of the original base system (like for example platform independence, performance, etc.) do not hold any longer if a new interpreter is provided. In situations where such characteristics are needed it is rather desirable to have an aspect-oriented system build on code instrumentation.

Systems that provide **dynamic weaving** need at least some mechanism to enforce weaving. I.e. some API is necessary to permit developers to start the weaving process. For example, in AspectS weaving is achieved by sending messages. On the other hand, dynamic weaving could be also achieved by a corresponding access to a running system where the system itself does not permit to enforce the weaving process, but which requires developers to enforce weaving by using some external tools[95].

6.3 AspectJ

A number of elements of the design dimensions were motivated and explained in terms of AspectJ. Especially a number of different kinds of join point properties and join point addressing are provided by AspectJ due to its rich join point selection language[96]. In the following the aspect-oriented language constructs of AspectJ are explained in terms of the design dimensions as introduced in Chapter 5: Join point model, selection, adaptation, and weaving are described in separate subsections.

[95] Such external tools would have some similarities to deployment tools known from for example Enterprise JavaBeans where administrators are permitted to deploy additional Enterprise Beans.

[96] The phrase **rich pointcut language** was used in [RaSu03] in order to explain that AspectJ's pointcut language differs noteworthy from join point selections like e.g. in Hyper/J.

6.3.1 Abstract Join Point Model

AspectJ provides two different kinds of join point adaptations namely introductions (or inter-type declarations) and advice which both permit to adapt different kinds of join points.

On the one hand AspectJ permits to select and adapt types (classes as well as interfaces) using introductions. Classes and interfaces represent **static** and **structural join points**. Java also provides as structural elements also anonymous classes, i.e. classes that do not have a name. They cannot be selected and adapted via introductions. Hence, AspectJ is with respect to static and structural join points **incomplete**.

On the other hand, AspectJ permits to select class and object initialization, method and constructor executions, method invocations, field accesses and assignments, and caught exceptions. These entire join points can be selected because of dynamic properties like the actual type of the currently involved object. Consequently, all of these join points are **dynamic**. Method calls, field assignments and accesses, as well as exception handlers are all encapsulated in structural elements and represent the application's behavior – hence they are **dynamic** and **behavioral join points**. Method execution join points directly refer to methods but can be furthermore restricted by additional runtime conditions. Hence, they represent **structural** and **dynamic join points**.

A number of expressions like typecasts or operators cannot be selected and adapted. Consequently, AspectJ is **incomplete** with respect to dynamic and behavioral join points. With respect to structural and dynamic join points AspectJ is also incomplete – AspectJ does no permit for example to express object-relationships within its pointcut language.

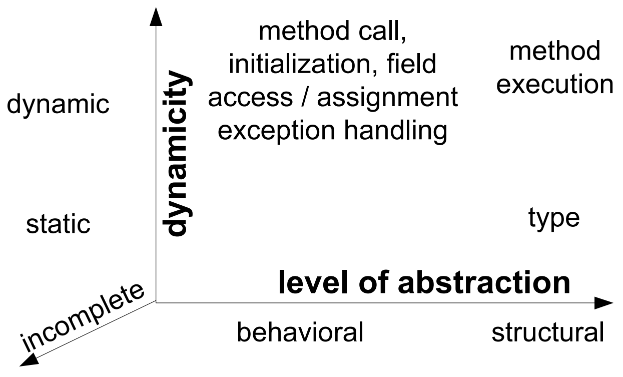


Figure 6-1. Abstract join point model.

Figure 6-1 illustrates the join point model underlying AspectJ in terms of the design dimensions. The entries within the design dimensions correspond to the different kinds of join points that AspectJ provides.

6.3.2 Join Point Properties and Property Addressing

Due to the different kinds of join points (static and dynamic join points adaptable via introductions and advice) AspectJ provides a different number of join point properties and different ways of how such properties can be addressed.

This section is structured as follows. First, the properties of static join points are described in terms of the design dimensions. Then the means to address such properties are explained. Afterwards, the properties of dynamic join points are described in terms of the design dimensions. Finally, the means to address such properties are explained in terms of the design dimensions.

6.3.2.1 Properties of static join points

Type join points that are used via introductions are encoded with a corresponding typename property – which is the only property of type declaration join points. This property represents the type via its name. The type name property is a **static property** since it can be directly derived from the base system's sources. Since the type name directly appears in the application the property's value has a **direct correspondence**. The structuredness of a typename property is not that obvious. In Java, the name of a type is structured – since a type name consists of a package name and a type identifier (in case of inner types the type name furthermore consists of the outer types' names). However, this structure is simply a chain of characters – the structure corresponds to the interpretation of such characters. Furthermore, it is not possible to deduce from a type's name whether or not it is an inner-type. On the other hand, all types that correspond to the same package have the same prefix.

Hence, the type name property is a **structured property**. Since type names can be directly derived from the position in the source code where they are declared, the type name property is **local**[97]. Furthermore, since full qualified type names are unique in Java, the type name property also represents an **identity** property.

6.3.2.2 Addressing properties of static join points

The type name property is being used within an introduction specification. However, AspectJ distinguishes between member and parent introductions. Parent introductions are specified by declaring any arbitrary type pattern. Member introductions are specified by declaring one single type name. Consequently, both kinds of introductions have a separate way of selecting join points.

Member introductions require to address the target type via its name. It is not possible to abstract from the type name using for example the wildcard `*`. Hence, member introductions refer to a type property via a **lexical**, **closed,** and **stand-alone** addressing. The selection is **unrestricted** since within an introduction any arbitrary type can be chosen (i.e. it is not restricted to a certain set of types).

Parent introductions refer to a type property via a **lexical** as well as an **indirect** addressing – lexical since the type name can be specified, indirect since the operators `+`, `&&`, `||`, `!`, and `*` can be used in order to address the property's value with respect to certain type relationships. The join point selection is due to the possible use of the previous mentioned operators **open**. For example, a type pattern `(A+ && B+)` selects all types which are subtypes of `A` as well as subtypes of `B`. This is an open selection since in future versions of the system new types might appear that extend `A` as well as `B`. For

[97] Not that in the presence of generic types the property representing an instantiated generic type does not fulfill such properties. However, such a feature does not (yet) exist in the AspectJ versions underlying this thesis.

the same reason as argued in the previous paragraph the selection is **stand-alone** as well as **unrestricted**.

With respect to its adjustability introductions are slightly problematic. In principle, it is not possible to refine a join point selection – a once specified selection criterion cannot be overridden or redefined. However, it is possible to select a type for an introduction that is later introduced to a different type – this usage of introduction represents the foundation of the container introduction idiom [HSU03]. The underlying idea is that the selection is specified in terms of an aspect-specific type, which is later on introduced to a number of different types using a parent introduction. Consequently, type patterns used within introductions are considered to be a restricted kind of **incremental** adjustability[98].

6.3.2.3 Properties of dynamic join points

The different kinds of dynamic join points have some properties in common and differ with respect to other properties.

In general, AspectJ provides four *kinds* of properties. The first three are:

- Properties describing types or type lists (`within`, `this`, `target`, `args`, as well as return types, and formal parameter types in `call`, `execution`, and `withincode`),
- Properties describing method names (as part of `call`, `execution`, and `withincode`),
- Properties describing objects (`this`, `target` and `args`)[99].

All of these properties are addressed in a similar way but they differ with respect to the design dimensions these properties are based on. The fourth kinds of property significantly differs from the other three, because no object or message but the control flow is encoded:

- The control flow property (`cflow`) describes the call stack join point occur in.

All dynamic join points have a `within` property which represents the class or the aspect containing the corresponding join point shadow. This property is quite similar to the type name property mentioned in the section 6.3.2.1 – it represents a **static**, **structured property** with a **direct correspondence**. However, the property differs with respect to the identity as well as with respect to its locality. The dynamic join points which are encoded by this property are for example method calls or field accesses – they are not uniquely represented by the corresponding type name because a class possibly contains more than one field access. Consequently, the `within` property itself is **no identity property**. For structural dynamic join points like method executions the `within` property becomes **part of the join point identity**. With respect to the locality the property represent a local property: A class definition is a module that lexically contains its join point shadows.

---

[98] It is restricted, because the developer has to preplan such an incremental adjustability by performing an introduction on an interface.

[99] Note that `this`, `target`, as well as `args` describe objects as well as actual types (cf. section 2.2).

The property `this` represents the object where a certain join point occurs in. Hence, the property is a **dynamic** one. The object is directly available at the corresponding join point. Hence, the property is **local** and has a **direct correspondence**. The property itself is **no identity property** since the actual type abstracts over the join point shadow. Furthermore, this property is a **current property** since no data from the past or the future is being represented. Whether the property is a data or a meta-data cannot clearly answered: The problem is that the property is being addressed by a type name. From this perspective it is a metadata property. On the other hand, the corresponding object can be passed to an advice – from this perspective the property is a data property. Since the data available at each join point can be accessed via `thisJoinPoint` within an advice, and requesting the `this` property returns the current object, the property is a data property. Nevertheless, the data cannot be addressed using selection constructs. Consequently, the conceptual data provided for the purpose of being selected is a **meta-data property** (the runtime type), while the data-property (the current object) serves different purposes (to be used within the advice)[100].

The property `target` represents the target object of a method call. The characteristics of this property are equal to the characteristics of `this`: It is a **dynamic, local, current, structured, meta-data property** representing **no identity** and has a **direct correspondence**.

The `args` property represents a list of actual parameters at a method call (or method execution) join point. The property corresponds with respect to the design dimensions to `this` and `target` – the property is **dynamic, local, current, meta-data** representing **no identity** and has a direct correspondence.

The signatures being used within `call`, `execution`, and `withincode` differ with respect to how they are derived from the base application. `Call` and `execution` describe the signatures of the methods being invoked or being executed, while `withincode` describes the signature of the method containing the corresponding join point. The signature itself is **structured** and consists of the return type, the declaring type, the method name and the parameter types declared in the corresponding method – consequently according to the discussion in section 5.4.3.3 all ingredients can be considered as properties on their own.

For execution join points the current return type property, the declaring type property, the method name property and the parameter types property are all **static properties**. All of them have a **direct correspondence** in the source code since method names and types are explicitly declared in Java. All properties are furthermore **local**. The method name is **unstructured**, the return type, declaring type, as well the parameter types are **structured**. Nevertheless, the structures differ – while the return type and declaring type is structured according to the package/type/inner type-relationship as describe above for the `within` property, the parameter types represent a list of types. Each type occurring in this list is itself structured in the same way as the return type. For execution join points the whole signature itself represents a shadow

[100] However, although only the type name can be addressed using the pointcut language the development effort is the same as providing the data property: The object needs to be stored within the `thisJoinPoint` object reflecting the current join point.

identity – since due to the semantics of Java it is prohibited to declare two methods within the same type with the same signature.

The signatures for call join points are similar in many points but differ in others significantly from the signatures of execution join points. The parallels are, that return types, declaring type, method name and parameter types are **static properties** with a **direct correspondence** and all except the method name are **structured**. However, the properties differ in their locality. The method name is obviously **local** (since a method name can be directly found in the sources at the corresponding locus). But in order to determine the types of the called method a number of typing relationships need to be known. The static type of the called object needs to be computed, the static types of the parameters need to be computed and the corresponding method in the target type has to be chosen according to the dispatch mechanism in Java. Although typing information can be derived from the syntax of the whole application it cannot be derived from the local syntax information available at the join point[101]. Consequently, the join point property is **non-local**.

The signature described by the `withincode` property consists of the same value as the `within` property as described above as well as the signature of the method where the corresponding join point occurs in. The signatures described in `withincode` and `execution` correspond to each other (since both unambiguously describe the corresponding join point shadow) – consequently the characteristics of the properties are the same. The only exception is the identity property. While `withincode` describes the shadow **identity** for the execution join points, it does no describe such an identity (**non-identity**) for method call join points since a number of method calls can occur within the same method.

The `cflow` property conceptually represents the "join point stack" where a certain join point occurs in. On the implementation level the `cflow` construct is (probably due to performance reasons) only a small extract from the call stack the developer can refer to. All elements on the stack conceptually consist of join points – the elements can be accessed with a subset of the selection language (for example the `if`-construct is prohibited). From this point of view, the `cflow` property can be understood as a list of join points expressing the current control flow. Consequently, the property is **dynamic** and **structured**. The elements in the list themselves are furthermore structured (since all join points have a number of structured properties). Since call-stack information is not directly available, the property is **abstract**. Since in Java the developer does not have any access to objects on the call stack, the `cflow` property reveals information that is not available in the ordinary execution context. Consequently, the `cflow` property represents a **non-local property** (see also the discussion in section 5.4.3.4). The information available via the `cflow` property is not sufficient to determine the current shadow's identity. Consequently, `cflow` is a **non-identity property**. Since the `cflow` contains elements that are themselves meta-data properties, the `cflow` itself is a **meta-data** property. According to the discussion in section 5.4.3.7 the cflow property is considered as a **past property**.

101 Note that the distinction between source code and byte code becomes important here: The Java byte code contains already static typing information which are constructed at compile-time.

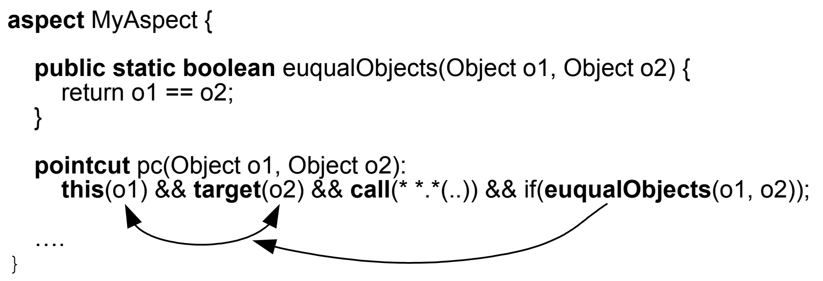
6.3.2.4 Addressing dynamic join point properties

In AspectJ the different kinds of properties are addressed in different ways – depending on whether the property is a type, object or a list of both, a method name, or the `cflow` construct. Furthermore, AspectJ provides the `if`-pointcut designator which is discussed afterwards.

Properties representing types can be addressed corresponding to the type selection of parent introductions (see section 6.3.2.2) – they can be addressed in a **lexical**, **indirect**, **closed** and **open, stand alone**, and **unrestricted** way. The properties that represent objects (`this`, `target`) are addressed in a similar way – **open**, **stand-alone**, and **unrestricted**. However, with respect to the directness of the addressing the implementation it is not obvious how the underlying dimension is implemented: Such properties are addressed either due to lexical characteristics of the object's dynamic types or due to an indirect addressing that refers to type relationships. Since the corresponding pointcuts permit to address a type by specifying the type's name in addition to combining the lexical specification with the operators `*`, `&`, `||`, and `+`, the corresponding addressing is **lexical** as well as **indirect**.

The structured property referring to type and object lists (the parameter type list in signatures as well as the actual parameter list in `args`) can be addressed according to the type properties – **lexical** and **indirect**, **closed** and **open, stand alone**, and **unrestricted**.

The `cflow` property is addressed by addressing properties of join points that occur within the control flow. Consequently, all parts of the `cflow` property are addressed corresponding to the previous discussion.



```
aspect MyAspect {

  public static boolean euqualObjects(Object o1, Object o2) {
    return o1 == o2;
  }

  pointcut pc(Object o1, Object o2):
    this(o1) && target(o2) && call(* *.*(..)) && if(euqualObjects(o1, o2));

  ....
}
```

Figure 6-2. Shared value specification in AspectJ.

A special kind of property addressing is the `if`-pointcut. Since all bounded objects within a pointcut specification can be passed to a method via the `if`-pointcut (but not the join point itself) the whole computational power of Java (including its reflective capabilities) can be used in order to specify selection criteria on these objects. Correspondingly, all object-properties encoded by `this`, `target` and `args` can be addressed additionally in a **shared** way – by specifying corresponding comparisons within the corresponding method. Such a shared value specification is illustrated in Figure 6-2: The join points are only selected in case objects `o1` and `o2` are identical. However, it should be noted that this join point selection is not achieved using the pointcut language. Instead, the base language is being used in order to determine the selection. Consequently, this value sharing is not part of AspectJ's pointcut language.

6.3.3 Join Point Adaptation

The two ways of adapting join points in AspectJ are introductions (or intertype declarations) and advices that need to be discussed separately.

6.3.3.1 Introductions

Conceptually, introductions are **structural** join point adaptations because they add fields or interfaces to existing types. However, introductions can also be behavior adaptations because of some (accidental) introduced members. From this thesis' point of view the behavioral adaptation performed by introductions is not considered as the conceptual focus of introductions but rather an accidental characteristic. Hence, introductions are considered to be pure structural adaptations.

Introductions are **constructive** in the sense that the type is extended by additional constructs. With respect to the variability introductions are **fix** – the code being executed cannot contain any variables. However, it needs to be pointed out that (of course) the executed code does not need to behave equal due to late binding in Java. Nevertheless, it is not possible to have any variables like types or method calls within introductions.

The join point's abstraction is also **fix**: The context of the introduced elements is determined by the selected join point. It is not possible to extend or reduce the context using the selection language used for introductions (type patterns). For member introductions the special variable `this` refers to the selected type, i.e. the context always corresponds to the target type.

6.3.3.2 Advice

The advice construct is a **behavioral** adaptation. AspectJ implements the weaver as a code transformer that generates new methods and relationships in order to weave in advice. Consequently, the structure is transformed. However, from the developer's perspective, this structural change is not relevant and the generated items are not directly accessible[102].

AspectJ provides three different kinds of advice – before, after, and around advice. The first two kinds are conceptually pure constructive adaptations: The original join point becomes executed. Of course, before and after advice can change the semantics of the executed join point – before advice by changing the join point's parameters, after advice by changing the returned value. But as pointed out in section 5.5.2, this does not contradict the intention of constructive adaptations. Furthermore, both kinds of advice possibly throw runtime exceptions and may change the control flow in that way. In the first case the original join point is not executed, in the latter case the control flow traces back the call stack to the first try-catch construct. Although in such situation the join point significantly changes, before and after advice are considered as being **constructive** adaptations. The around advice permits the developer to decide based on some runtime data whether the original join point is to be executed using `proceed`. Consequently, around advices without any occurrence of proceed are **destructive** adaptations, around

[102] Only in case the underlying application uses the introspective capabilities of Java, the resulting woven code is visible for the developer. Nevertheless, the intention of advice is not to rely on such structural changes.

advices containing a proceed are **conditional constructive** join point adaptations. Likewise introductions, an advice is **fix** with respect to its level variability. With respect to the join point abstraction, advice provide **variable join point abstractions** because the parameters passed to the advice depend on the underlying join point selection: Additional parameters coming from the control flow can be passed to the advice and which are not directly available at the corresponding join point. Furthermore, the context can be restricted by for example passing only few elements from the original join point context to the advice (for example only some actual parameters of method executions instead of all).

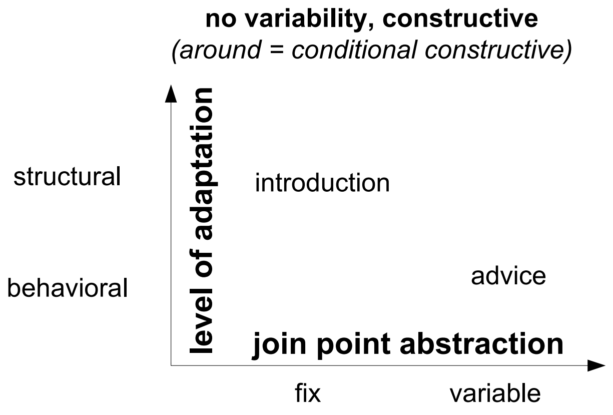


Figure 6-3. Design dimensions of join point adaptation in AspectJ.

Figure 6-3 illustrates the design decisions of AspectJ's join point adaptations according to the underlying design dimensions (whereby the design dimensions of variability and constructiveness are not illustrated due to the discussion above).

6.3.4 Weaving

Weaving in AspectJ is realized in previous versions at compile time on the source code level – the AspectJ compiler computes from the join point selection those shadows whose dynamic join points are potentially adapted. The compiler adds some runtime checks to such join points (cf. [HiHu04]) and inserts appropriate invocations according the join point adaptation. The developer is not able to change the woven code at runtime. In newer versions of AspectJ, the join point selection and adaptation is performed in a compilation-like process that is also executed before runtime. From the design dimension's perspective the distinction between byte code adaptation and source code adaptation is not important: The AspectJ weaver is a **static** weaver based on **code instrumentation**.

6.4 Hyper/J

This section explains the aspect-oriented constructs of Hyper/J (see section 2.3) in terms of the design dimensions as introduced in Chapter 5. Join point model, selection, adaptation and weaving are described in separate subsections.

6.4.1 Abstract Join Point Model

Hyper/J provides a number of different composition rules for composing classes like merge and override for composing methods. Furthermore, bracket relationships are provided in order to add additional code to method calls.

The means of composing classes in Hyper/J permit to select a number of classes that composition rules are applied to. Since classes correspond to the static structure, they are **static** and **structural join points**. In the same way, methods (or operations in the Hyper/J terminology) are selected and adapted with additional functional code. Since methods represent elements from the static structure, they are also **static** and **structural join points**.

Method calls on the other hand are selected depending on the class they are located in, on the static target type of the call, and on the called method's name. No dynamic elements are being used in order to select such method calls. Hence, method calls in Hyper/J are also **static join points**. Since such method call join points are encapsulated within structural elements (the method definitions), they represent **static** and **behavioral join points**.

In Hyper/J method calls are the only behavioral join points - casts, supercalls, operations, etc. do not represent join points. Furthermore, anonymous classes or relationships between classes, etc. are no join points either. Consequently, the join point model of Hyper/J is incomplete with respect to behavioral as well a structural join points.

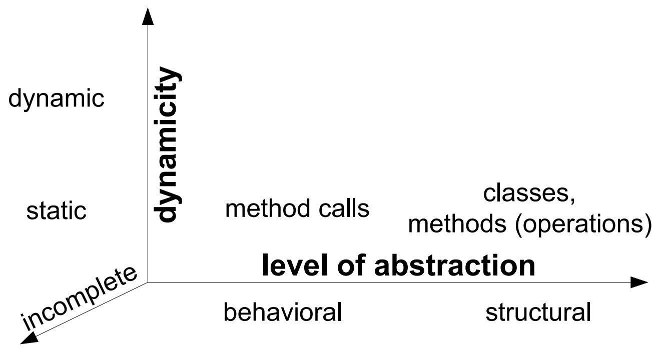


Figure 6-4. Hyper/J's abstract join point model.

Figure 6-4 illustrates the abstract join point model of Hyper/J according to the underlying dimensions of join point models. Since all join points are static elements, Hyper/J has a **pure static join point model**.

6.4.2 Join Point Properties and Join Point Addressing

The JPSL of Hyper/J is much more restrictive than the one in AspectJ. First, there are join points which are selected within the concern mapping file. Furthermore, join points can be selected in the hypermodule (which are based on the selections specified in the concern mapping). As a consequence, a join point selection occurs in the concern mapping as well as in the hypermodule, i.e. the join point selection is not localized in one single construct, but in two. The properties available within the concern mapping as well as within the hypermodule are the same. Consequently, the properties do not differ depending on where they occur. However, the way of how properties are addressed differs between both specifications. Consequently, the discussion of the property addressing is separated into the concern mapping property addressing and hypermodule property addressing.

The next section describes the join point properties in terms of the corresponding design dimensions. Afterwards, the means to address such properties within the concern mapping are described. Finally, the property addressing within hypermodules is described in terms of the design dimensions.

6.4.2.1 Join point properties

In general, Hyper/J provides type names, operation names, and field names as join point properties[103] for type definition and method definition join points. All names are **structured**: Class-names are structured in compliance to the full type qualification as defined in Java [JSGB00]. Operation names are structured according to the type defining the corresponding method and the name of the method. Type information like return type or parameter types are not represented by properties (or part of a property). However, similar to the structure of signatures in AspectJ pointcuts like `call` or `execution`, this structure can be considered simply as different properties. A class has a name property and a package property. An operation (and a field) has a package property, a class name property, and a name property describing the member itself. The name properties are **atomic**, the type properties are **structured**.

All of these properties are **static** and have a **direct correspondence**. Furthermore, they are **local properties**: Type names are locally available at the type definition join points (direct within the syntax), method names of method definition join point are available at each method definition. The same is true for field names of field definition join points. Only the names of packages and types for type join points represent **identity properties** (since types are uniquely identified by their full qualified name). Operations on the other hand are only identifiable via their names. Consequently, Hyper/J does not permit to distinguish between overloaded methods within the same type, since parameter types are not provided as properties. Consequently, there are **no identity** properties for method definition join points.

Method call join points (which are selected and adapted via bracket relationships) have some properties representing the called method as well as the calling method. The called method is encoded with the called type as well as with the method's name. The type is represented by the package name as well as the type's name, the method is represented by its name. This corresponds to the declaring type and method name property of method calls in AspectJ (see section 6.3.2.3). The method name is a **static, atomic, local, non-identity property**[104] with a **direct correspondence**. The target type is a **static**, **structured**, **non-identity property** with a **direct correspondence**. Likewise to AspectJ, the static type of the called object needs to be computed. Although

[103] Hyper/J permits also to specify packages within concern mappings which simply means that all classes within the package are added to the specified concern.

[104] It is a non-identity property corresponding to the previous discussion: Hyper/J does not distinguish between methods defined within the same type but wich differ in respect to their parameter types (overloaded methods).

typing information can be derived from the syntax of the whole application, it cannot be derived from the syntax information locally available at the join point[105]. Consequently, the target type property is **non-local**.

The properties which represent the operations in which the corresponding method calls occur, correspond to the `withincode` property in AspectJ (see section 6.3.2.3). They can be divided into a property representing the calling type via its name (**static, local, structured, non-identity property** with a **direct correspondence**) and the method name property (**static, atomic, local, non-identity property** with **direct correspondence**).

6.4.2.2 Property addressing within a concern mapping

Within the concern mapping all elements are identified directly via their name – all names need to be explicitly enumerated to be assigned to a concern mapping. It is also possible to leave out some values. For example, it is possible to specify only an operation name and select that way all operations in all types having such a name. Also, it is possible to specify only type names and select that way all classes matching this name. Furthermore, selecting classes implicitly selects all methods defined in this class and assigns them to the corresponding concern. However, it is not possible within the concern mapping to specify names only partly (for example by specifying only the first characters of type or operation names). Furthermore, it is not possible to specify common property values for different properties, for example enforcing the operations being selected to have the same name as the type they are contained in.

Consequently, join points are addressed within the concern mapping in a **lexical**, **closed** and **stand-alone** way. Since a concern mapping cannot be overridden or incrementally refined, the selection is furthermore specified in a **monolithic** way.

6.4.2.3 Property addressing within a hypermodule

The property addressing in hypermodules is slightly more advanced than property addressing in concern mappings. The way how properties are addressed differs in three different ways. First, the elements from the concern mapping need to be imported in an explicit way. Second, a number of elements are *implicitly* selected for a certain composition rule due to their underlying properties. Third, the concerns (and also usually class descriptions) are being used within the concern relationships in order to specify some additional compositions that are not already determined by the general composition rule.

The import of the concern mappings is just an explicit demarcation of the elements that are being used for the composition. This is similar to the use of pointcuts within an advice in AspectJ that states what selected join points are being adapted. Consequently, it looks like that the import does not need to be considered. However, the import has a direct impact on the join point selection because of implicit property addressing: Composition rules like merge or override rely on matching elements. Elements match, if they either have an equal name of if they are explicitly declared to match using the match relationship. What elements match is not only a matter of the corresponding join

105 Note that the distinction between source code and byte code becomes important here: The Java byte code contains already static typing information which are constructed at compile-time.

point selection in the concern mapping, it is also a matter of the properties' values. For example, if there are two classes that have the same name within different concerns, the application of `mergeByName` creates a class that combines attributes of both. The characteristic of "matching" is in such a case no explicit property addressing because both classes are not explicitly declared to match. Instead, both classes within the different concerns are implicitly selected for the adaptation because the values of the name property correspond.

However, even this implicit selection can be mapped to the design dimensions because this selection is simply based on a lexical comparison of properties and the selection criterion is "having equal names". Such equality of name is specified without directly referring to the property within the join point selection. However, the elements that adapt the corresponding entities require having a corresponding name. Consequently the adapting element (not the adaptation construct itself) refers to the corresponding property via its name[106]. Therefore, the implicit join point addressing is a **lexical addressing**. Since matching requires class and operation names to be equal, the addressing is furthermore a **closed addressing**. It is not possible to specify any conditions for a number of properties. Consequently, the implicit property addressing is **stand-alone**. Since the implicit selections can refer to all other join points, the selection is **unrestricted**. With respect to the adjustability this selection is **incrementally adjustable** by an application of the match relationship.

For the different relationships (**bracket**, **match**, or **compose**) Hyper/J provides different means to address properties.

In the bracket relationship properties are addressed in different ways depending on what role the corresponding Java entities play for the composition. First, the properties of the target operation (declaring type and operation name) are addressed by specifying patterns that are closely related to the type and method name patterns in AspectJ (see section 6.3.2.2). Hyper/J permits to specify a character sequences partially and provides negation and conjunction operators in order to exclude or include certain character sequences. Consequently, the target class name property as well as the target operation name property is addressed via **a lexical addressing**. Furthermore, a **closed addressing** (in case the complete name is specified) as well as **open addressing** is provided. Furthermore, since it is not possible to specify the same patterns for a number of different properties the addressing is **stand-alone**. Similar to AspectJ, any arbitrary join point can be selected from any existing modules. Consequently, the addressing is **non-restricted**.

The properties describing the class name and (potentially) the operation name within which the corresponding join point occurs is being specified within the *from* clause of the bracket relationships and require a value specification of the corresponding property. Consequently, both properties are being addressed in a **lexical**, **closed**, **stand-alone** and **non-restricted** way.

The match relationship corresponds to the property addressing of the bracket relationship. The target entities that are to be matched are described in terms of a class

[106] Note that in this case the join point selection is not achieved via a Hyper/J selection constructs but via a corresponding definition of the corresponding Java entity.

and operator patterns. Consequently the corresponding properties are addressed in a **lexical, stand-alone**, **closed,** as well as **open addressing** that is **non-restricted**.

The compose relationship differs with respect to bracket and match: The target of a compose relationships needs to be specified completely via its name. For example, if two classes are composed, the name of the target classes to which another class should be added to has to be explicitly specified. Consequently, the properties are addressed in a **lexical**, **closed**, **stand-alone**, and **non-restricted** way.

6.4.3 Join Point Adaptation

In Hyper/J there are mainly two ways of adapting join points. First, classes and methods are adapted because of the applied composition rules (merge and override) which are refined by the composition relationships. Furthermore, the bracket relationships are to be applied, which do not directly refer to the corresponding composition rules.

The general composition rules merge and override are conceptually **structural** as well as **behavioral** join point adaptations. Both kinds of composition rules are structural adaptations because type structures are extended by introducing additional members (which come either from matching types or which are explicitly declared via compose relationships) as well as new super types. Consequently, type definition join points are structurally extended. Likewise, method declaration join points can be structurally as well as behaviorally extended (which differs from AspectJ). A structural extension is possible for example by composing different methods that vary with respect to their exceptions. Figure 6-5 illustrates a structural extension of a method `m` by composing it with an additional method that declared a thrown exception. Other kinds of structural adaptations (like adding parameters, etc.) are not possible for method definition join points.

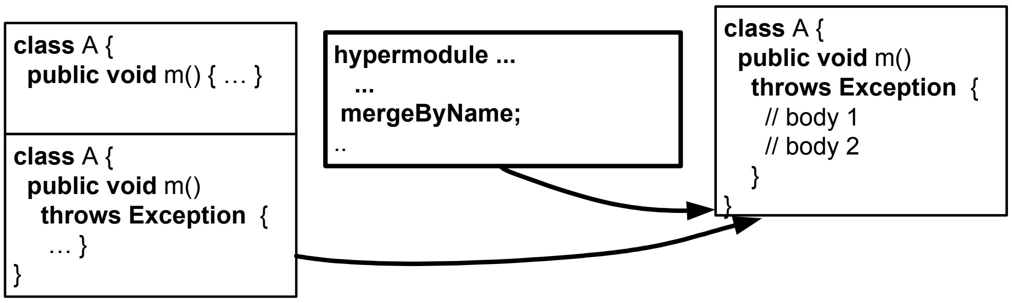


Figure 6-5. Structural method adaptation in Hyper/J.

The composition rules are also behavioral adaptations since matching operations are composed. Original methods can be completely replaced by (or merged with) other methods (i.e. the bodies of different methods are composed). The characteristic of being a structural or behavioral adaptation depends on the corresponding concern the composition rules are applied to. This is in contrast to AspectJ where the characteristic of being a structural or a behavioral adaptation is an inherent characteristic of the underlying kind of language feature.

With respect to the variability, the composition rules provide **non-variable adaptations**. The way how each join point is to be adapted is completely specified by the types being used for the composition and their operations. With respect to the join point abstraction, Hyper/J provides a **fix abstraction**. The context, in which the code is to be executed, is the same as the original one.

The composition rules are **constructive** as well as **destructive** (and will be discussed in the following paragraphs).

Merge relationships are constructive, since different types are merged together to a new type. All type relationships of the old type (to which another one it being merged) are still valid type relationships of the new type (**structural** and **constructive adaptation**). Applying the merge relationship to operations results in merging all method bodies (**behavioral** and **constructive adaptation**).

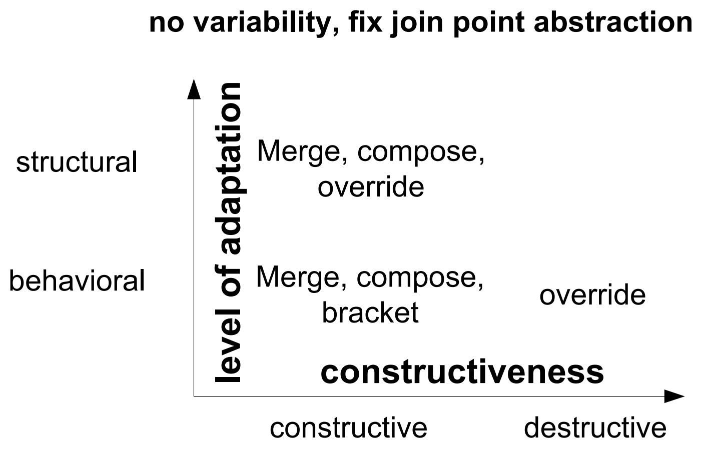


Figure 6-6. Join point adaptations in Hyper/J.

The override relationship is **constructive** as well as **destructive**. If it is applied to types all members in the overriding type become new members of the overridden type. Hence, it is **structural** and **constructive**. It is **behavioral destructive** because all matching operations in the overridden type are replaced and the original behavior at such methods becomes lost. If applied to operations, the overriding operation simply replaces the overridden method.

The design dimensions for the compose relationship is similar to the merge composition rule: Compose adds further type relationships to types or additional operation bodies to existing operations. Consequently, the compose relationships is a **constructive**, **non-variable, structural**, as well as **behavioral adaptation** with a **fix join point abstraction**.

The situation is slightly different for the bracket relationships. This relationship is explicitly designed to adapt static and behavioral method call join points. The underlying structure is not adapted and the original method call is still part of the composed application. With respect to the variability as well as the join point abstraction the bracket relationship is designed in the same way as all other adaptations. Consequently, the bracket relationship is a **constructive**, **non-variable, behavioral adaptation** with a **fix join point abstraction**.

Figure 6-6 summarizes the different means of adapting join points in Hyper/J. Since all adaptation mechanism are equal with respect to their level of variability as well as their join point abstraction, only the dimensions level of adaptation and level of constructiveness are considered.

6.4.4 Weaving

The term weaving applied to Hyper/J describes the composition of hypermodules. Hypermodules are composed by transforming the underlying bytecode. Therefore, an additional representation of the bytecode is being created which is being used by the transformer in order to determine those join points that are actually addressed and adapted (cf. [TO00]). The bytecode transformation is achieved in a compilation-like process that is being executed before runtime. In case operations are being composed in a constructive manner, Hyper/J generates additional methods containing the code of the original methods and generates within the new composed method corresponding method calls. Consequently, the Hyper/J weaver is a **static** weaver based on **code instrumentation**.

6.5 AspectS

A conceptual analysis of AspectS (see section 2.4) in terms of the underlying design dimensions is slightly different than the analysis of Hyper/J and AspectJ: AspectS does not provide additional language features on top of Smalltalk nor does it provide any external tools in order to compose Smalltalk entities. Instead, AspectS is a framework (in the sense as described in [JoFo88]) written in the same language as the corresponding base language. This makes especially the discussion of join point properties and join point addressing slightly different, because it is not possible to speak about new language features, but about framework implementations.

In fact, AspectS hardly provides any additional properties for join points but relies on the metaobject facilities provided by Smalltalk. With respect to the property addressing, AspectS permits to use all Smalltalk constructs – i.e. the whole computational power of the base language – in order to address join point properties. Consequently, mapping AspectS's property addressing to the design dimensions means to map the language constructs of Smalltalk to the design dimensions.

6.5.1 Abstract Join Point Model

AspectS provides (in correspondence to AspectJ) two different kinds of join points due to the different join point adaptations introduction and advice. Introductions are adaptations of class definition join points (**static** and **structural join points**). With respect to advice more discussion is required in order to determine what the underlying abstract join point model is.

In AspectS advices refer to join points described by join point descriptors. Since the join point descriptors refer to method definitions (via the class object and the method selector), it looks like that join points in AspectS are static, structural join points. However, AspectS also provides advice qualifiers that permit to specify additional constraints that also potentially depend on the runtime specific information (like for example call stack information as being provided by `#cfFirstClass`). From this point of view, it is disputable whether advice refer to dynamic join points. Furthermore, AspectS permits to define caller-specific constraints. From this point of view is seems also disputable whether AspectS advices refer to dynamic, behavioral join points.

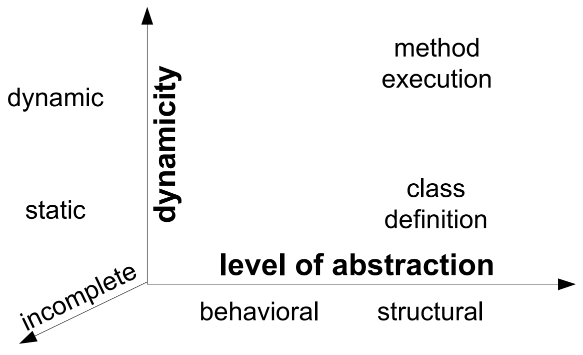


Figure 6-7. AspectS's abstract join point model.

From the developer's perspective it is important that an advice requires a set of method definitions as join points which can also be restricted by runtime-specific information. Consequently, the *felt* join point is a dynamic method definition join point (**dynamic** and **structural join point**). It is a structural join point because of the join point descriptor. It is a dynamic join point because of the dynamic constraints. Nevertheless, in contrast to AspectJ the static properties of the join points (class and method name) have to be specified in order to select the appropriate join points.

Considering the dimension completeness AspectS's join point model is incomplete with respect to dynamic structural elements as well as static structural elements – there are methods in the system that cannot be wrapped. For example primitive operations like `instVarAt:` cannot be adapted.

6.5.2 Join Point Properties and Join Point Addressing

AspectS provides two different ways of addressing join points. First, there are join point descriptors that are initialized with selector names and class objects. Second, there are join point qualifiers which are provided by AspectS and which are declared by the developer for initializing the corresponding advice.

6.5.2.1 Join point descriptors

The selector name part of the join point descriptors is similar to the method name property in AspectJ or Hyper/J for method definition join points: It is a **static, direct, and local property**. The selector is furthermore **structured** because selectors in Smalltalk are lists of identifiers separated by the token" **:** " which implicitly describes the number of parameters of the corresponding method. It is furthermore part of a join point identity (in conjunction with the class object).

The class object is an ordinary Smalltalk object. Nevertheless, this property is directly extracted from the underlying code base. Consequently, the property is a **static**, **direct** and **local property** (which corresponds so far to AspectJ). Furthermore, the property is **structured** (since it is an object). However, in contrast to AspectJ the structure of the class object differs. In fact, since the class object is an object that reflects on the class definition, it provides a large variety of properties of elements due to the large number of fields in the objects (and corresponding methods). Furthermore, the class object is part of the shadow **identity**.

In principle, the class object and selector property can be addressed using the whole computational power of Smalltalk. The selector as well as class object can be addressed via a **direct addressing** (by specifying directly a matching selector or the class object via its name) or via an **indirect addressing** (by specifying a function returning a number of matching selectors or class objects). With respect to its openness, both properties can be addressed in a **closed way** (e.g. by specifying only the selector's characters or the class's name) as well as in an **open way** (e.g. by specifying a function which performs a computation on the base program). With respect to property sharing AspectS permits to specify **shared** values among properties e.g. by specifying a function that computes class objects as well as selectors at the same time. Due to the ability to perform incremental modification provided by the base language, join points can be addressed in an **incremental way**. Finally, since arbitrary class objects and selectors can be selected the join point selection is **unrestricted**.

6.5.2.2 Advice qualifiers

Aspect's advice qualifiers reveal additional properties that can be addressed by the developer.

The qualifiers `#senderClassSpecific` and `#senderInstance Specific` permit to restrict the join point selection to certain sender instances and classes. Conceptually, the underlying properties for the sender object as well as the sender class are **dynamic** ones that have a **direct correspondence**[107]. Since both properties have objects as values they are **structured**. From the executed method's perspective both properties are **non-local** since information from the call stack are needed. However, such information is directly available in the underlying base language[108]. None of the properties contributes to the join point's identity (since the underlying join point is a dynamic method join point). Obviously, the sender instance property is a **data property** while the sender class property is a **metadata property**. Furthermore, the properties are **past properties** since they refer to previous objects in the call chain.

The conceptual properties underlying `#receiverClassSpecific and` `#receiverInstanceSpecific` corresponds to the `this` property of method execution join points in AspectJ: It is a **dynamic**, **local**, **current** and **structured data property** with a **direct correspondence**.

The sender class, sender instance, receiver class, as well as receive instance properties are addressed in the same way: The developer needs to pass the corresponding object to the aspect in order to select the underlying join point. Consequently, the way of how such properties are addressed corresponds to the join point addressing of the class object property as described in the previous section: **Direct** as well as **indirect**

[107] On the implementation level both properties are represented by the same data that is the last element on the call stack.

[108] Note that this view on the locality characteristic corresponds to the discussion in section 5.4.3.4: according to the conceptual model of method definitions call stack information are considered to be non-local.

addressing, **closed** as well as **open** with the capability to address the properties in an **incremental** and **shared** way.

The other advice qualifiers (`#cfFirstClass`, `#cfFirstInstance`, `#cfAllButFirstClass`, `#cfAllButFirstInstance` , `#cfFirstSuper`, and `#cfAllButFirstSuper`) differ substantially from the previous ones: They do not require to specify any addition values in contrast to the previous ones.

The conceptual properties underlying such qualifiers can be interpreted in two different ways:

- First, the properties are boolean properties whose values determine whether a certain hard-coded condition is fulfilled. The `cfFirstClass` property determines whether the current class occurs for the first time on the call stack, the `cfAllButFirstClass` property corresponds to the negation of the first class property, etc. By adding such a qualifier to an advice the developer states that the corresponding condition should be fulfilled.
- Second, the advice qualifiers simply represent a way of addressing the call-stack property, i.e. there is only one property that represents the call stack[109].

From the first perspective, all properties have the same characteristics. First, all of them are obviously **dynamic properties**, because whether or not the current call stack fulfills the corresponding characteristic depends on runtime-specific information. The corresponding boolean value represents the result of a computation on the call stack. Consequently, the properties are **abstract** and **atomic** with **an indirect correspondence**. For the same reason the property is a **data property**. With respect to the locality as introduced in section 5.4.3.4 all of these properties are non-local, since they depend on call-stack information. However, such information is directly available in the underlying base language. With respect to the join point identity none of the properties contributes to the join point identity (**non-identity property**).

From this perspective the properties are addressed in a direct and closed way by the corresponding qualifiers. Due to the ability to push the qualifiers into methods that can be used by different aspects and overridden using subclassing, **shared addressing** and **incremental selection** is permitted. Furthermore, the addressing is **unrestricted**.

In the second interpretation the property represents the call stack that is directly available as a data-object in Smalltalk. Consequently, the property is a **dynamic**, **structured**, **non-local**, **direct data** property representing **current data**. From this interpretation the join point qualifiers address the property in an **indirect**, **shared**, **incremental** and **unrestricted** way.

However, from the developer's point of view only the way of how the addressing is achieved is important. Consequently, the first interpretation is considered to be appropriate.

[109] On the implementation level the second interpretation is being used: the qualifiers refer to the call stack which is in Smalltalk directly available using the keyword `thisContext`.

6.5.3 Join Point Adaptation

The join point adaptation in AspectS with respect to the design dimensions is closely related to AspectJ. First, there are introductions referring to class definition join points. Second, there are advices referring to method execution join points.

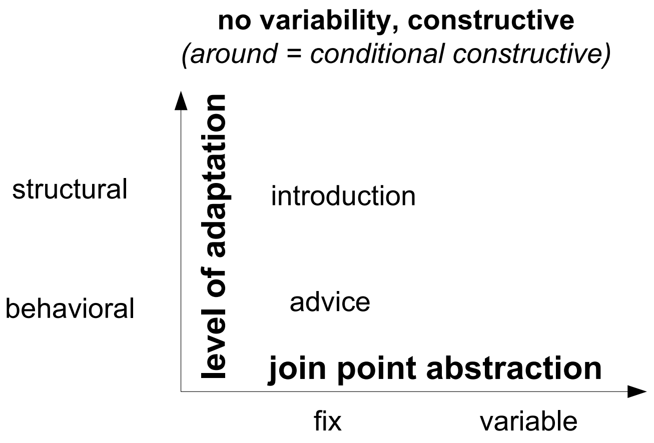


Figure 6-8. AspectS join point adaptation.

Introductions are (in correspondence to AspectJ and in correspondence to the discussion in Chapter 3.2) **structural** and **constructive adaptations**. In correspondence to AspectJ the adaptation has a **fix join point abstraction**: The context within the adaptation does not vary depending on the corresponding selection but is determined be the target objects and the declaring aspect. However, the join point abstraction differs from AspectJ: While in AspectJ the context being used by an introduction corresponds to the context available for each method, the context of an AspectS introduction refers to the defining aspect because of the self-reference within blocks[110].

Furthermore, the adaptation is non-variable (which also corresponds to AspectJ). However, a non-variable adaptation in Smalltalk is different than a non-variable adaptation in Java. Due to the strong-typing characteristics of Java (and in AspectJ) every expression defined in the adaptation requires to be typed. Consequently, all types participating in a collaboration defined within the adaptation need to be known. Since Smalltalk is an untyped language, such restrictions do not exists in Smalltalk dialects like Squeak underlying AspectS.

The underlying design decisions of advice are also very similar to AspectJ. First, advices are **behavioral adaptations**: An advice does change the structure (the signature) of the underlying method but only its behavior. Second, before and after advice are **constructive** and while around advice are **conditional constructive**.

However, there is a difference between advices in AspectS and AspectJ considering the join point abstraction: All join points addressed by the corresponding abstraction

[110] See section 2.4.1 for a detailed description of the self-reference handling in introductions and advice.

constructs have the same context. All parameters of method calls are being delivered to each advice. Consequently, AspectS as a **fix join point abstraction**.

6.5.4 Weaving

AspectS weaves aspects via changing metaobjects (compiled methods): A compiled method is being replaced with a different objects (the wrapper) which executes the corresponding blocks and potentially forwards messages to the wrapped compiled method. The original compiled method still exists after wrapping but has a different position in the corresponding method dictionary.

Creating a wrapper (and establishing in that way the connection to the aspect-oriented system) is done dynamically just at the time when an aspect receives an install message. Consequently, AspectS provides **dynamic weaving**. The underlying code representation of the participating classes is being changed, since the class's method dictionary changes. Consequently, weaving in AspectS is based on **code instrumentation**. Starting the weaving process requires an explicit statement within the application's code. Consequently, the dynamic weaving is **user driven**.

6.6 Sally

Likewise AspectJ and AspectS, Sally clearly distinguishes between join point selection and join point adaptation (see Chapter 3). The pointcut language permits to specify the join points that are to be adapted. Sally provides two different kinds of join point adaptation: Wrappers and introductions that refer to different kinds of join points. However, as discussed in Chapter 3 the underlying pointcut language is being used for wrappers as well as for introductions. Consequently, it is not necessary to distinguish between different kinds of adaptations for the underlying properties and the corresponding property addressing.

6.6.1 Abstract Join Point Model

Sally provides introductions and wrappers. Introductions permit to add additional members or additional supertypes and wrappers permit to change the behavior at certain join points. In contrast to for example AspectJ, Sally does not permit to refer to any runtime-specific information within the selection. Consequently, the underlying join point model is a **pure static join point model**.

Introductions are applied to types: Java interfaces or class definitions represent the corresponding join points. Consequently, the underlying join points are **static** and **structural**. Wrappers are applied to field accesses (field reads as well as field assignments), method calls, as well as method definitions. The first two kinds of join points are **static**, **behavioral** join points while the latter ones are **static**, **structural** join points. Both, behavioral as well as structural join points in Sally are **incomplete**, because a number of behavioral as well as structural elements are not covered by Sally (e.g. operators as behavioral elements, and field declarations as structural elements). Figure 6-9 summarizes the underlying abstract join point model of Sally.

6.6.2 Join Point Properties and Addressing

The properties available for each join point correspond to the facts available for each join point in the underlying logic programming language provided by the corresponding predefined pointcuts. The available properties are closely related to the static join point encoding of AspectJ. Consequently, all properties are rather briefly described in terms of the underlying design dimensions here. Since Sally is based on a static join point model, all properties are static. Hence, this characteristic is no longer mentioned in the following description.

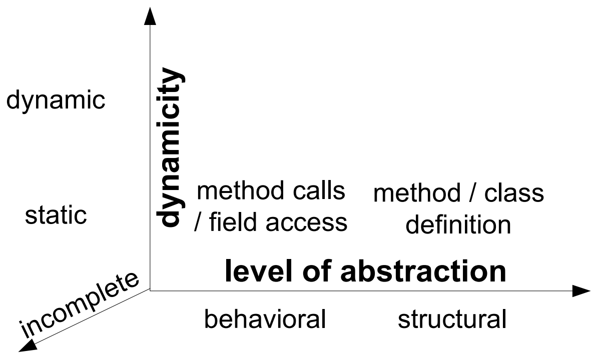


Figure 6-9. Sally join point model.

All source code elements that are selectable using the corresponding pointcuts (for example `typeDeclarations`, `methodDeclarations`, etc., cf. section 3.4.2.3) are equipped with a corresponding identity property. Obviously, this property represents a **local**, **non-structured**, **identity** property with an **abstract correspondence**.

The properties of type declarations, method calls, method declarations, field access and field assignment properties correspond to the static properties available in AspectJ. For example, the method call join point is equipped with a list of parameter types representing the declared parameter types of the method which is to be invoked in this location. This property has is **structured**, has a **direct correspondence** and is **non-local** (see section 6.3.2.3 for the corresponding discussion on the mapping of such properties).

Additional properties available in Sally are those ones that determine the static types being used within expressions. The corresponding pointcuts are `callExpressionTypes` and `fieldSetType`. However, from the design dimension's perspective there is no difference between such properties and the previous mentioned ones. The `pointcutCallExpressionTypes` reveals a list of types that occur on the client side while `fieldSetType` reveals a single type. Both properties have a **direct correspondence** and are **non-local** (see section 6.3.2.3 for the corresponding discussion on type information with respect to the design dimensions). While the type list provided by `pointcutCallExpressionTypes` is **structured**, the single type property provided by `fieldSetType` is **atomic**.

In contrast to the previous mentioned approaches the design dimensions of the property addressing in Sally can be relatively easily explained without the need to differentiate a number of exceptions. The reason for this lies in the uniform way of how

Sally addresses all join point properties using the underlying logic programming language independent of what information the addressed properties represent[111].

Sally permits to address all properties in a lexical as well as in an indirect way. A lexical addressing is achieved either via specifying a property **lexically** within the corresponding pointcut definition or **indirectly** via setting a variable for the corresponding property and specifying additional constraints on the variable in a different pointcut[112]. In the same way it is possible to address join point properties in a **closed** as well as in an **open** manner: Closed value addressing is achieved for example via a direct declaration of the corresponding property while an open addressing is achieved for example achieved via a variable that refers to all types in the system[113]. Consequently, Sally provides all kinds of join point addressing according to the design dimensions as illustrates in Chapter 5, Figure 5-36.

With respect to the design dimensions adjustability and scope Sally permits an **incremental join point selection** (due to the inheritance of pointcuts in correspondence to AspectJ) as well as an **unrestricted selection** (since any arbitrary join point can be selected from within an aspect). Furthermore, Sally permits a **shared value specification** by using common logic variables for different properties.

6.6.3 Join Point Adaptation

Sally provides introductions to adapt static structural join points, and wrappers to adapt behavioral as well as structural join points.

Introductions are (in correspondence to introductions in AspectJ) **structural** and **constructive** adaptations, because the corresponding types are increased. Furthermore (in correspondence to AspectJ) introductions in Sally have a **fix join point abstraction**, because the context of the adaptation is determined by the underlying join point and not by the corresponding selection. However, in contrast to AspectJ, introductions are **parametric**.

Wrappers in Sally are pure **behavioral adaptations** (although the corresponding join point for method wrapper is a structural join point). Wrappers are **conditional constructive** since they permit to refer to the original join point using the `wrap`-construct (see section 3.4.3). Wrappers also correspond to introductions with respect to the level of variability as well as to the join point abstraction. First, they are parametric because code elements can be imported from the corresponding pointcut specification. However, the context in which the adaptation is to be executed is fix, because the runtime parameters being accessible within wrappers are determined by the kind of wrapper (see section 3.4.3 for the available runtime parameters in wrappers). Consequently, wrappers have a **fix join point abstraction**. Second, the code blocks to

111 This is for example in contrast to AspectJ's + operator that is only applicable to type patterns.

112 Note that in that way it is also theoretically possible to specify a join point identity in a lexical way. However, although theoretically possible, there does not seems to be any reasonable case to do this because the join point identity is created and maintained by Sally and it is not the identity's purpose to be used in a lexical manner.

113 In such a case adding new types to the system binds new types to the corresponding variable. Consequently the specified property also evolves in parallel with the base system itself.

be executed are **parametric**, because static parameters can be passed from the specified pointcut to the wrapper. Figure 6-9 summarizes the mapping of Sally's join point adaptation with respect the to corresponding design dimensions.

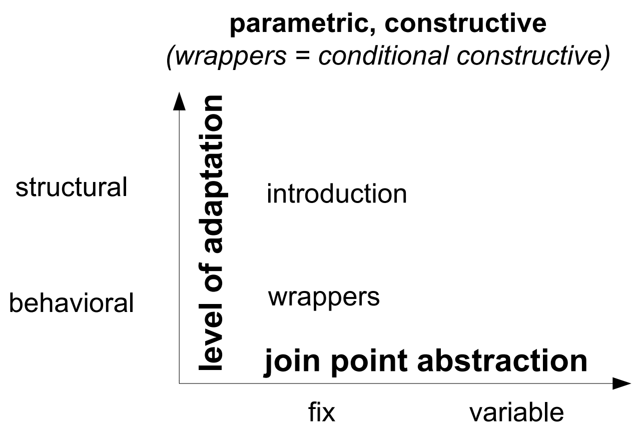


Figure 6-10. Join Point Adaptation in Sally.

6.6.4 Weaving

Weaving in Sally is performed via a code transformation of all classes participating in a crosscutting relationship. Based on the pointcut descriptions and the corresponding join point adaptations the target join points are computed and adapted before generating the corresponding bytecode. While introductions simply add new elements to the type hierarchy, wrappers add additional methods representing the aspect-specific behavior. Consequently, Sally's weaver is a **static** weaver based on **code instrumentation**.

6.7 Morphing Aspects

Morphing aspects (see Chapter 3) do not represent a new system in a sense that a new software framework or system is being provided for selecting and adapting join points: The proposed implementation in Chapter 3 is based on the system AspectS. Consequently, the underlying design dimensions of join point model, join point properties, join point addressing and weaving correspond to the description for AspectS provided in section 6.5.

However, as noted in Chapter 3, morphing aspects are not considered to be a implementation technique specific for AspectS but a technique that has a number of requirements to the underlying systems (see section 4.7): Based on the design dimensions it is possible to define more specific the required characteristics. First of all, in order to simplify the specification of the morphing process a **dynamic join point model** is desirable. This permits to specify the join point selection using the corresponding selection constructs of the underlying system[114]. In case the underlying

[114] However, in fact the existence of a dynamic join point model does not guarantee that the corresponding join point dependencies can be expressed in terms of the provided join point selection constructs. For

join point model is a **static** one, it is necessary that the system provides **constructive join point adaptation** mechanisms. In such a case join point checks need to be specified within the join point adaptation that invoke the original join points in case such checks fail.

The approach of morphing aspects mainly addresses the reduction of join point checks. Consequently, the join point checks must not be added statically (since otherwise there would not be any benefit in using morphing aspects). Instead, the physical adaptation of join points is required just in time when they are needed. Consequently, the underlying weaver must be based on **dynamic code instrumentation**. Furthermore, the morphing process is a developer specified computation of join points that are to be adapted during morphing. Consequently, it is necessary that the weaver permits a **user-driven weaving** being specified within the join point adaptation of join points triggering the morphing process.

The terminology of how morphing aspects are to be applied by utilizing dependencies among join points (see section 4.3.1) and the corresponding join point process is to be specified (see section 4.3.2) can be described in terms of the corresponding design dimensions.

The independent join points (i.e. those join points which do not depend on any other join point) require to be woven directly. Consequently, developers need to specify a corresponding selection for this statement. How this selection cannot be determined on the abstract level upfront without a given concern at hand. For a specific concern (like the tracing example, see section 4.2.1) and a corresponding implementation (see section 4.4) the underlying properties are **static** (method and class names), **lexical properties** with a **direct correspondence** (due to the Smalltalk specific representation of selectors, the selector is **structured** while the class name is **atomic**). The corresponding property addressing is a **lexical**, **closed**, **stand-alone** addressing.

The dependent join points are to be computed using the reflective capabilities of the underlying language. Consequently, morphing aspects explicitly encourage an **indirect** value addressing: Values of properties are computed using reflection and a corresponding join point selection is to be specified (and within the morphing process code that represents the **user-defined weaving** is being executed). Furthermore, the properties are (typically) addressed in an **open** way. In the tracing example not all properties are known whose corresponding join points are to be adapted. Also, morphing aspects encourage the use of **shared value addressing**: The result of the reflective computation is being used to address a number of properties.

6.8 Roles in Terms of Design Dimensions

The concept of roles (cf. for example [KrØs96, Kris96, GSR96]) is very closely related to aspect-oriented software development (see for example [HaUn02b, HSU05b] among many others for an exhaustive discussion). However, since an underlying conceptual model of aspect-oriented systems was missing, it was hardly possible to give a

example, the tracing implementation as proposed in 4.4 was implemented by specifying additional join point checks within the join point adaptation. Although technically possible, this is rather considered as a workaround for missing dynamic selection critera instead of a conceptual clear solution.

conceptual description of the parallels and differences between aspect-oriented approaches and the role concept.

This section shortly introduces the role concept as introduced in [KrØs96, Kris96] and explains this concept in terms of the aspect-oriented design dimensions. Although there are implementations available that correspond to the roles concept from [KrØs96, Kris96] (cf. for example [NeZd99, Zdun01]) the concept can be partly mapped to the design dimensions without referring to a specific implementation.

6.8.1 Introduction to Roles (the Kristensen Perspective)

Roles are temporary views on an object. A role's properties can be regarded as subjective, extrinsic properties of the object the role is assigned to. During its lifetime an object is able to adopt and abandon roles. Thus, an object's environment can access not only the object's **intrinsic**, but also its **extrinsic** properties. In [KrØs96, Kris96] some characteristics of roles are articulated:

- **Identity**: An object and its actual role can be manipulated and viewed as one entity.
- **Dynamicity**: Roles can be replaced during an object's lifetime.
- **Dependency**: Roles only exists together with its corresponding object.
- **Extension only**: A role can only add further properties to the original object, but not remove any.
- **Multiplicity**: An object can have more than one instance of the same role at the same time
- **Abstractivity**: Roles are classified and organized in hierarchies

In [GSR96] Gottlob et al. emphasize another characteristic of roles:

- **Behavior**: A role may change an object's behavior.

The feature **abstractivity** emphasizes that roles are well planned and organized in hierarchies similar to object-oriented ones. On the other hand, this characteristic insinuates that the role concept is highly connected to class-based programming languages and hence roles are classified by classes. Nevertheless, it should be emphasized that role concepts can also be used in class-less object-oriented programming languages[115].

An important characteristic of roles is that roles are dynamically added to objects whereas a role itself has properties (fields and methods). Hence, the accessible properties of a single object differ from perspective to perspective and from time to time. The **root object** describes the intrinsic object, i.e. the original object without any roles. A **role object** is the instance of a role, which is added to a certain root object. A *role* is a generalization of its roles similar to classes. For reason of simplification this thesis uses the term role instead of role object except in situations where it is necessary to stress the difference. A **subject** is a special perspective on a root object including

[115] In fact, language features of prototypical languages are very close to language features provided by role systems.

(some of) its roles. A root object has several subjects whereby every subject contains a different set of included roles. The interface of a subject is an aggregate consisting of the root object's interface plus every role object's interface.

One interesting property of roles (in comparison to aspect-oriented programming) is the behavior characteristic. A role when added to an object may change the object's behavior. While [GSR96] describes this as an intrinsic role feature, [KrØs96] and [Kris96] regards it as a special feature of role that they call a **method role**. A method role is a role's method that is bound to an intrinsic method of the root object. It is important to emphasize that the cardinality between intrinsic method and method role is `1:n`, i.e. every intrinsic method may have several method roles, but every method role has exactly one intrinsic method.

If and how a method role changes an object's behavior depends on what kind of method role it is. There are method roles, which alter a root object's behavior because the object's user is aware of the role, i.e. the role containing the method role is part of a subject used by the user. In that case this thesis calls the role method to be **subjective**. In the other case there are method roles that replace the root object's behavior independently of the user's perspective. Although such a behavior is not part of the root object's intrinsic behavior it is independent of a user's perspective. This thesis calls such method roles **non-subjective**.

It is obvious that there are several conflicting situations, since more than one role can be assigned to an object. Whenever an object's intrinsic methods are invoked the underlying environment has to determine if and how the corresponding roles influence the resulting behavior. The following conflict situations occur:

- *Multiple subjective method roles*: There is more than one subjective method role assigned to the invoked method.
- *Multiple non-subjective method roles*: There is more than one non-subjective method role assigned to the invoked method.
- *Mixed method roles*: There are at least one subjective and one non-subjective method role assigned to the invoked method.

Furthermore, there is a conflict if there are at least two members from different roles with the same selector within the same subject. If an object uses such a selector to access a member it has to be determined what member to choose.

Figure 6-11 illustrates a person that has two jobs in parallel as a bartender. A job is a temporal role, because persons usually do not keep their job for the whole lifetime. The person has some properties like name and day of birth, which are not influenced by any role. On the other hand there are the properties phone number and income. The phone number is on the one hand an intrinsic property, because it describes a person's private phone number. On the other hand it is an extrinsic property, because it describes a phone number specific to the bartender role (and contains a pub's phone number). If the phone number property is implemented as a method then the bartender's phone number methods are subjective since if someone asks a person in private for his number he expects to get the private number but if he asks a bartender for his number he expects to get the bar's number. On the other hand, a person's income directly depends on the income at his jobs. So the income methods of both bartender roles are non-subjective roles methods.

Although the benefit of role concepts has been accepted widely, popular object-oriented programming languages like C++ or Java do not support roles as first class entities on language level. The reason for it is quite simple: The underlying assumption for static typed, class-based programming languages is that an object's properties are entirely known at compile-time and can therefore be classified. Hence, class-based programming languages do not distinguish between intrinsic and extrinsic properties ([Lieb86] discusses this topic in detail). Therefore additional techniques are needed to support roles in class-based languages.

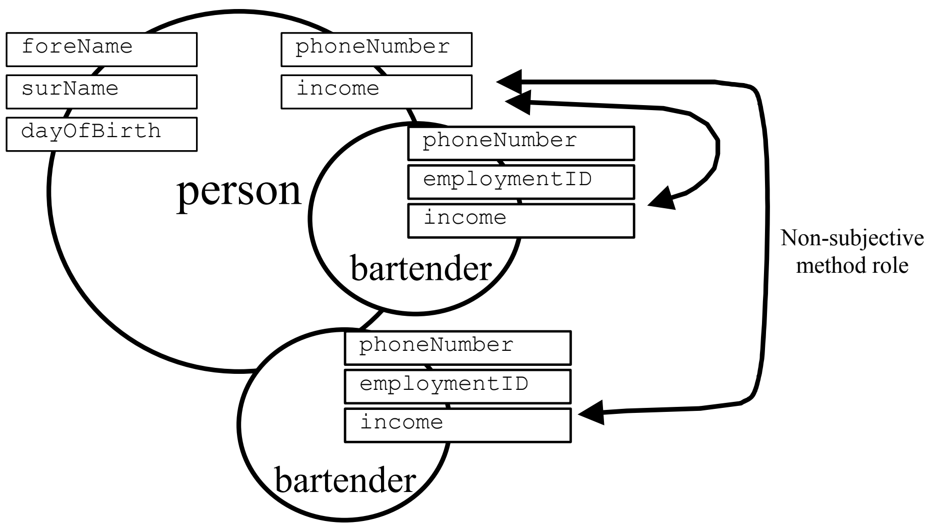


Figure 6-11. Person object with two bartender roles.

6.8.2 Abstract Join Point Model

The join point model of roles comes from the two different ways of the impact of roles on the application. First, roles applied to a root object introduce new members that are accessible via the corresponding subject. Consequently, the adaptation is performed on objects, which are dynamic structural join points. Second, methods role are behavioral adaptations of object-specific methods. Such object-specific methods represent dynamic structural join points.

Figure 6-12 illustrates the corresponding abstract join point model of roles with respect to the design dimensions of abstract join point models. Since only objects and object methods are available as join points, the join point model is furthermore incomplete.

6.8.3 Join Point Properties and Join Point Addressing

Because of both different kinds of join points (objects via roles and object methods via method roles) there are two different kinds of properties and different kinds of how the properties are addressed.

Join point properties in role systems are first of all the root objects to which a particular role is assigned. This property corresponds to the root object's identity (or a reference to the root object). Furthermore, the target method of a role method is the join point which is represented by join point properties and addressed by corresponding addressing constructs.

The target object is obviously a **dynamic property** (because objects appear at the system's runtime), which has a **direct correspondence**: The object itself is the property, which corresponds to the object as it appears runtime. The property is furthermore (in all programming languages with an implicit management of object identities like Java or C++) **atomic**. Since each object has knowledge about its identity property it is a **local property**. The object itself furthermore obviously represents a **current data property**.

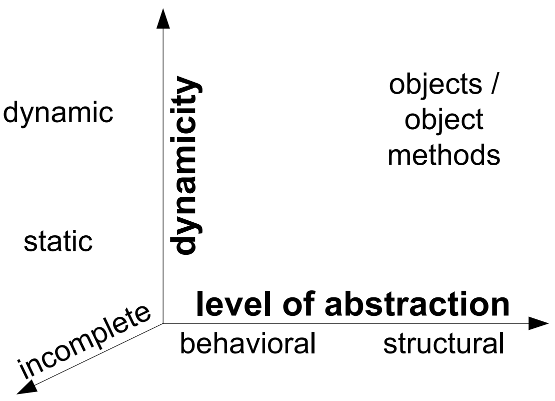


Figure 6-12. Join Point model of roles.

The property is being addressed **lexically** as the value of the property has to be specified, i.e. the very object has to be delivered. However, the computation of the corresponding root object is not determined by certain selection constructs of the underlying programming language. Instead, the full computational power of the base language can be used to determine the object to which a role is to be added. Beyond that, the property can be addressed in an **open** way as roles can be applied to arbitrary objects using the full computational power of the base language. Due to the same reason, the addressing is **shared** as well as **incremental**.

The properties encoding the target method join points consist on the one hand of the target object as described above, and properties describing the method itself. The way of how signatures are represented depends strongly on the underlying object-oriented language as well as on the role implementation itself: On the one hand, all known object-oriented languages identify methods via their name[116]. Such a method name corresponds to the method name property of AspectJ or Hyper/J (see sections 6.3.2.3 and 6.4.2.1): It is a **static**, **atomic** and **local** property with a **direct correspondence**[117]. Whether or not the method name is part of a shadow identity also depends on the underlying language (as also discussed in 6.3.2.3 and 6.4.2.1): In case the roles implementation is based on a programming languages where a method name itself

[116] In Smalltalk such a name also determines the number of arguments for the corresponding method.

[117] In correspondance to section 6.5.2.1 the method name (or selector) is **structured** in Smalltalk systems.

does not identify the method itself (like Java) the property is a **non-identity property**, otherwise it is a **identity property**[118] in conjunction with the corresponding root object.

In the role model as introduced in [KrØs96, Kris96], the method names need to be addressed in a **lexical** way: The names defined in the method roles correspond to the method names as they appear in the root object. Furthermore, each property needs to be specified in a **closed** way (because each method role has exactly one intrinsic method, see the previous section).

Since in the approach described in [KrØs96, Kris96] there are only two properties available (object identity and method name) and both properties differ in their nature, there is no value sharing between both properties (**stand alone property addressing**). The addressing is furthermore **unrestricted** (since all arbitrary objects and methods can be defined) as well as **monolithic**.

6.8.4 Join Point Adaptation

Roles provide two different ways of adapting join points: Either objects are adapted in a **structural way** (by introducing additional methods and fields) or object methods are adapted in a **behavioral way** (since methods roles change the behavior of methods without changing a method's structure). The structural adaptation of objects is constructive since only additional members can be added but no members can be removed. Most implementations permit to refer from within the method roles to the methods they replace in order to invoke the overridden behavior[119]. Consequently, the behavioral adaptation is **conditional constructive**.

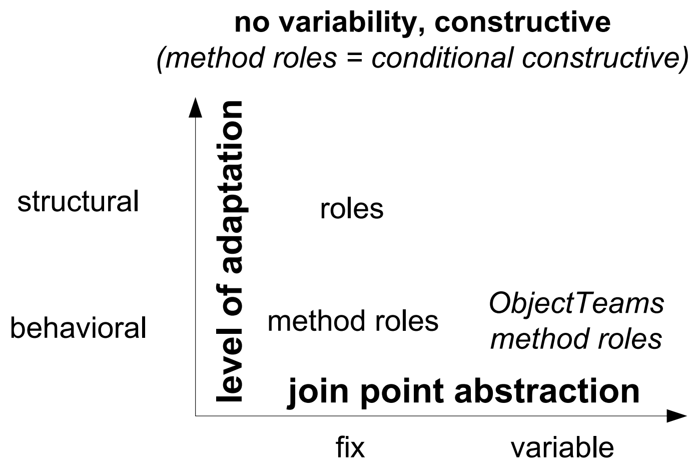


Figure 6-13. Join point adaptation of roles.

According to most implementations that do not provide any generic constructs, both kinds of join point adaptations are **fix**. With respect to the join point abstraction, the

118 In some role implementations such as ObjectTeams [Herr02] the method name is part of the identity property (together with the object id) although the underlying language provides method overloading.

119 See for example the `next`-statement in [NeZd99].

structural as well as the behavioral adaptation have a fix abstraction: Depending on the concrete implementation the code within roles either has the role object itself or the whole subject as a context. Method roles have the role object or the whole subject as their context plus the method's parameters. Since such parameters correspond to parameters of the root object's method and cannot be refined by the developer, the corresponding join point abstraction is fix.

It should be noted, that there are role implementations that provide also variable join point abstractions for method roles. For example, ObjectTeams [Herr02] implements the role concept and permits to chose among the variables of a method which one should be imported to the current implementation[120]. Furthermore, it should be noted that there are static role models that permit to adapt only classes, but not single objects. In such cases, the underlying join point model is static (but the instantiation for the other dimensions is still the same).

Figure 6-13 summarizes how the role concept relates to the design dimensions of join point adaptation.

6.8.5 Weaving

The way of how the role concept is implemented cannot be determined by the conceptual view on roles. However, most implementations (such as [Herr02, Zdun01]) implement roles via a dynamic weaver based on code interpretation. The reasons for such an implementation are quite simple:

- First, it is an inherent characteristic of roles according to [KrØs96, Kris96] to be dynamic. Consequently, the implementation of a static weaver seems to be rather inappropriate.
- Second, implementing roles by code instrumentation either requires to determine all potential shadows upfront (in case a static weaver is considered for the implementation), or to perform the code instrumentation has at runtime. This is typically a time-consuming task (see for example the measurement of adapting method implementations in AspectS in section 4.5).

Furthermore, since the attachment of a role to a root object is determined by code being executed at runtime, the underlying implementation is achieved via a **user-driven weaving**.

6.8.6 Discussion and Conclusion

According to the previous discussion, the roles concept can be described in terms of the proposed design dimensions. According to this, role implementations can be considered as very special kinds of aspect-oriented systems (see furthermore e.g. [HSU05b, HaUn02b] for further discussions as well as additional literature on that topic).

[120] However, it should be noted that the context can only be extended up to to the same parameters as passed to the corresponding method. Hence, it can be rather argued that ObjectTeams provide a very restricted form of variable join point abstraction.

However, roles (based on the model as introduced in [KrØs96, Kris96]) do not provide a strict separation between base language constructs and join point selection language constructs. Instead, the base language is being used to determine the root object to which objects are assigned. In contrast to most other approaches, the selection of the target object is not specified using declarative language constructs. Nevertheless, the declaration of the target method names within the method roles is a declarative construct that directly specified a property of the corresponding join point. The language construct for this declaration corresponds in most implementations to the same construct for specifying signatures for methods.

However, since roles have been introduced as a concept without referring to a special implementation, their mapping to the design dimensions is slightly incomplete. For example, the available join point properties for the role concept cannot be directly determine from a pure conceptual point of view, because the design dimensions were build to describe concrete systems on an abstract level. For the same reason, it cannot be determined (without referring to a concrete implementation) what the corresponding implementation details are in terms of weaving. However, as argued in section 5.6 weaving is considered as an implementation specific detail. Consequently, the weaving itself is rather less important for a conceptual understanding of a certain system.

6.9 Chapter Summary and Conclusion

This chapter mapped the design dimensions as introduced in Chapter 5 to different systems: AspectJ, Hyper/J, AspectJ, Sally, and Morphing Aspects. Furthermore, the role concept was explained in terms of the aspect-oriented design dimensions.

Since all of such systems are commonly accepted called aspect-oriented approaches, the mapping represents a case study with respect to the applicability of the design dimensions. Furthermore, the mapping explains the design dimensions in terms of concrete systems, which eases the understanding of the design dimensions for developers who are familiar with one of the previous mentioned systems.

Explaining morphing aspects as well as the role concept showed, that it is not completely possible to map a pure conceptual approach to the design dimensions. This is caused by the fact that the design dimension's purpose is to explain design decisions of aspect-oriented systems with a concrete implementation. Consequently, some design dimensions cannot be determined for a certain concept without a concrete implementation. However, for morphing aspects as well as for the role concept the application of the design dimensions turns out to be useful. For morphing aspects the requirements of the underlying system as well as the morphing process could be explained in a more concrete way based on the vocabulary provided by the design dimensions. For the role concept the abstract characteristics of the role concept could be explained in terms of the design dimensions and the deviations of role implementations from this concept could be expressed in terms of the design dimensions.

7

DESIGN DIMENSIONS-BASED COMPARISON AND SELECTION

7.1 Introduction

The design dimensions proposed in Chapter 5 represent a conceptual framework for the design of aspect-oriented systems – aspect-oriented systems and their ingredients can be understood in terms of these design dimensions. However, from the dimensions themselves it is not directly clear how they can be used in order to compare different aspect-oriented systems or to assess aspect-oriented systems in order to determine what systems are adequate to solve a given crosscutting problem.

This chapter shows that the design dimensions help to compare aspect-oriented systems as well as to estimate the appropriateness of a certain system (see also [HSU05a]) in order to handle a given crosscutting problem.

Section 7.1 discusses the term phrase *appropriateness of a system to modularize a given crosscutting concern*. Section 7.2 discusses the usability of the design dimensions in order to compare aspect-oriented systems and to determine their appropriateness. Section 7.3 applies the design dimension in order to select aspect-oriented systems. Section 7.4 summarizes and concludes this chapter.

7.1 Appropriateness of Aspect-Oriented Systems

In order to determine whether a certain system is appropriate to modularize a given crosscutting concern, it is necessary to state what exactly the term appropriate means in this context.

In general, it is possible that an aspect-oriented system provides technically the ability to solve a crosscutting problem. However, potentially this is achieved without providing the ability to specify a selection criterion that reflects on the conceptual selection criterion underlying the developer's intention within its join point language[121].

[121] See also [SHU05] for a discussion on the conceptual mismatch between the developer's conceptual model and the actually specified selection criteria.

For example, a Java-based system that permits to adapt method call join points and that provides only method names as properties for such join points does not permit to distinguish between the different (static or dynamic) target types of the method call. Consequently, if developers are interested in selecting all method calls to a certain type, they need to specify on the one hand the corresponding method names within the join point selection. On the other hand, the developers need to specify additional constraints within the join point adaptation (in AspectJ for example using the **pointcut method idiom**, cf. [HSU03]) in order to determine whether the current join point corresponds to the desired one[122].

Such a solution is valid in the sense that the aspect-oriented system permits that way to intercept messages at certain join points in order to perform additional functionality. However, such an approach needs to be considered as rather inappropriate: The join point selection constructs turned out to be insufficient to specify the selection criteria. Instead, the adaptation itself contains additional statements that implement parts of the join point selections. Hence, the selection constructs no longer reflect on the conceptual model of the join point selection and the join point adaptation becomes part of the selection. Consequently, such an approach does not cleanly separate between the join point selection and the join point adaptation.

Another example where developers can build some kind of workaround in order to handle a certain crosscutting problem is to adapt a structural join point although a behavioral join point is the actual focus of the adaptation. For example, in order to determine in AspectJ whether an object changes its state it is possible to select and adapt the corresponding behavioral join point – this thesis' considered this as the *right* join point, because this is actually the point in the execution of a program where the state change occurs. Another alternative is to select and adapt the corresponding methods containing such behavioral join point based on a given set of naming conventions (via so-called set-methods, i.e. methods whose name begin with the prefix `set`) [123] that indicate such a state change.

This thesis considers the second alternative to be inappropriate: Instead of referring to the original join point that represents the situation where an object is about to change, a join point is chosen that is only *closely related* to the real join point. Furthermore, the problem is that the fulfillment of the naming conventions is already an implicit join point selection: The developers of a given class already know that state changes are of a special interest for a different developer (because this is the main reason for establishing such a naming convention). Consequently, if for some reasons the naming conventions are not followed the aspect-oriented system is not able to adapt the right join points.

This thesis considers an aspect-oriented system to be appropriate in order to modularize a given concern if

122 The **pointcut method pattern** [HSU03] is typically applied whenever AspectJ's pointcut language turns out to be insufficient for selecting a join point. In such situations the corresponding advice invokes an additional method (the pointcut method) that determines whether or not the current join point is to be adapted.

123 Such an approach is for example chosen in [RaCh03] for implementing a persistency aspect in AspectJ.

- The right join point class (corresponding to the requirements of the crosscutting concern) is provided by the underlying aspect-oriented system,
- The selection of the join point can be completely specified using the JPSL (i.e. no helper code within the join point adaptation is required to determine whether aspect-specific code needs to be executed),
- The join point selection corresponds to the rules coming from the concern's requirements, and
- The provided join point adaptation constructs correspond to the concern's requirements.

The first two points have already been discussed previously. The third point simply states that a join point selection should reflect on the underlying mental rule for selection join points. If for example, the join point selection should be based on a certain naming convention, the join point selection should represent this convention instead of simply enumerating the corresponding join point properties. The fourth point is quite straightforward: If for example for a certain join point a certain kind of adaptation construct is required (for example a structural constructive adaptation of a single object), then the aspect-oriented system should provide such kind of construct. This rule also implies that the context of the adaptation corresponds to the mental rule of the adaptation – that the elements required for the adaptation can be provided using corresponding selection criteria within the adaptation.

Consequently, the above points reflect on the strict separation of join point selection and join point adaptation whereby the actually implemented join point selection and adaptation reflects on the underlying conceptual models.

7.2 Comparing Aspect-Oriented Systems

In order to compare different aspect-oriented systems it is necessary to discuss how far each design dimension can be used to assess and compare different systems, i.e. how expressive each design dimension is to express essential distinguishing characteristics. Thereto, this section briefly discusses the means to assess aspect-oriented systems in respect to their join point models, join point selection languages, join point adaptation languages, and weavers – whereby each ingredient is described in terms of the design dimensions.

7.2.1 Comparing Join Point Models

The abstract join point model of an aspect-oriented system (section 5.3) gives a first estimation of what the system provides in order to modularize a given crosscutting concern: The kinds of join points determine where an aspect possibly contributes to the base application.

However, the abstract join point model does not give a complete view on the possible join points nor does it state anything about the means to adapt such join points. Nevertheless, in order to determine whether a certain aspect-oriented system is inappropriate to solve a given crosscutting problem the abstract join point model gives already some valuable information.

As stated in the previous section, an aspect-oriented system is considered appropriate in order to handle a given crosscutting concern, if the right kinds of join points are provided. Consequently, if it can be determined from the crosscutting concern that for example a behavioral join point needs to be adapted, the information that a system does not provide behavioral join points is already sufficient to determine that such a system is not appropriate for handling such a concern. Hence, comparing systems according to what kinds of join points they provide (expressed by the abstract join point model) is reasonable – as long as one system provides a certain kind of join point which is not provided by another system.

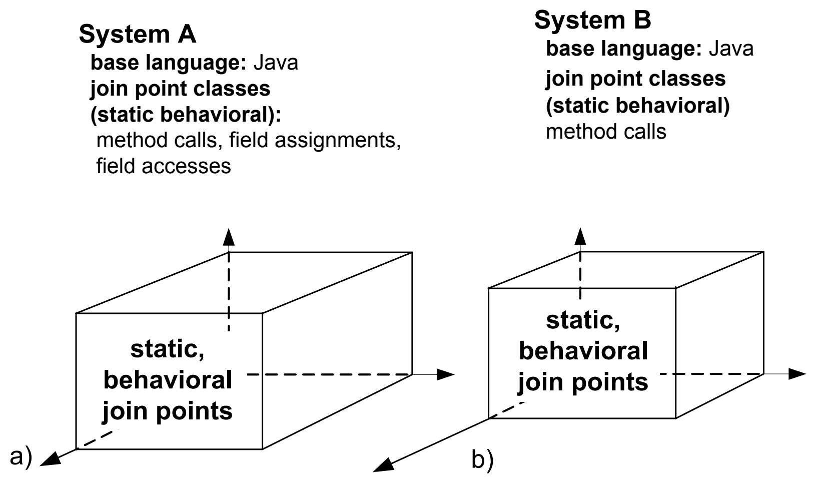


Figure 7-14. Level or Coverage for a) a system A covering b) system B.

Comparing systems with respect to the level of coverage seems to be reasonable, too. If with respect to a certain point in the 2x2 matrix constructed by the dimensions dynamicity and level of abstraction all join point classes of a system A are included in a system B (for the same base language), the phrase *higher level of coverage* is meaningful. System *A* is able to select and adapt the same elements from the base application with respect to a certain part of the join point model as system *B* plus some additional ones[124]. For example, a system B that provides only method calls as static and behavioral join points is covered by a system A that provides method calls and operators as static and behavioral join points. Consequently, system A has a higher level of coverage than system B with respect to static, behavioral join points.

Join Point Model Coverage: An aspect-oriented system A covers an aspect-oriented system B with respect to an abstract join point model element, if each join point class for the corresponding element in the concrete join point model of system B is included in the concrete join point model of A.

[124] However, as mentioned previously, this does not imply directly that the kind of adaptations available for these join points are the same in different systems.

Consequently, a system A that covers system B with respect to an abstract join point model element can be described by having a larger distance from the origin on the level of coverage axis.

Figure 7-14 illustrates the level of coverage for two different systems. Both systems are based on the same programming language Java. System A provides method calls, field assignments, and field accesses as static, behavioral join points, while system B provides only method calls as join points. Since both systems are based on the same programming language, and all join point classes in system B are included in system A, system A covers system B with respect to static, behavioral join points. Because of that, system A has a larger distance to the origin on the level of coverage axis than system B.

While the previous example used the term join point coverage to compare systems based on the same base language, it is also possible to compare aspect-oriented systems based on different base languages relying on the same paradigm. For example, all object-oriented programming languages provide constructs like messages and objects. Consequently, *message* and *object* are potentially available as join point classes in different object-oriented languages. Hence, corresponding join points classes can be compared in different languages, and the join point model of one system can be checked whether it covers the join point model of a different system based on the same paradigm.

7.2.2 Comparing Join Point Properties and Addressing

Most of the design dimensions of join point properties are rather weak for being used to compare different aspect-oriented systems. The reason for this lies in the granularity of join point properties.

Join point properties describe implementation specific elements on a finer level of granularity in comparison to abstract join point model. While for the abstract join point model a statement like “a system provides dynamic behavioral join points” is a reasonable statement, a statement such as “a system provides a static property with a direct correspondence” is less reasonable. This is because an aspect-oriented system typically provides a large set of properties for all provided join point classes. Furthermore, it is quite common to provide a number of static properties with a direct correspondence for join points. Furthermore, if the information is already available that a certain aspect-oriented system has a dynamic join point model then this already implicitly means that there are at least some dynamic properties provided for certain kinds of join points (see section 5.4.3.7 for further discussion). Consequently, a statement like “having a dynamic property” can be directly derived from the abstract join point model.

In order to determine reasonable criteria to compare aspect-oriented systems based on the design dimensions of join point properties it is either necessary to describe for all corresponding join point classes what kinds of join points they provide, or to find design dimensions of join point properties where aspect-oriented systems significantly differ. The first approach would mean that concrete information about the system needs to be considered. The second approach seems reasonable without directly referring to implementation specific details: It turns out that certain constellations of join point properties do not occur in certain systems.

The mapping in Chapter 6 reveals already properties that occurred only in certain systems. For example, the approaches AspectJ and AspectS differ with respect to the provided dynamic properties: In AspectJ all dynamic properties are meta-data properties while AspectS also provides data properties. Consequently, if it can be concluded from a concern specification that a selection criterion is to be specified on the data the distinction of data and meta-data becomes important because in such a case AspectJ can be already excluded as an appropriate aspect-oriented system.

Further essential design dimensions are the locality as well as the application's progress. In the implementations of AspectJ and AspectS there are only properties available that refer to the application's past that also are bound to the current control flow. Such a property does not exist in systems like for example Prose [PGA02], furthermore, in the future there will probably be systems where the past properties do not only refer to the control flow specific information, but for example also on data-flow specific information (see for example [MaKa03]) or any other arbitrary information from the application's past. Consequently, it seems desirable to compare past properties whether they are control flow specific or not.

7.2.3 Comparing Join Point Adaptations

With respect to join point adaptations, the design dimension level of variability represents a significant distinguishing characteristic. In fact, the missing ability to specify parametric join point adaptations has been the main argument for developing Sally (see Chapter 3, especially 3.3). Consequently, only few systems currently provide the ability to specify paramedic join point adaptations. In case it becomes clear from a concern requirements that a parametric join point adaptation is required, the absence of such a feature is already a criterion for determining a system to be inappropriate to handle the concern.

Another essential distinguishing characteristic comes from the dimensions of join point abstraction. In Chapter 6 it has been argued that neither Hyper/J, Sally, nor AspectS provide the ability to abstract over the execution context in which the adaptation is to be executed. Consequently, if it can be determined from the application's context that a user-defined join point abstraction is required, all systems that have a fix join point abstraction can be excluded from the set of possible appropriate systems for modularizing the crosscutting concern. Consequently, a statement like "system A provides a variable join point abstraction while system B does not" is a reasonable statement for comparing systems.

It can be concluded that the dimensions variability as well as join point abstraction can be used to compare aspect-oriented systems simply by stating whether a certain system provides a variable adaptation or a variable join point abstraction, or not.

However, due to the abstract nature of the design dimensions it cannot be concluded from the existence of a variable adaptation or a variable join point abstraction that a certain system is appropriate to handle a given concern. On the one hand, the adaptation constructs might be provided only for a certain kind of join point (and not the required one from the given concern, see also section 6.2.3). On the other hand, the concrete implementation might not provide the right kinds of parameters[125].

[125] For example, in AspectJ it is not possible to pass the return value of a method call to a join point adaptation.

7.2.4 Comparing Weavers

According to the argumentation in section 5.6 this thesis does not consider weaving as a conceptual element for aspect-oriented systems. The way aspects are woven should not influence the semantics of the selection and adaptation constructs. Instead, the different design dimensions simply represent different ways a weaver can be implemented in order to provide the intended semantics of join point selection and adaptation. Nevertheless, the decision how weaving is achieved must not influence the semantics of join point selection and join point adaptation.

Consequently, the design dimensions of weaving hardly play any role in order to determine whether a certain system is able to modularize a given crosscutting concern. Therefore, they and will not be considered in this chapter. Nevertheless, there might be technical reasons why developers rather prefer to have a certain kind of weaver. For example, if for a given project the runtime system to be chosen is already determined and this decision cannot be revised, a system whose weaver is based on code interpretation already can be excluded from the set of possible techniques to be used to handle a given concern. This is due to the fact that such systems require a different runtime system that the base system's original one.

Another issues in this context is the possible performance penalty of dynamic weaving approaches based on code instrumentation. As argued and measured in Chapter 4, the performance overhead of dynamic weaving might be immense[126]. Consequently, for performance reasons developers may not want to use dynamic weaving at all.

7.3 Selecting Systems for Observer Implementations

In order to illustrate that the appropriateness of aspect-oriented systems in order to modularize a given crosscutting concern can be estimated, four different occurrences of implementations of the observer design pattern [GHJV95] in object-oriented systems are being used here as exemplary crosscutting concerns.

The reason for using the observer design patterns here is mainly motivated by fact that such implementations are in general considered to be typical cases of crosscutting concerns in the aspect-oriented literature (see sections 1.2.2.1 and 4.2.2 as well as for example [GyBr03, SHU02, VeHe03]). Hence, there is not need to argue that the problem domain corresponds to the one being addressed by aspect-oriented programming in general – i.e. it is already known that the observer problematic results from crosscutting code that needs to be handled by aspect-oriented systems.

All examples are based on the same business object – a class `Student` in a class-based object-oriented system. The class `Student` provides a number of properties like `lastName`, `firstName`, `street`, `city`, `zipCode`, and `studentId`. The class provides furthermore a set of corresponding setter and getter methods (according to a naming convention used in a number of frameworks) that read and write the

126 However, a general statement of the performance overhead of dynamic weaving cannot be done without referring to a concrete implementation.

corresponding fields. `Student` objects should be observable in order to present them in a graphical user interface. According to the terminology proposed in [GHJV95], `Student` objects are *subjects* and the user interface objects are *observers*. The benefit of using the observer is that changes of student objects directly lead to changes of the corresponding user interface objects[127].

The implementations proposed in this section significantly differ in their complexity. The first implementation considers a simple class (neglecting any kind of polymorphism for implementing students) and requires notifying observers whenever a state-changing method is being invoked. The second implementation requires notifying the observers *just in time* when a state change occurs. The third implementation differs from the second one that the existence of student subclasses is assumed. The last one assumes the application of the **adapter design pattern** [GHJV95] in order to implement the business object.

Each observer concern is described first with respect to its technical characteristics. Then, each observer is described in terms of the design dimensions. Finally, the concern's design dimensions are compared with the implementations of the design dimensions according to Chapter 6 in order to estimate appropriate systems.

7.3.1 General Characteristics

Due to the general characteristics of the observer-pattern (and its typical implementations) a number of requirements of this crosscutting concern can already be determined without having a concrete subject implementation in mind. According to the implementation proposed in [GHJV95] (which also has been explained in section 1.2.2.1) it is necessary to add a field `observers` as well as the methods `attachObserver(Observer)`, `detachObserver(Observer)`, and `notifyObservers()` to the subject.

Consequently, a **structural join point** is required (representing the target), whereby it is not important whether the target join point is a class (**static join point**) or an object (**dynamic join point**). The kinds of properties necessary for describing the target or the way of how the target is to be addressed is not that important from this abstract point of view. Nevertheless, it looks like a static join point providing a class name as a property is sufficient here. Such a property could be addressed in a **direct** and **closed** way (in fact, this corresponds to the usual implementation of the observer pattern). Neither the design dimensions of variability nor the design dimensions of join point abstraction are important and can be neglected here.

Since adding new members to the target is required as the join point adaptation, a **structural, constructive join point adaptation** mechanism is required.

Another requirement comes from the observation of how the application is to be adapted. Since state changes are to be addressed in the application but the original join points still need to be executed, a **constructive, behavioral join point adaptation** is furthermore required.

[127] For reasons of simplicity the user interface classes are not discusses here. Instead it is assumed that they have a certain method `update` receiving the changes subject as a parameter.

By comparing such general requirements with the mappings of aspect-oriented systems to the design dimensions (Chapter 6), it is observable that all systems provide a corresponding join point model and corresponding join point adaptation mechanisms.

7.3.2 Interface-based, non-polymorphic subject observation

7.3.2.1 Technical Concern Description

The first kind of student subject being addressed corresponds to a straightforward implementation of a student class. The class's responsibility is to provide corresponding fields in order to store the person information. Consequently, the class directly provides the properties as being described before. Furthermore, it is assumed that `Student` does not have any subclasses.

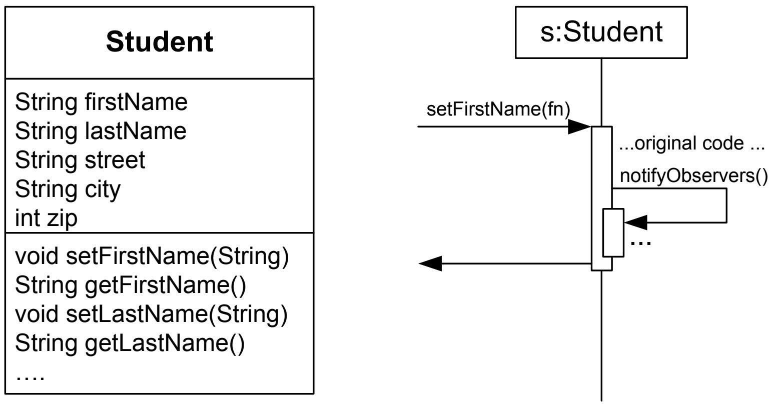


Figure 7-1. Simple `Student`-class with interface-based observation.

The notification of observers should occur whenever an instance of `Student` receives a set-message. Figure 7-1 illustrates the class for the assumed `Student` as well as an illustration of one possible interaction resulting in the notification of observers. The interaction diagram visualizes that an invocation of method `setFirstName` requires the invocation of method `notifyObservers` (in the same object).

7.3.2.2 Concern Design Dimensions

Based on the concern description it can be concluded that the corresponding aspect-oriented system requires **structural join points** (in particular method definition join points in the concrete join point model): The corresponding set-method definitions represent the join points to be adapted.

Whether instances of `Student` (dynamic join points) are being addressed or whether the corresponding source code elements (static join points) in the `Student` class are being addressed is irrelevant[128]. Hence, for the example the dynamicity of the underlying join point model is not important.

[128] This is due to the fact that `Student` does not have any subtypes nor supertypes (except the common supertype) and consequently no method definition join points are inherited to or from other classes.

The properties being addressed are the name of class `Student` itself as well as the method names within class `Student` (**static**, **local**, **atomic** join point properties with a **direct correspondence**). The addressing of the method name properties requires to be done in a **lexical** and **indirect** way since the underlying naming convention is a lexical rule for the method names.

7.3.2.3 Appropriate Systems

By considering the concrete implementations of the design dimensions as presented in Chapter 6 it turns out that all systems provide corresponding implementations. Hence, from the design dimension's point of view all systems are considered to be appropriate to modularize the interface-based, non-polymorphic observation of student subjects.

In fact, due to the simplicity of the crosscutting concern this is no surprise because a (straightforward) constructive and behavioral adaptation of method definition join points is the easiest mechanism to realize an aspect-oriented system.

Nevertheless, it should be mentioned that although all systems provide an indirect lexical addressing of join point properties, it cannot be concluded directly whether each system is appropriate. For example, it needs to be determined whether the underlying naming convention can be expressed by the corresponding language constructs.

However, by simply considering the way of how each system provides the lexical indirect addressing of method name properties it is obvious that all systems are able to express the underlying naming convention. This is due to the fact that the naming convention simply consists of prefixing method names with the characters `get` and `set` and no complex computation of method names is required.

7.3.3 State-change-based, non-polymorphic subject observation

7.3.3.1 Technical Concern Description

The second kind of student subject considers that the observer notification should occur just in time – right when the corresponding object change takes place. Hence, the notification occurs right after the assignment of values to the corresponding fields.

The main argumentation for such an approach is that additional code in the corresponding methods potentially takes too much time to execute. Typical examples for such time-consuming code are integrity rules that need to be checked (for example to check `null`-assignments, checking whether there is already a person with the same name and the same address, etc.). Consequently, in case the observers require to be notified as soon as possible, a notification strategy according the interface-based observation explained in the previous section is inappropriate since the notification of observers occurs to late.

Figure 7-2 illustrates the corresponding interaction and the required resulting behavior after the aspect is woven. In contrast to Figure 7-1, the notification of observers is not achieved at the end of the method, but somewhere in between – right after the assignment of the corresponding fields.

7.3.3.2 Concern Design Dimensions

In analogy to the previous example, the dynamicity of the underlying join point model is not important. The class `Student` does not have any subtypes nor supertypes and no method definition join points inherited from other classes. Hence, it is irrelevant whether instances of `Student` (dynamic join points) are properties being addressed, or whether the corresponding source code elements (static join points) in the `Student` class are being addressed.

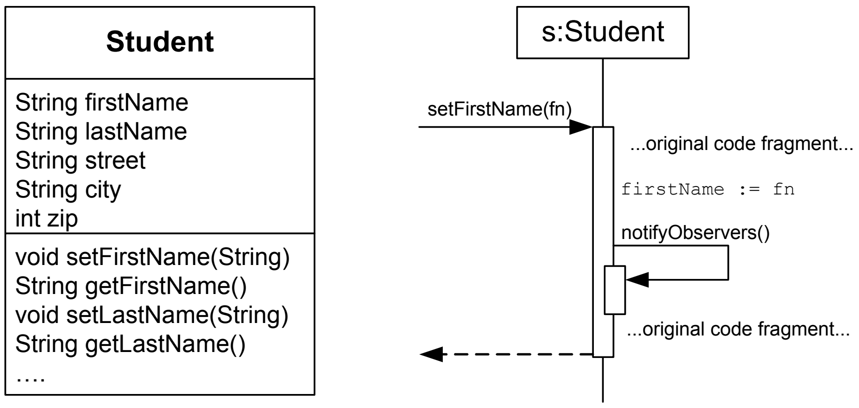


Figure 7-2. Simple `Student` class with state-change-based observation.

However, in contrast to the previous examples a different kind of join point is needed: Instead of referring to structural join points, this examples requires the adaptation of **behavioral join points** which represent the state changes (i.e. field assignments in the underlying programming language).

The properties of such behavioral join points and the corresponding property addressing correspond to the previous example: The properties being addressed are the name of the class `Student` itself as well as the names of the fields defined within the class `Student` (**static**, **local**, **atomic** join point properties with a **direct correspondence**).

7.3.3.3 Appropriate Systems

The main difference of this example to the previous example is that (instead of structural join points) behavioral join points need to be selected and adapted. By comparing the abstract join point models of the aspect-oriented systems described in Chapter 6, it can be observed that AspectS is a system that does not provide behavioral join points. Consequently, AspectS can be considered inappropriate to handle the state-change-based subject observation[129].

Investigating the abstract join point models of all other systems these are considered to be appropriate. However, reflecting on the concrete join point models of the systems it turns out that this estimation is slightly preliminary: The concrete join point model of Hyper/J includes method calls as behavioral join points only – field assignments are not part of the concrete join point model. Consequently, Hyper/J must be removed from the set of possible appropriate systems.

[129] For the same reason the role concept is inappropriate, because it also does not provide behavioral join points.

7.3.4 Interface-based, polymorphic subject observation

7.3.4.1 Technical Concern Description

The third kind of `Student` subject being addressed considers the existence of inheritance in the business object's class to be observed. Instead of having all fields and methods define in a class `Student`, there is a superclass `Person` that defines a number of fields and attributes.

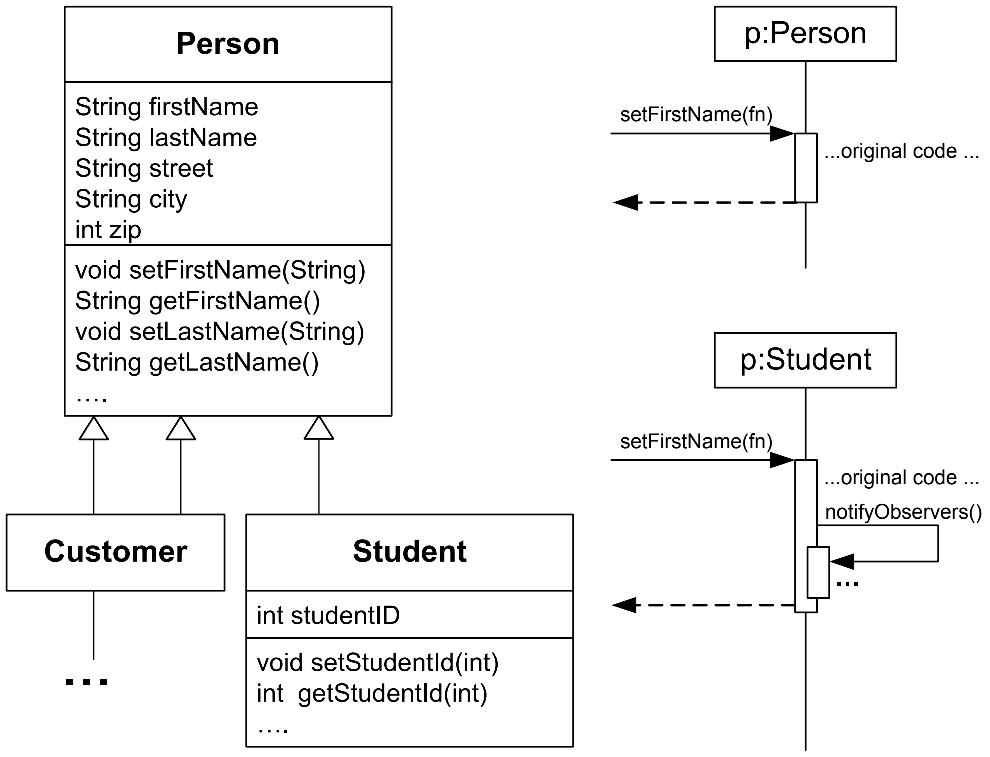


Figure 7-3. Class `Student` in inheritance hierarchy.

Figure 7-3 illustrates a corresponding class hierarchy for the class `Student`. `Student` extends class `Person` and adds the additional field `studentID`. To illustrate that the class `Person` is also used as superclass for other classes the figure shows an additional class `Customer` which is a subclass of `Person`. Furthermore, the figure illustrates two collaborations that involve objects of classes coming from the hierarchy. The first collaboration considers an object created from class `Student`. If such an object receives a state-changing set-method its observers should be notified. In case other objects which are not created from class `Student` (or any other subtype) receive such a message, this should only result in the execution of the original method, i.e. no aspect-specific code is to be executed. Of course, the same is true for objects directly instantiated from class `Person`.

7.3.4.2 Concern Design Dimensions

Based on the concern description it can be concluded that the corresponding aspect-oriented system requires **structural join points** (in particular method definition join points in the concrete join point model) - since setter methods need to be adapted.

In contrast to the previous examples, a static join point model is problematic. Static join points are appropriate for the selection of the set methods defined in class `Student` (and corresponding subclasses) since all executions of such methods represent the corresponding join points to be adapted. This is not the case in class `Person`. For example, if a set-method in class `Person` is being invoked, the corresponding object might be either directly instantiated from `Person` (i.e. it is no student), or it might be instantiated from a different (non-`Student`) subclass. In case an aspect-oriented system with a static join point model would be chosen, it is the developer's task to specify within the corresponding (behavioral, and conditional constructive) join point adaptation additional conditions that check for the runtime type of the object receiving the set-message. In case the runtime type turns out to be a `Student`, the notification of the observers has to be performed. Otherwise, the original code has to be performed.

Consequently, in such an approach the join point adaptation contains code fragments that logically belong to the join point selection – i.e. in this case the condition that checks for the actual runtime type of object receiving the message. As discussed in 7.2 this approach is not considered as being appropriate.

Based on the discussion it can be concluded that the example requires an aspect-oriented system with a **dynamic** and **structural** join point model.

Furthermore, the example requires information about runtime type of the target object receiving the message. Consequently, a **meta property** is required here.

7.3.4.3 Appropriate Systems

Since a dynamic join point model is required here, the approaches of Sally and Hyper/J are inappropriate to modularize the concern described here. The join point models of AspectJ as well as AspectS are appropriate. In respect to the required meta-property, both systems provide corresponding meta properties. Hence, AspectJ as well as AspectS are appropriate.

7.3.5 Interface-based, adapted, polymorphic subject observation

7.3.5.1 Technical Concern Description

Another variation from of the observer pattern comes from the application of the adapter pattern. Instead of providing its own fields for storing the address information, the class `Person` applies a simple (object-based) version of the **adapter pattern** [GHJV95]: A `Person` object refers to an `Address` object and adapts its interface and simply forwards all messages to it. The original `Address` class does not provide an interface matching the desired naming convention. The class `Person` provides such an interface (see Figure 7-4).

The intention of the proposed design is to reuse the existing class `Address` in the context of a `Person` object. Furthermore, this example assumes once more that the notification of observers should be realized depending on the reception of corresponding state changing methods rather than the assignments that actually lead to the state change.

The application of the adapter pattern is a known best practice in object-oriented systems and the proposed design is reasonable and straightforward. However, it has a number of significant changes with respect to the (aspect-oriented) implementations of the observer pattern and the subsequent requirements for aspect-oriented systems.

One characteristic is that the place within the inheritance hierarchy where observers attach and detach themselves differs from the positions where state changes occur that lead to the notification of observers. In the previous examples a state change of `Student` objects resulted into additional messages `notifyObservers` to the same `Student` object. The adapter-based design requires that state changes of an `Address` object results into the message `notifyObservers` being sent to a different object – the referencing `Student` object. Furthermore, if the state of an address object changes (that is possibly used by a different object not of type `Student`) no additional actions need to be done.

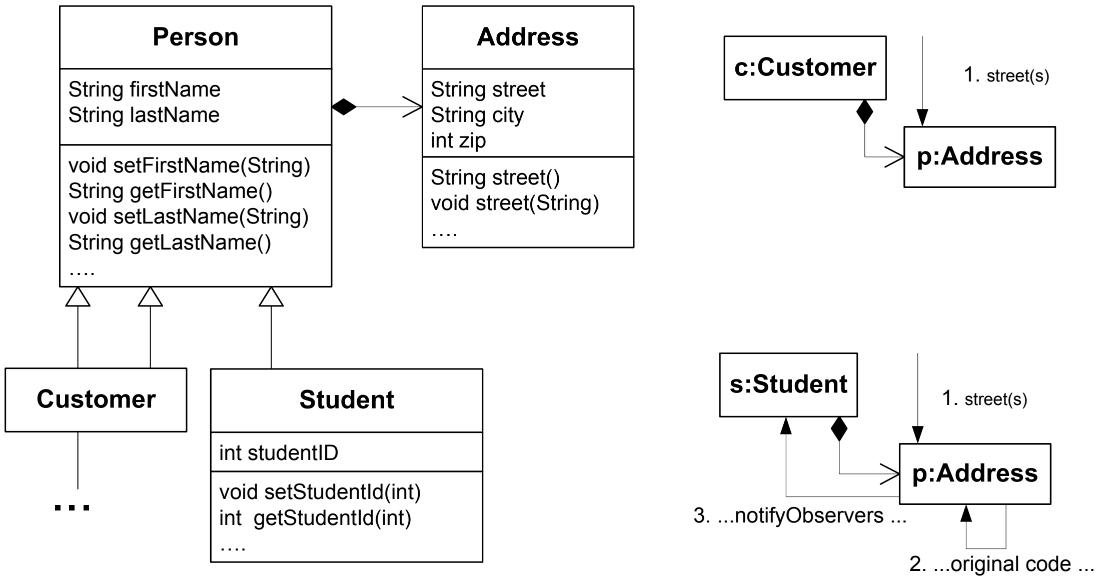


Figure 7-4. Class Student in inheritance hierarchy and adapted `Address` with interface based observation.

Figure 7-4 outlines collaborations that illustrate the desired behavior of the observer notification. In case an `Address` object referenced by a non-`Student` object (e.g. `Customer` object in Figure 7-4) receives a state changing message (e.g. the message `street`) no aspect-specific action needs to be executed because no `Student` object changes its state. In case an `Address` object referenced by a `Student` object receives the same message, `notifyObserver` needs to be sent to the corresponding student object and, of course, the original method needs to be executed.

Apart from that, the collaborations described in Figure 7-3 are still valid here: A student object receiving a state changing method (including such methods for address objects) needs to notify its observers.

7.3.5.2 Concern Design Dimensions

It can be concluded from the concern description that an appropriate aspect-oriented system requires **dynamic** and **structural join points** - state changing methods need to be adapted depending on the actual type of the receiving object as well as depending on the relationships of that object.

One characteristic of the example is (in addition to the ones for the interface-based, polymorphic observer implementation) that the triggering of observer notification depends on the actual type of the owning object, i.e. the actual type of the object owning an `Address` object. The relationship to the owning object is a **current data-property**. The owning object's type information is a **current metadata property**. This ownership information is not directly available at the corresponding join point because the relationship described in Figure 7-4 is unidirectional from class `Person` to class `Address`. Hence, the corresponding property is **non-local**.

In the example, another design dimension of join point adaptation has to be considered: The notification message has to be sent to a different object than the one receiving the state-changing method. Consequently, the context being provided by the join point is not sufficient to perform the corresponding adaptation. Instead, a **variable join point abstraction** is required whereby the corresponding variable corresponds to the target object to which the message `notifyObservers` has been sent.

7.3.5.3 Appropriate Systems

In fact, reflecting on the implementations of the design dimensions in actual aspect-oriented systems (Chapter 6), no system turns out to be appropriate. The argumentation for such inappropriateness can be done in a number of different ways.

Sally as well as Hyper/J turn out to be inappropriate because they do not provide a dynamic join point model. Hence, they are not able to specify within their selection language the runtime type of the object whose state is about to change. However, the inappropriateness can also be described by the absence of a variable join point abstraction: Neither Hyper/J nor Sally provide the ability to abstract over the join point context which is about to be adapted. Consequently, it is not possible to pass a reference of the object owning the address object to the join point adaptation.

AspectJ fulfills almost all requirements. Especially, the characteristics of a variable join point adaptation is fulfilled by AspectJ. Nevertheless, AspectJ does not provide a data property that reflects on object relationships which is required for determining the owning object of an address object. Consequently, AspectJ also is not able to express the ownership relationship via its selection constructs and is not considered to be an appropriate system. The inappropriateness of AspectJ to handle this observer implementation can also be argued from a different perspective. The only (dynamic) non-local property provided by AspectJ is the call stack. However, this observer implementation requires a different kind of (dynamic) non-local property.

AspectS does not provide (in correspondence to Sally and Hyper/J) a variable join point abstraction. Consequently, AspectS is also not able to express the adaptation in an appropriate way. Hence, in order to send the corresponding notification to the observers AspectS would require to retrieve the corresponding reference to the owning object from within its join point adaptation.

7.4 Chapter Summary and Conclusion

This chapter discussed in section 7.2 the potential to compare aspect-oriented systems based on their implementation of the aspect-oriented design dimensions. Furthermore, the chapter expressed a number of observer implementations in terms of the design dimensions. Then, this description is compared to the design dimension implementations of actual systems (section 7.3). This comparison showed that by expressing a given crosscutting concern in terms of the design dimensions it is possible to estimate the appropriateness of a given system (which is also explained in terms of the design dimensions) to handle such a concern.

The design dimensions turn out to be useful for an abstract comparison of aspect-oriented systems as well as for the selection of aspect-oriented systems for a given crosscutting problem. However, they permit only to estimate whether two systems are appropriate to handle crosscutting concern. This is due to the fact that the design dimensions abstract from a number of implementation specific details. For example, it has been noticed for the state-change-based observer implementation that from the abstract join point model it cannot be concluded that Hyper/J is not appropriate. Hyper/J provides behavioral join points, but not the *right* ones (field assignments are not provided by Hyper/J). Information from the concrete join point model was required to determine the inappropriateness of Hyper/J to handle the observer implementation. Consequently, the design dimensions are able to express exactly whether a system is *not* able to handle a given crosscutting concern.

Nevertheless, with the aid of the design dimensions it is possible to estimate the appropriateness of an aspect-oriented system without the need to know implementation specific details. This estimation already provides a first selection and reduces the effort to learn all known aspect-oriented systems: Only those systems that turn out to be appropriate from the design dimension's perspective need to be studied. Furthermore, the design dimensions determine implicitly a guide to how to proceed with the evaluation of possible aspect-oriented solutions. In the mentioned example, the different implementations of the join point model in Hyper/J needed to be studied in detail. Consequently, a crosscutting concern described in terms of the design dimensions also permits a structured approach of learning in detail whether a system is appropriate.

8

RELATED WORK

8.1 Introduction

Chapters 3 and 4 described implementations of aspect-oriented systems and systems directly related to them are discussed in the corresponding chapters. Hence, implementations of aspect-oriented systems are not considered here, in this chapter, as related work. The main reason for this decision is that the main part of this thesis – the design dimensions of aspect-oriented systems – is not focused on a specific aspect-oriented implementation technique. Consequently, mainly approaches that either classify aspect-oriented systems (or programming systems in general) or that introduce conceptual models for such systems are considered as related work for this thesis.

There are approaches that also provide a conceptual understanding of systems similar to the here proposed design dimensions – although their focus in not on aspect-oriented systems. Furthermore, there are works that directly stem from the aspect-oriented literature that are related to the aspect-oriented design dimensions.

8.2 Conceptual Descriptions of Object-Oriented Systems

For object-oriented systems there are a number of works giving a conceptual description of the key characteristics of object-oriented software development and object-oriented systems without relying on specific techniques and specific languages semantics.

Probably the most popular conceptual description of object-oriented systems is being provided in [Wegn87] where object-oriented systems are described by the (informal) equation:

object-oriented = objects + classes + inheritance[130]

[130] However, it should be noted that this view on object-oriented languages slightly changed since prototypical languages (see for example [Lieb86, Taiv96]) are nowadays also commonly accepted called object-oriented languages although they do not provide a class-construct.

In [Wegn87] terms like **object**, **class**, **inheritance**, and **delegation** are being introduced in an informal way. Likewise to this thesis the intention of [Wegn87] is to describe the design space of object-oriented systems.

Further works like for example [WeZd88] include conceptual descriptions of different kinds of inheritance mechanisms. Also, [CaWe85] gives an (informal) conceptual classification of different kinds of polymorphism[131] that distinguishes for example between **universal** and **ad-hoc polymorphism**. Such a classification provides with an abstract conceptual understanding of the underlying concepts and mechanisms to developers of object-oriented languages as well as to developers using a certain object-oriented language. [Snyd86] describes the relationship between different kinds of inheritance and encapsulation in an informal way and specifies a number of requirements for encapsulation.

The here proposed design dimensions correspond to the mentioned works above in a way that they also provide a conceptual understanding of a certain approach (in contrast to the previous ones, to the aspect-oriented approach) and deliver such a conceptual understanding by a non-formal framework. Similar to [Wegn87] (for the object-oriented approach) the intention of the design dimensions is to describe the design space for aspect-oriented systems.

Nevertheless, the focus of the design dimensions is also on the assessment of a system in respect to whether it is able to handle a certain crosscutting problem. Such a mapping of problem and solution is not the focus of previously mentioned classifications for object-oriented systems.

Also, in correspondence to the mentioned approaches above the design dimensions provide a vocabulary for developers in order to communicate different kinds of systems and different kinds of design decisions that certain systems can be based on.

8.3 Enumeration-Based Crosscutting, Lexical Crosscutting, Quantification, and Obliviousness

In the aspect-oriented literature there are already a number of terms that describe characteristics of aspect-oriented systems on an abstract level. Examples for such terms are **enumeration-based crosscutting** [GyBr03], **lexical crosscutting** [LLM99], **quantification** and **obliviousness** [FiFr00, Film01]. While the first two terms describe special characteristics of aspect-oriented systems, the latter ones try to describe the core ingredients of aspect-orientation in general.

The terms enumeration-based and lexical crosscutting have been already exhaustively discussed in section 5.4.4. In general, both terms are not defined precise enough to determine the underlying characteristics of an aspect-oriented system.

[131] [CaWe85] also gives some formal descriptions of different kinds of polymorphism based on the lambda-calculus. Nevertheless, the conceptual understanding of different kinds of polymorphism is being extracted by reasoning on different kinds of inheritance implementations.

However, with the help of the here proposed aspect-oriented design dimensions both terms can be defined more precisely.

- The term enumeration-based crosscutting as illustrated in [GyBr03] refers to the characteristic that join point properties require to be explicitly numerated. Consequently, such enumeration is a **closed** join point value addressing (see section 5.4.4.2) and a **stand-alone** property addressing (see section 5.4.4.3). Since the examples given in [GyBr03] refer furthermore to elements that are directly extracted from the application's syntax, the join point properties are extracted in a **direct way** (see section 5.4.4.1).
- The term lexical crosscutting as illustrated in [LLM99] refers to **static properties**[132](5.4.3.1) and **direct property values** (see section 5.4.4.1) that are addressed in a **lexical way** (see section 5.4.4.1).

Based on an explanation using the aspect-oriented design dimensions, the difference between both terms can be easily distinguished. Enumeration-based crosscutting emphasizes the closed value addressing, while lexical crosscutting emphasizes the lexical addressing of the underlying join point selection language.

The terms obliviousness and quantification are often used to describe the core elements of aspect-orientation (see for example [TuKr03, KHH+01, Nord01] among many others). Section 5.1.2 discussed exhaustively the inadequacies of the terms obliviousness and quantification. In general, the problem is that both terms are too generic as if they are able to express the conceptual model underlying aspect-oriented systems. From this thesis's point of view the elements join point model, join point selection, and adaptation are too essential for the conceptual understanding of aspect-oriented systems and cannot (and should not) be replaced by abstractions that are even more general. Furthermore, although both terms permit to convey a general impression of aspect-oriented software development, they are too general to differentiate the aspect-oriented approach from other approaches (see section 5.1.2 for a corresponding example). Also, the terms obliviousness and quantification try to describe the common characteristics of aspect-oriented systems but do not try to describe variabilities among them. Hence, based on these terms it is not possible to compare different systems.

In contrast to the previous terms the here proposed design dimensions represent a much finer grained conceptual model for aspect-oriented systems that gives not only a general idea of aspect-oriented systems as a whole but also a conceptual understanding of the ingredients of aspect-oriented systems.

8.4 Join Point Model by Masuhara et al.

[MKD03] identifies a number of characteristics of join point models. Such characteristics include the concept of join points and distinguishes between *lexical join points that represent locations in the program text* and *dynamic join points* that are *run-time actions,*

[132] In [LLM99] the authors only refer to elements from the code that can be extracted via a syntactical analysis. Consequently, the underlying properties are static ones. Possibly, the authors (although not explicitely mentioned [LLM99]) also refer to dynamic properties. In such a case the term lexical crosscutting does not imply any statement in respect to the underlying properties.

such as events that take place during the execution of the program. These characteristics of join points can be directly mapped to the join point models explained in section 5.3. The term lexical join point as used in [MKD03] is similar to **static join points**: Both have a direct representation in the code. The term dynamic join point as used in [MKD03] is equivalent to **dynamic** and **behavioral join points** as introduced in section 5.3.3. However, both terms do not permit to distinguish all kinds of potential join points.

1. Understanding lexical join points as pure *textual elements* is not sufficient: Systems like for example AspectS provide static join points although they are not derived from the program text but from the system's metaobjects. Furthermore, the term textual elements does not consider the difference between join points representing structural elements and those representing behavioral elements.
2. The term dynamic join point as used in [MKD03] is equivalent to dynamic and behavioral join points as introduced in section 5.3.3. The term does not reveal the ability to have dynamic join points that represent structural elements.

Dynamic and structural join points are not described by the proposed model. Furthermore, the proposed model does not explain the relationship between the elements that identify join points and the join points itself. As discussed in section 5.4.3.1, the distinction of whether or not a join point is dynamic or not is not only a matter of the join point itself, but also a matter of the distinguishing properties provided by a certain class of join point. I.e. if a system does not provide any dynamic properties, the underlying join point model is inherently static.

Furthermore, [MKD03] distinguishes between *means of identifying join points* and *means of effecting join point*. The phrase *means of identifying join points* corresponds to the concern of join point selection, but it does not reveal what kind of design decisions such a selection language and the corresponding encoding is based on.

Similarly, the phrase *means of effecting join point* identifies join point adaptation as a separate concern in the design of an aspect-oriented system but it does not reveal what different ways of effecting join points exist.

Another important difference between the here proposed characterization and the one proposed in [MKD03] is that this thesis explicitly distinguishes between the join point model and the join point adaptation. This is due to the fact, that a join point might be a structural join point, its the adaptation could be behavioral. Conceptual differences between systems that provide dynamic join point models like AspectJ and AspectS cannot be described by the conceptual model system as proposed in [MKD03]. Consequently, the model does not permit to describe the differenced among aspect-oriented systems, but only their parallels.

Finally, it can be concluded that although the model described in [MKD03] proposes a good characterization of aspect-oriented systems, it does not provide distinguishing features to compare different aspect-oriented systems.

8.5 Modeling Framework for Aspect-oriented Systems

In [MaKi03] a modeling framework for aspect-oriented systems is proposed. The framework models aspect-oriented mechanisms as a weaver that combines two input programs to produce a woven system.

A weaver is modeled as a 11-tuple where each element of the tuple represents a different view on the system: There are elements for representing the set of join points, the set of join point adaptations, the set of elements that represent distinguishing characteristics to identify join points, etc. Afterwards, based on that framework, a number of implementations are expressed in terms of the 11-tuple that represent different aspect-oriented mechanisms (like for example pointcuts and advice in AspectJ, composition strategies in Hyper/J, etc.).

The framework describes commonalities between weavers whereby the elements of the 11-tupel represent common characteristics of aspect-oriented systems, but the approach does provide distinguishing features that can be used to compare different systems. The implementation of pointcut and advice rather suggests that there are no conceptual differences between systems that provide an explicit selection of behavioral join points and behavioral join point adaptation. Hence, such differences that exist between systems cannot be described.

Furthermore, the modeling framework does not reveal what potential design decisions exist for different aspect-oriented systems. For example, it does not describe how join point properties potentially differ and what different kinds of join point selections an aspect-oriented system may provide.

An interesting observation of the proposed implementations of *open classes* in [MaKi03] (which correspond to introductions in AspectJ) is a class definition is also described as a join point. This corresponds to the mapping of class definitions to static class definition join points in AspectJ (see section 6.3).

8.6 Two-dimensional Taxonomy of Aspect-Oriented Systems

Aspect-oriented system are characterized in [RaSu03] by means of a 2-dimensional taxonomy. One dimension represents the *richness of the underlying pointcut language* and one dimension represent the *level of weaving*. The taxonomy distinguishes between *simple* and *rich* pointcut languages along one axis and *type* and *instance level weaving* in the other axis.

The approach is very similar to the here proposed design dimensions: They identify orthogonal dimensions along which aspect-oriented system can be classified. According to this, AspectJ is classified as a language with a rich pointcut language which provides instance and type level weaving. According to this, the taxonomy provides already a first foundation for the comparison of aspect-oriented systems: Systems that are mapped to different points in the 2x2 matrix differ and permit to describe other kinds of crosscutting concerns. However, the work in [RaSu03] hardly discusses what exactly such kinds of crosscutting concerns are.

Nevertheless, the taxonomy does not reveal the underlying criteria which classify a pointcut language to be rich or not. Hence, it is not possible to apply the taxonomy to a new system without applying the term richness in a rather intuitive way. Furthermore, characteristics like how properties are extracted from the underlying base program are not considered.

8.7 Join Point Designation Diagrams

Join point designation diagrams (**JPDDs**) as introduced in [SHU04] permit to specify join point selection in a visual way using the UML [OMG01]. The intention of JPDDS is to concentrate on the way of how join points are selected when modeling aspects[133].

Thereto, JPDDs extend the existing UML metamodel and provide a number of features for specifying variables and selections within UML diagrams. For example, JPDDs permit to specify variables for signatures or parameters in order to specify additional constraints on them later on. Furthermore, JPDDS introduces additional symbols like for example path expressions that abstract over message chains. Also, JPDDS permit to combine diagrams expressing join point selections and shared variables between them[134].

The applicability of JPDDS is shown in [SHU04] via expressing known join point selection criteria from the aspect-oriented literature.

In general, the here proposed design dimensions differ significantly from JPDDs: The design dimensions explain the design space of aspect-oriented systems while JPDDs are one specific aspect-oriented approach based on the base language UML.

Nevertheless, JPDDs have in common that the conceptual model as introduced in section 5.2 corresponds to JPDDs: JPDDs emphasize the separation between join point selection constructs and join point adaptation constructs[135]. In principle, it should be possible to map JPDDs directly to the design dimensions. However, this mapping is out of scope of this thesis.

8.8 Viewpoints

Viewpoints [FKN+92] permit to specify and integrate multi perspectives on a heterogeneous software system. Heterogeneous means in this context that viewpoints permit to describe the relationship between software elements coming from **representation styles** (also called **domains** in viewpoints). Examples of such

[133] In fact, the whole approach is not only restricted to join point selection but includes also means for specifying the adaptation of join points.

[134] The underlying mechanisms for sharing variables can be compared to the language approach of Sally as introduced in Chapter 3: The underlying selection language is closely related to a logical programming language and type information as provided by the UML are being used for specifying selections.

[135] In fact, the design of JPPDs was highly influenced by the work presented in this thesis.

representation styles are **Petri Nets**, functional hierarchies or action tables (see [FKN+92]).

In addition to domains, viewpoints contain a variety of elements that permit to describe different software elements. Furthermore, the viewpoint framework permits to describe relationships among different viewpoints.

In general, the affinity between the viewpoint approach and aspect-oriented software development has been notices already in the literature (cf. e.g. [GBN+03, TO00, TOHS99]). It seems as if viewpoints itself can be considered as an approach of aspect-oriented software development on a very abstract level: Instead of having a concrete programming language with a specific semantics and corresponding language constructs that select and adapt modules, viewpoints describe element across different programming languages. From this point of view, the here proposed aspect-oriented design dimensions are quite similar, because they also provide such an abstract view.

However, the here proposed design dimensions intend to describe the commonalities and differences of actual aspect-oriented systems. The focus of viewpoints is to integrate different technologies coming from different sources. Consequently, the addressed domain of the design dimensions and viewpoints is different.

Whether the resulting design dimensions also represent a reasonable conceptual model for viewpoints has not been studied throughout this thesis. However, because of the variety of underlying technologies of viewpoints, it seems rather doubtable that it is reasonable to explain viewpoints in terms of the design dimensions.

8.9 Taxonomy of Software Change

In [BMZ+04] a taxonomy of software change is proposed. Although this taxonomy does not explicitly refer to aspect-oriented systems, there are a number of parallels between the proposed taxonomy of software change and the here proposed design dimensions.

The taxonomy distinguish between a number of *themes* and *dimensions* of software change which are extracted from the focus of *when* (*temporal properties*) do changes occur, *where* (*objects of change*) do changes occur, and *how* (*change support*) changes are accomplished[136]. The here identified ingredients of aspect-oriented system are very similar. Join points can be regarded as the objects of change, and join point adaptations can be regarded as the change support. Dynamicity of weaving can be regarded as the temporal property of changes.

However, since the taxonomy proposed in [BMZ+04] has a different intention than the here proposed dimensions of aspect-oriented system design, the taxonomy does not help to distinguish between different kinds of aspect-oriented systems on a finer grained level. For example, the design dimensions of join point properties focus on a very detailed description of the system with respect to how a join point is extracted and what properties it has.

136 The taxonomy furthermore classifies *what* (*system properties*) changes occur. However, this thesis does not see any direct correspondance to the design dimensions in respect to such system properties.

8.10 Design Patterns

The work proposed in this thesis has a number of parallels to **design patterns** (cf. for example [GHJV95, BMRS96] among many others). The background of patterns comes from the work of Christopher Alexander [Ale79] which originates in the area of architecture:

- *Each pattern is a three-part rule that expresses a relation between a certain context, a problem, and a solution.*
- *As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.*
- *As an element of language, a pattern is an instruction which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.*

Design patterns are the application of the patterns idea to software construction[137]. The main intention of design patterns is to document recurring problems and their solutions in software development. There are a number of different notations for design patterns (cf. e.g. [RiZü96]), i.e., the way of how each pattern is documented. In general, design patterns describe a **context**, a **problem**, a set of **forces**, and a **solution**. Depending on the underlying notation, patterns might also describe a set of **known uses**, **consequence**, **related patterns**, etc..

The solution part is being described mostly in terms of code examples that illustrate possible applications of the pattern. For example, throughout this thesis a number of typical implementations of design patterns are being used like **observer** (section 1.2.2.1), **visitor** (section 1.2.2.2), **singleton** (section 3.3.1), **decorator** (section 3.3.3), and **adapter** (section 7.3.5) are taken from the catalog described in [GHJV95]. However, such implementations do not represent *the* solution-part of the pattern, although they occur quite often in exactly that way as being described. In fact, the solution-part of patterns is just an *illustration* of a *possible* solution: Each pattern comes with a large number of different implementations. For example, section 1.2.2.1, 1.2.2.2, and 7.3 briefly discuss some possible variations of the solution.

The patterns as described for example in [GHJV95, BMRS96, Zdun01] are not described in a formal way: Context, forces and solutions are described in an informal way. Consequently, understanding and applying a pattern is still a process that requires the interpretation of the reader: It is not possible to compute whether a design pattern's context matches the developer's context, whether the problem matches the developer's problem, nor what concrete solution is appropriate.

Although the misinterpretation is a potential source of error (or the misuse of a certain pattern), the benefit of design patterns is commonly accepted (see for example [PUPT02] for an empirical study on design patterns).

[137] In fact, a number of authors consider design patterns to be just a very special kind of applying the pattern idea to software construction (cf. e.g. [RiZü96]).

Among many advantages of design patterns described in literature the following ones occur very often (see for example [PUPT02, p. 596]):

- Design patterns encourage developers to apply best patterns.
- Design patterns improve communication, both among developers and from developers to maintainers.

The difference between the here described design dimensions of aspect-oriented systems and the design pattern approach is that the design dimensions describe the design space for aspect-oriented systems. The intention is to describe a large variety of design alternatives developers have in order to build an aspect-oriented system. Design pattern catalogs like [GHJV95] describe in contrast to this a number of typical solutions for a given problem: Their intention is not to describe the whole design space for a given problem. Furthermore, design patterns exist of different levels of abstractions. For example, the **template method design pattern** [GHJV95] is closely related to the underlying code base while other design patterns like **reflection** [BMRS96] provide a more abstract view on the problem and the corresponding solution.

However, there are a number of parallels between the design dimensions and design pattern.

First, the intention of both is the same: Providing a foundation for developer to communicate design decisions. The design dimensions provide an abstract vocabulary to communicate the kind of design decisions the aspect-oriented system is based on. In the same way, pattern catalogs or pattern languages provide an abstract vocabulary that help developer to communicate design decisions in a certain domain.

Second, the design dimensions help developer to get an understanding of the underlying aspect-oriented system they use. This corresponds to the benefit gained from documenting software using design patterns. If an aspect-oriented system is equipped with the design dimensions the system is based on (which corresponds to the mapping of the design dimensions for a number of chosen system in section 6), such information helps developers to understand the systems, since a vocabulary known from other systems can be reused in order to understand a new one.

Third, since the dimensions are very abstract views on aspect-oriented systems, each dimension provides a large number of implementations. This corresponds to the abstract idea provided by each pattern, which also results into a number of different implementations.

8.11 Aspect-Oriented Design Patterns

Aspect-oriented design patterns as introduced for examples in [HaCo03, HSU03] represent often occurring problems in the design of software systems base on aspect-oriented languages.

Some aspect-oriented patterns have already been explained and used throughout this thesis, for example **container introduction** (see sections 2.2.1 and 3.2.1) and **abstract aspect** (see section 2.2.4), although they were not introduced here in pattern form. In [HSU03] such patterns are introduced using the Alexandrian form (see [RiZü96] for a description of the Alexandrian form and alternative forms).

The aspect-oriented design patterns describe recurring problems and recurring solutions for such problem in aspect-oriented systems. Consequently, such design patterns already imply a number of parallels between such systems. However, these parallels are not explicitly written down: It is up to the developer to determine whether his system provides language constructs in order to implement such patterns in an appropriate way.

Similar to the aspect-oriented design patterns, the here proposed design dimensions represent an abstract view on aspect-oriented systems. However, the aspect-oriented patterns focus on applications written in an aspect-oriented system while the design dimensions focus on the aspect-oriented languages and systems itself: While the aspect-oriented patterns operate on the application level, the design dimensions operate on the language level.

8.12 Design Patterns for Aspect-Oriented Systems

In [Zdun04] a pattern language for the construction of aspect-oriented system is provided based on a number of patterns present in [Zdun01]. Exemplary patterns of the pattern language are **parse-tree-interpreter**, **message interceptor**, **hook injector** and **invocation context**.

The parse-tree-interpreter describes the implementation of an aspect-oriented system based on parse-tree information where the parse-tree interpreter provides full parse-tree information. Message interceptors address the problem of implementing control tasks over the message flow, such as interception, modifications of messages, or traces, in a programming language that does not support such techniques natively. Hook injector describe the addition of explicit hooks within the base application that are used to invoke aspect-specific behavior directly or indirectly. The invocation context provides dynamic execution context in order to determine the caller and the callee of a message send. The invocation context is used to determine based on dynamic information whether aspect-specific code is to be invoked.

Furthermore, [Zdun04] discusses how a number of different aspect-oriented systems can be understood in terms of the pattern language.

In general, the work proposed in [Zdun04] and the here proposed design dimensions of aspect-oriented systems are closely related since both provide an abstract view on aspect-oriented systems. Both approaches provide a vocabulary to developers in order to communicate design decisions underlying aspect-oriented systems. Furthermore, the method of extracting design elements and proving their benefit is the same in both approaches: The patterns were extracted based on an abstract reasoning on aspect-oriented systems and their appropriateness is being shown via explaining aspect-oriented systems in terms of such patterns. The method of extracting and showing the appropriateness of the design dimensions chosen throughout this thesis corresponds to this approach.

The main difference is that the pattern language is mainly focused on the implementation of the weaver, while the design dimensions mainly concentrate on the conceptual model of aspect-oriented systems. For example, [Zdun04] does not contain any description of the underlying conceptual model of aspect-oriented systems consisting of join point decomposition, join point selection, and join point adaptation, which is according to section 5.2 considered to be an essential part of aspect-oriented systems. Furthermore, the conceptual model of join points as described in 5.3 is not considered in [Zdun04].

The pattern language described in [Zdun04] can be considered mainly as a pattern language for implementing weavers. Consequently, it fits into the conceptual model of weavers as proposed in section 5.6. While the design dimensions of weavers are quite abstract and do not directly tell the developer how to implement them, the pattern languages provides concrete proposals of how each pattern can be implemented (see also [Zdun01]). For example, hook injectors are a way to implement weavers based on code transformation (see section 5.6.2). Consequently, it seems reasonable to combine the insights of the design patterns with the design dimensions of the weavers. However, this has been out of the scope of this thesis.

9

CONCLUSION

9.1 Summary

This thesis introduced so-called **design dimensions of aspect-oriented systems**. Such dimensions describe the design space of aspect-oriented systems by a number of orthogonal dimensions for the different ingredients. The identified ingredients of aspect-oriented systems are namely the join point model, the join point selection language (consisting of the encoding of join point properties and join point addressing), the join point adaptation, and the weaver.

The design dimensions are extracted by analyzing of different characteristics of aspect-oriented with respect to such ingredients, and are documented in a textual way. The applicability of the design dimensions has been shown by mapping the aspect-oriented approaches AspectJ, Hyper/J, AspectS, Sally, and Morphing Aspects to them. Furthermore, their applicability has been shown by illustrating how these dimensions help to estimate the appropriateness of a system to solve a given crosscutting problem.

Chapter 1 (*Introduction*) briefly introduces the motivation for aspect-oriented software development and illustrated the problem domain being addressed by aspect-oriented systems. Thereto the chapter introduces the necessary vocabulary like *composition*, *decomposition*, *decomposition criteria*, *separation of concerns*, *crosscutting concern*, *crosscutting concern*, *crosscutting code*, etc.

The design dimensions are motivated by a lack of a conceptual view on aspect-oriented systems. Thereto, this thesis introduced 5 different kinds of aspect-oriented systems. First, the systems **AspectJ**, **Hyper/J**, and **AspectS** (Chapter 2 - *Examples of Aspect-Oriented Systems*) were introduced. Then, the system **Sally** (Chapter 3 - *Sally – Specifying Generic Aspects*) as well as the aspect-oriented approach **Morphing Aspects** (Chapter 4 - *Morphing Aspects*) have been introduced, which both have been implemented as part of this thesis. These systems reveal a large variety of language constructs and idioms that are implemented in aspect-oriented systems. Hence, these systems represent the foundation for illustrating and arguing for the design dimensions.

Chapter 5 (*Design Dimensions of Aspect-Oriented Systems*) represented the main part of this thesis. Before introducing the design dimensions, the chapter discussed in detail the inadequacies of the known terminology in aspect-oriented software development and argues for the need of an appropriate abstract conceptual model for aspect-oriented systems. Then, the chapter characterized aspect-oriented systems as software systems having a **base language** (and a **base application**), a **join point representation** (based on a corresponding **join point model**) of the base system, constructs for specifying **join point selections**, constructs for specifying **join point adaptations**, and a **weaver** that technically realized the integration of the aspect-specific code. Based on this model, a number of design dimensions were proposed for the join point model, join point properties (representing the available data of join points), join point property addressing, join point adaptation, and weaving. A main characteristic of these design dimensions (with only few exceptions) is that they are orthogonal. Consequently, design decisions of the same features can be described in terms of different design dimensions independent of each other.

Chapter 6 (*Implementations of Design Dimensions*) applied the design dimensions in order to describe the aspect-oriented systems AspectJ, Hyper/J, AspectS, Sally, and Morphing Aspects. Furthermore, the role concept was explained in terms of the design dimensions.

Chapter 7 (*Design Dimensions-Based Comparison and Selection*) described how the design dimensions can be used in order to estimate the appropriateness of aspect-oriented systems to modularize a given crosscutting concern. Thereto, the chapter discussed first how the design dimensions can be used to compare different aspect-oriented systems (that are already described in terms of the design dimensions). Then, the chapter expressed different implementations of the observer design pattern in terms of the design dimensions. These examples were compared with the implementations of the design dimensions as explained in Chapter 6. The example showed that it is possible to express a given crosscutting concern in terms of the design dimensions and that an estimation of a set of aspect-oriented systems that are able to handle the crosscutting concern is possible.

Chapter 8 (*Related Work*) discussed work related to the here proposed design dimensions.

9.2 Future Works

The design dimensions provide a large variety of possible future works upon them. Future implementations of aspect-oriented systems may make it reasonable to think about an evolution of the design dimensions – in case it turns out to that there are different common tendencies in the future in the way how indirect join point addressing is achieved, it might be reasonable to differentiate different implementations of indirect addressing on a finer level or granularity.

The best benefit can be gained from the design dimensions if there is a large number of mappings of existing aspect-oriented systems to them. In such a case choosing an appropriate implementation in order to modularize a crosscutting concern potentially leads to a large number of different implementations so that additional technical concerns (like desired base languages, desired weaving techniques, tool support, etc.) can be taken into account. Consequently, future works include also the description of additional aspect-oriented systems in terms of the design dimensions.

In application development typically a large number of different concerns appear whereby an analysis of the concerns may reveal that some concerns should be better handled by one system while other concerns should be handled by another system. In such a situation it is desirable to know whether it is possible to apply different systems at the same time and what needs to be considered if different systems are applied at the same time. However, the design dimensions do not explain the relationship between different techniques with respect to whether they may be applied at the same time in the same application. Hence, an analysis of conflicts between different kinds of aspect-oriented systems is needed as well as a set of best practices how to handle such conflicts. Such an analysis may be done in terms of the design dimensions.

The area of aspect-oriented software development is relatively new and there is still the lack of best practices and guidelines that determine how to build high quality aspects. In the presence of such guidelines it might turn out that certain design dimensions are more important than others. Consequently, the process of selecting appropriate systems requires to be adapted in the future based on such guidelines.

The design dimensions were extracted based on the different kinds of implementations observable in current aspect-oriented systems. However, for the design of new aspect-oriented systems it seems desirable to have guidelines starting from the problem description of concrete crosscutting concerns to corresponding language features. Consequently, it seems reasonable to write down a number of design dimensions as a pattern language to guide developers from a problem to exemplary solution implementations.

9.3 Discussion and Conclusion

The main motivation of this work was the observation that there are currently no criteria available that determine the *aspect-orientedness* of systems that permit to compare aspect-oriented systems, nor to estimate the appropriateness of aspect-oriented systems with respect to their ability to modularize a given crosscutting concern. Especially for the last point it has been necessary to study known aspect-oriented systems deeply before being able to state whether they are able to handle such a given concern. The first two points result in a fuzzy understanding of the term aspect-orientation. This makes it hard to communicate the key characteristics of aspect-oriented systems as well as the benefit of aspect-orientation in general.

The proposed design dimensions and the corresponding description of aspect-oriented systems - that identified the core ingredients join point model, join point properties, property addressing, join point adaptation, and weaving - provide a reasonable conceptual understanding of aspect-oriented systems. On the one hand some design dimensions (like the design dimensions of join point models) are abstract enough to provide a significant impression of the aspect-oriented system without the need to explain any language specific element. Others (like dynamicity and level of correspondence for join point properties) are closely related to the implementation. Furthermore, the design dimensions of a certain ingredient of an aspect-oriented system reflect on different possible perspectives when providing certain features.

However, the proposed dimensions also have some drawbacks. First, there are situations where the underlying design dimensions are not clearly defined and it is up to the user of the design dimensions to interpret an unambiguous meaning. For example, section 5.4.3.4 discussed exhaustively how the locality dimension could be understood in different ways. Such potential sources of errors in the application of the design dimensions are known to this thesis. However, in the mapping of the design dimensions to known aspect-oriented systems as well as in the application of the design dimensions to a set of crosscutting concerns the proposed definitions of the terms turned out to be sufficient.

There are also examples, where the design dimensions only roughly describe the underlying design decisions. Examples for such design dimensions are the directness of join point addressing or level of value sharing. In principle, it would be possible to describe such dimensions more concretely for example by distinguishing indirect value addressing by the means of formal languages. However, for current aspect-oriented systems such a distinction is rather misleading: Either systems provide no indirect addressing for certain properties at all (like method names for merge relationships in Hyper/J), or they provide very simplified abstractions (like wildcards in AspectJ and Hyper/J), or they provide Turing-complete languages to describe possible values (according to the property addressing of AspectS). However, until now only the use of wildcards can be considered as a common direction in the aspect-oriented systems in order to provide lexically abstractions over property values, as it is provided by a large number of systems. Before considering to distinguish indirect addressing on a finer level of granularity the future development of aspect-oriented systems has to be examined.

Based on the design dimensions, it is possible to understand aspect-oriented systems without relying on implementation specific characteristics and system specific vocabulary. When learning a new aspect-oriented system it is possible to participate from the understanding of known systems because the differences of the new system can be explained in terms of the design dimensions. Furthermore, it has been shown that with the aid of the design dimensions it is possible to estimate the appropriateness of aspect-oriented systems in order to modularize a given crosscutting concern. Consequently, the design dimensions represent a valuable contribution to the research direction of aspect-oriented software development.

BIBLIOGRAPHY

[ABD+89] Atkinson, M.; Bancilhon; F.; DeWitt, D.; Dittrich, K.; Maier, D.; Zdonik, S.: *The Object-Oriented Database System Manifesto.* In Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, pages 223-40, Kyoto, Japan, December 1989, pp. 3-20.

[Aks03] Aksit, M. (ed.): Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, Boston, MA, March 17 - 21, ACM, 2003.

[Aldr04] Aldrich, J.: *Open Modules: Reconciling Extensibility and Information Hiding*, Workshop of Software Engineering Properties of Languages for Aspect Technologies (SPLAT) in conjunction with AOSD, March 18, 2004.

[Ale79] Alexander, C.: *The Timeless Way of Building.* Oxford Univ. Press, 1979.

[ÅLSM03] Åberg, R. A.; Lawall, J.; Südholt, M.; Muller, G.: *Evolving an OS Kernel using Temporal Logic and Aspect-Oriented Programming.* In 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Boston, Massachusetts. March, 2003, pp. 7-12.

[AspJ03] AspectJ Programmers Guide, http://eclipse.org/aspectj/, version 1.1.1 (November 03).

[AßLu99] Aßmann, U.; Ludwig, A.: *Aspect Weaving with Graph Rewriting.* In: Czarnecki, K.; Eisenecker, U. W. (Eds.): Generative and Component-Based Software Engineering, 1st International Symposium, GCSE'99, Erfurt, Germany, September 28-30, 1999, LNCS 1799, Springer-Verlag, 2000, pp. 24-36.

[Aßma03] Aßmann, U.: *Automatic Roundtrip Engineering*, Electronic Notes in Theoretical Computer Science, Vol. 82, No. 5, Elsevier, 2003, pp. 854-860.

[ASU86] Aho, A.; Sethi, R.; Ullman, J.: *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986.

[AWB+93] Aksit, M.; Wakita, K.; Bosch, J.; Bergmans, L.; Yonezawa, A.: *Abstracting Object-Interactions Using Composition-Filters.* In: Object-based Distributed Processing, R. Guerraoui, O. Nierstrasz and M. Riveill (Eds.), LNCS, Springer-Verlag, (1993), pp. 152-184.

[Bare84] Barendregt, H. P.: *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, 1984.

[Bato03] Batory, Don S.: *A Tutorial on Feature Oriented Programming and Product-Lines.* Proceedings of the 25th International Conference on Software Engineering (ICSE), May 3-10, 2003, Portland, Oregon, USA. IEEE Computer Society 2003, pp. 753-754.

[BCDM02] Bryant, A.; Catton, A.; De Volder, K.; Murphy, G.: *Explicit Programming*, In: [Kic02], pp. 10-18.

[BCK98] Bass, L.; Clements, P.; Kazman, R.: *Software Architecture in Practice*. Addison-Wesley, 1998.

[Beck02] Beck, K.: *The Metaphor Metaphor*, Keynote at 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, November 4-8, 2002, Seattle, Washington, USA.

[BFRJ98] Brant, J.; Foote, B.; Johnson, R. E.; Roberts, D.: *Wrappers to the Rescue*, Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP), LNCS 1445, 1998, pp. 396-417.

[BHMO04] Bockisch, C.; Haupt, M.; Mezini, M.; Ostermann, K.: *Virtual Machine Support for Dynamic Join Points*, In: [Lieb04], pp. 83-92.

[BMDV02] Brichau, J.; Mens, K.; De Volder, K.: *Building Composable Aspect-Specific Languages with Logic Metaprogramming*. In: Batory, Don S.; Consel, C.; Taha, W. (Eds.): Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings. LNCS 2487, Springer 2002, pp. 110-127.

[BMRS96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.: *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

[BMZ+04] Buckley, J.; Mens. T; Zenger, M.; Rashid, A.; Kniesel, G.: *Towards a taxonomy of software change*. To appear in: Journal on Software Maintenance and Evolution: Research and Practice, John Wiley & Sons, 2004.

[BOSW98] Bracha, G.; Odersky, M.; Stoutamire, D.; Wadler, P.: *Making the future safe for the past: Adding Genericity to the Java^{TM} Programming Language*, Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA), Vancouver, British Columbia, Canada, October 18-22, 1998, SIGPLAN Notices 33(10), 1998, pp. 183-200.

[Bruc02] Bruce, K. B.: *Foundations of Object-Oriented Languages: Types and Semantics*, MIT Press, 2002.

[BSI03] British Standards Institute (BSI): *The C++ Standard*, John Wiley and Sons, 2nd edition, 2003.

[Cann82] Cannon, H.: *Flavors: A non-hierarchical approach to object-oriented programming*, Symbolics Inc., 1982.

[CaWe85] Cardelli, L.; Wegner, P.: *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985, pp. 471-522.

[CBE+00] Constantinides, C. A.; Bader, A.; Elrad, T.; Netinant, P.; Fayad , M. E.: *Designing an aspect-oriented framework in an object-oriented environment*, ACM Computing Surveys, 2000, Article No. 41.

[CBE00] Constantinides, C. A.; Bader, A.; Elrad, T.: *A Two-Dimensional Composition Framework to Support Software Adaptability and Reuse*, Proceedings of the 6th International Conference on Software Reuse (ICSR), Vienna, Austria, 2000, pp. 388-401.

[CDHJ] Cibrán, M. A.; D'Hondt, M.; Jonckers, V.: *Aspect-Oriented Programming for Connecting Business Rules*, 6th International Conference on Business Information Systems, Colorado Springs, USA, June 2003.

[Chib95] Chiba, S.: *A Metaobject Protocol for C++*, Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October; 1995, pp. 285-299.

[Chib00] Chiba, S.: *Load-time structural reflection in Java*. Elisa Bertino (Ed.): ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings. LNCS 1850, Springer-Verlag (2000), pp. 313–336.

[CHJ03] Cibran, M.; D'Hondt, M.; Jonckers, V.: *Aspect-Oriented Programming for Connecting Business Rules*, In: Proc. of the 6th International Conference on Business Information Systems (BIS'03). Colorado Springs, USA, June 2003.

[CHMB03] Cilia, M.; Haupt, M.; Mezini, M.; Buchmann, A.: *The Convergence of AOP and Active Databases: Towards Reactive Middleware*, Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE), Erfurt, Germany, September 22-25, 2003, LNCS 2830, Springer-Verlag, 2003, pp. 169-188.

[CKFS01] Coady, Y.; Kiczales, G.; Feeley, M.; Smolyn, G.: *Using AspectC to Improve the Modularity of PathSpecific Customization in Operating System Code*, Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001, pp. 88-98.

[ClBa05] Clarke, S.; Baniassad, E.: *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison-Wesley, 2005.

[CLCM00] Clifton, C.; Leavens, G. T.; Chambers, C.; Millstein, T.: *MultiJava: modular open classes and symmetric multiple dispatch for Java*, Proceedings of the 15th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Minneapolis, Minnesota, USA, October 15-19, 2000. SIGPLAN Notices 35(10), 2000, pp. 130-145.

[CLCM00] Clifton, C.; Leavens, G.; Chambers, C.; Millstein, T.: *MultiJava, Modular open classes and symmetric multiple dispatch for Java*, In Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000. SIGPLAN Notices 35(10), October 2000, pp. 130-146.

[ClLe03] Clifton, C.; Leavens, G. T.: *Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy*, Iowa State University, Department of Computer Science,Technical Report, TR #03-15, December 2003.

[ClWa01] Clarke, S.; Walker, R.: *Composition Patterns: An Approach to Designing Reusable Aspects*, pp. 5-14.

[Codd70] Codd, E. F.: *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[Copl98] Coplien, J. O.: *Multi-Paradigm Design for C++*, Addison-Wesley, 1998.

[Cour85] Courtois, P. J.: *On time and space decomposition of complex structures*, Communications of the ACM, Volume 28, No. 6 (1985), pp. 590-603.

[CzEi00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming: Methods Tools and Applications*, Addison-Wesley, 2000.

[Dijk76] Dijkstra, E.: *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[DLS+01] Dutchyn, C.; Lu, P.; Szafron, D.; Bromling, S.; Holst, W.: *Multi-Dispatch in the Java Virtual Machine: Design and Implementation*, Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS),January 29 - February 2, 2001, San Antonio, Texas, USA, 2001, pp. 77-92.

[DV98] De Volder, K.: *Type-Oriented Logical Meta Programming*, PhD thesis, Vrije Universiteit Brussel, Belgium, 1998.

[DVDH99] De Volder, K.; D'Hondt, T.: *Aspect-Oriented Logic Meta Programming*, Pierre Cointe (Ed.): 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), Saint-Malo, France, July 19-21, 1999, LNCS 1616, Springer-Verlag, 1999, pp. 250-272.

[EAK+01] Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.: *Discussing Aspects of AOP*, Communication of the ACM, Vol. 44, No. 10, October, 2001, pp. 33-38.

[EMO04] Eichberg, M.; Mezini, M.; Ostermann, K.: *Pointcuts as Functional Queries*, 2nd Asian Symposium on Programming Languages and Systems (APLAS 2004), Taipei, Taiwan, November 4-6, 2004. Proceedings. Lecture Notes in Computer Science 3302, Springer, 2004, pp. 366-381.

[FECA04] Filman, R. E.; Elrad, T.; Clarke, S.; Aksit, M. (Eds.): *Aspect-Oriented Software Development*, Addison-Wesley, 2004.

[FiFr00] Filman, R. E.; Friedman, D. P.: *Aspect-Oriented Programming is Quantification and Obliviousness*, Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, October 2000.

[Film01] Filman, R. E.: *What is AOP: Revisited*, , Workshop on Multi-Dimensional Separation of Concerns at ECOOP, Budapest, Hungary, June 18-22, 2001.

[FKN+92] Finkelstein, A.; Kramer, J.; Nuseibeh, B.; Fiielstein, L.; Goedicke, M.: *Viewpoints: A Framework for Integrating Multiple Perspectives in System Development*, International Journal of Software Engineering and Knowledge Engineering, vol. 2(l), 1992, pp. 31-57.

[Floy79] Floyd, R. W.: *The Paradigms of Programming*, Communications of the ACM, Volume 22, No. 8 (1979), pp. 455-460.

[FoJo86] Foote, B.; Johnson, R. E.: *Reflective Facilities in Smalltalk-80*, Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings. SIGPLAN Notices 21(11), November 1986, pp. 327-335.

[Fowl99] Fowler, M.; Beck, K.; Brant, J.; Opdyke, W. F.; Roberts, D.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[GBN+03] Gray, J.; Bapty, T.; Neema, S.; Schmidt, D. C.; Gokhale, A.; Natarajan, B.: *An Approach for Supporting Aspect-Oriented Domain Modeling*, Generative Programming and Component Engineering (GPCE 2003), Springer-Verlag LNCS 2830, Erfurt, Germany, September 22-25, 2003, pp. 151-168.

[GHJV95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[GoRo89] Goldberg, A.; Robson, D.: *Smalltalk 80 – The Language*, Addison-Wesley, 1989.

[Gran98] Grand, M: *Patterns in Java: Volume. 1*, John Wiley & Sons, 1998.

[GrJo89] Graver, J. O.; Johnson, R. E.: *A type system for Smalltalk*, Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p.136-150, December 1989, San Francisco, California, United States, pp. 136-150.

[GSR96] Gottlob, G.; Schrefl, M.; Röck, B.: *Extending Object-Oriented Systems with Roles*, ACM Transactions on Information Systems, Vol. 14, No. 3, July 1996, pp. 268-296.

[GyBr03] Gybels, K.; Brichau, J.: *Arranging Language Features for More Robust Pattern-based Crosscuts*, In: [Aks03], pp. 60-69.

[HaCo03] Hanenberg, S.; Costanza, P.: *Connecting Aspects in AspectJ: Strategies vs. Patterns,* 1st Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD'02, Enschede, The Netherlands, April 23, 2002, pp. 40-45.

[HaKi02] Hannemann, J.; Kiczales, G.: *Design pattern implementation in Java and AspectJ*. Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, November 4-8, 2002, Seattle, Washington, USA. SIGPLAN Notices 37(11), ACM-Press, pp. 161-173.

[HaOs93] Harrison, W.; Ossher, H.: *Subject-Oriented Programming (A Critique of Pure Objects)*, In: Paepcke, A. (Ed.): Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, 26 September - 1 October, Washington, DC, USA, Proceedings. SIGPLAN Notices 28(10), October 1993, pp. 411-428.

[HaUn01] Hanenberg, S.; Unland, R.: *Using and Reusing Aspects in AspectJ,* Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA, Tampa, Florida, October, 2001.

[HaUn01a] Hanenberg, S.; Unland, R.: *Concerning AOP and Inheritance*, Workshop Aspekt-Orientierung der GI-Fachgruppe 2.1.9, Paderborn, May 3-4, 2001, pp. 1-6.

[HaUn02a] Hanenberg, S., Unland, R.: *A Proposal For Classifying Tangled Code*, Workshop Aspekt-Orientierung der GI-Fachgruppe 2.1.9, Bonn, Germany, February 21-22, 2002, pp. 7-12.

[HaUn02b] Hanenberg, S.; Unland, R.: *Roles and Aspects: Similarities, Differences, and Synergetic Potential*, 8th International Conference on Object-Oriented Information Systems (OOIS) LNCS 2425, Springer-Verlag, 2002, pp. 507-521.

[HaUn02c] 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.

[HaUn03a] Hanenberg, S.; Unland, R.: *Parametric Introductions*, In: [Aks03], pp. 80-89.

[HBU01] Hanenberg, S.; Bachmendo, B., Unland, U.: *An Object Model for General-Purpose Aspect Languages*, In Bosch, J. (Ed.): Generative and Component-Based Software Engineering, Third International Conference, GCSE 2001, Erfurt, Germany, September 9-13, 2001, Proceedings. Lecture Notes in Computer Science 2186, Springer, 2001, pp. 80-91.

[Herr02] Herrmann, S.: *Object Teams: Improving Modularity for Crosscutting Collaborations*, In: Aksit, M.; Mezini, M.; Unland, R. (Eds.): Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers. Lecture Notes in Computer Science 2591 Springer 2003, pp. 248-264.

[HeSc91] Heuer, A.; Scholl, M. H.: *Principles of Object-Oriented Query Languages.* In: Appelrath, H.-J. (Hrsg.), Proceedings der 4. BTW (GI Fachtagung), Nr. 270 der Reihe Informatik-Fachberichte, Springer-Verlag, 1991, pp. 178-197.

[HHU04] Hanenberg, S.; Hirschfeld, R.; Unland, R.: *Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving,* 3rd International Conference on Aspect-Oriented Software Development (AOSD), Lancaster, England, March, ACM Press, 2004, pp. 46-55.

[HHUK03] Hanenberg, S.; Hirschfeld, R.; Unland, R.; Kawamura, K.: *Aspect Weaving: Using the Base Language's Introspective Facilities to Determine Join Points*, Workshop Advancing the State-of-the-Art in Run-Time Inspection at ECOOP '03, July 21, Darmstadt, Germany, 2003, pp. 111-126.

[HiHu04] Hilsdale, E.; Hugunin, J.: *Advice Weaving in AspectJ*, In: [Lieb04], pp. 26-35.

[Hirs02] Hirschfeld, R.: *AspectS - Aspect-Oriented Programming with Squeak*. In M. Aksit, M. Mezini, R. Unland (Hrsg.): Objects, Components, Architectures, Services, and Applications for a Networked World, LNCS 2591, Springer, 2003, pp. 216-232.

[Hirs02b] Hirschfeld, R.: *AspectS: Aspects in Squeak*, Workshop on Tools for Aspect-Oriented Software Development at OOPSLA, Seattle, November 4, 2002.

[Hirs03] Hirschfeld, R.: *Advice Activation in AspectS*, Proceedings of the 2nd Workshop on Aspect-Oriented Software Development by the GI SIG 2.1.9 – Object-Oriented Software Development, Bonn, February 21-22, 2003, pp. 55-59.

[HKG+01] Hilsdale, E.; Kiczales, G.; Griswold, B., Isberg, W.; Kersten, M., Palm, J., *Tutorial: Aspect-Oriented Programming with AspectJ*, Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, September 2001.

[HMU01] Hopcroft, J. E.; Motwani, R.; Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley Publishing Company, 2001.

[HOU03] 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.

[HSU03] Hanenberg, S.; Schmidmeier, A.; Unland, R.: *AspectJ Idioms for Aspect-Oriented Software Construction,* Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany, 25th–29th June, 2003, UVK Universitätsverlag Konstanz, 2004, pp. 617-644.

[HSU05a] Hanenberg, S.; Stein, D.; Unland, R.: *Eine Taxonomie für aspektorientierte Systeme*, In: Liggesmeyer, P.; Pohl. K.; Goedicke. M. (Eds.): Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, March, Essen. Lecture Notes in Informatics 64, GI, 2005, pp. 167-178.

[HSU05b] Hanenberg, S.; Stein, D.; Unland, R.: *Roles From an Aspect-Oriented Perspectiv*e, Workshop on Views, Aspects and Roles — VAR '05 in conjunction with ECOOP, Glasgow, 25th July, 2005.

[HZU05] Hanenberg, S.; Zubairov, R.; Unland, R.: *Modularizing Security Related Concerns in Enterprise Applications – An Case Study with J2EE and AspectJ*, Accepted for publication in : Proceedings of Net-ObjectDays, Erfurt, 2005.

[IKM+97] Ingalls, D. H. H.; Kaehler, T.; Maloney, J.; Wallace, S.; Kay, A. C.: *Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself*, Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97), Atlanta, Georgia, October 5-9, 1997. SIGPLAN Notices 32(10), October 1997, pp. 318-326.

[IPW99] Igarashi, A.; Pierce, B.; Wadler, P.: *Featherweight Java: A Minimal Core Calculus for Java and GJ*. Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999, SIGPLAN Notices 34(10), October 1999. pp. 132-146.

[IwZh03] Iwamoto, M.; Zhao, J.: *Refactoring Aspect-Oriented Programs*, Workshop in Aspect-Oriented Modeling With UML, UML'2003, San Francisco, California, USA, October 20, 2003.

[JoFo88] Johnson, R. E.; Foote, B.: *Designing reusable classes*, Journal of Object-Oriented Programming, Vol. 1, No. 5 (1988), pp. 22-35.

[JSGB00] Joy, B.; Steele, G.; Gosling, J.; Bracha, G.: *Java^{TM} Language Specification*, 2nd Edition, Addison-Wesley, 2000.

[KCA04] Kniesel, G.; Costanza, P.; Austermann, M.: *JMangler – A Powerful Back-End for Aspect-Oriented Programming*, In: Filman, R.; Elrad, T.; Clarke, S.; Aksit, M. (Eds.): Aspect-Oriented Software Development, Prentice Hall, 2004.

[KDRB91] Kiczales, G.; des Rivières, J.; Bobrow, D. G.: *The Art of the Metaobject Protocol*, MIT Press, 1991.

[KeHo98] Keller, R.; Hoelzle, U.: *Binary component adaptation*. Proceedings of the 12th Conference on European on Conference Object-Oriented Programming. LNCS 1445, Springer-Verlag (1998), pp. 307-329.

[KhCo86] Khoshafian, S.; Copeland, G. P.: *Object Identity*. In: Meyrowitz, N. K. (Ed.): Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings. SIGPLAN Notices 21(11), November 1986, ACM Press, 1986, pp. 406-416.

[KHH+00] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold W. G.: *Semantics-based crosscutting in AspectJ*, Workshop on Multi-Dimensional Separation of Concerns at ICSE, June 6, 2000.

[KHH+01] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: *An Overview of AspectJ*, In: Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 2072, Springer-Verlag, 2001, pp. 327-353.

[Kic02] Kiczales, G. (ed.): *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 22-26, ACM, 2002.

[KiGu02] Kienzle, J.; Guerraoui, R.: *AOP: Does It Make Sense? The Case of Concurrency and Failures*. In: Magnusson, B. (Ed.): Proceedings of 16th European on Conference Object-Oriented Programming, Malaga, Spain, June 10-14, 2002, LNCS 2374, Springer 2002, pp. 37-61.

[KiMe05] Kiczales, G.; Mezini, M.: *Aspect-Oriented Programming and Modular Reasoning*, Proceedings of the International Conference on Software Engineering (ICSE 05). St. Louis, Missouri, May 2005.

[KLM+97] 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.

[Knie00] Kniesel, G.: *Dynamic Object-Based Inheritance With Subtyping*, PhD Thesis, University of Bonn, Germany, 2000.

[Knie96] Kniesel, G.: *Objects Don't Migrate - Perspectives on Objects with Roles*, Technical Report IAI-TR-96-11, University of Bonn, April 1996.

[KPRS00] Klaeren, H.; Pulvermueller, E.; Rashid, A., Speck, A.: *Aspect Composition Applying the Design by Contract Principle*, In: Butler, G., Jarzabek, S. (Eds.): Generative and Component-Based Software Engineering, Second International Symposium, GCSE 2000, Erfurt, Germany, October 9-12, 2000, Revised Papers. LNCS 2177, Springer, 2001, pp. 57-69.

[KRH04] Kniesel, G.; Rho, T.; Hanenberg, S.: *Evolvable Pattern Implementations Need Generic Aspects*, Workshop on Reflection, AOP and Meta-Data for Software Evolution at ECOOP, Oslo, Norway, June 15, 2004.

[Kris96] Kristensen, B. B.: *Object-Oriented Modeling with Roles*, Proceedings of the 2nd International Conference on Object-Oriented Information Systems (OOIS'95), Dublin, Ireland, 1995, Springer, 1996, pp. 57-71.

[KrØs96] Kristensen, B. B.; Østerbye, K.: *Roles: Conceptual Abstraction Theory & Practical Language Issues.* Theory and Practice of Object Systems, Vol. 2, No. 3, 1996, , pp. 143-160.

[Kuhn70] Kuhn, T.: *Structure of Scientific Revolutions*, University of Chicago Press, 1970.

[Labb03] Labbad, R.: *AspectJ in Action: Practical Aspect-Oriented Programming*, Mannning Publications, Greenwich, 2003.

[Läm03] Lämmel, R.: *Adding Superimposition To a Language Semantics*, Workshop on Foundations of Aspect-Oriented Languages (FOAL'03) at AOSD 2003, Northeastern University, Boston, Massachusetts, USA, March 17, 2003.

[Lieb04] Lieberherr, K. (ed.): *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, March 22-26, ACM-Press, 2004.

[Lieb86] Lieberman, H.: *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, In Meyrowitz, N. K. (Ed.): Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings. SIGPLAN Notices 21(11), November, 1986, pp. 214-223.

[Lieb96] Lieberherr, K.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, 1996.

[LLM99] Lieberherr, K.; Lorenz, D.; Mezini, M.: *Programming with Aspectual Components*, Technical Report, College of Computer Science, Northeastern University, March, NU-CCS-99-01, Boston, MA, 1999.

[Lope04] Lopes, C: *AOP: A Historical Perspective.* In: Filman, R.; Elrad, T.; Aksit, M.; Clarke, S. (eds.): Aspect-Oriented Software Development, Addison-Wesley, 2004.

[Maes87] Maes, P.: *Concepts and Experiments in Computational Reflection*, Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Orlando, Florida, 1987, pp. 147 - 155.

[MaKa03] Masuhara, H.; Kawauchi, K.: *Dataflow Pointcut in Aspect-Oriented Programming*, In Ohori, A. (Ed.): Proceedings 1st Asian Symposium of Programming Languages and Systems (APLAS), Beijing, China, November 27-29, 2003, Lecture Notes in Computer Science 2895, Springer, 2003, pp. 105-121.

[MaKi03] Masuhara, H.; Kiczales, G.: *Modeling Crosscutting in Aspect-Oriented Mechanisms*, In: Cardelli, L. (Ed.): Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany, July 21-25, 2003, Lecture Notes in Computer Science 2743, Springer 2003, pp. 2-28.

[MaYo93] Matsuoka, S.; Yonezawa, A.: *Analysis of inheritance anomaly in object-oriented concurrent programming languages*, in: Agha, G.; Wegner, P.; Yonezawa, A. (Ed.): *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993, pp. 107-150.

[MBL97] Myers, A.; Bank, J.; Liskov, B.: *Parameterized types for Java*, Symposium on Principles of Programming Languages, ACM, 1997, pp. 132–145.

[McBa98] McDowell, C. E.; Baldwin, E. A.; *Unloading Java Classes That Contain Static Fields*, ACM SIGPLAN Notices, Volume 33, No. 1, January 1998, pp. 56 - 60.

[McHs03] McDirmid, S.; Hsieh, W. C.: *Aspect Oriented Programming with Jiazzi*, In [Aks03], pp. 80-89.

[MeOs03] Mezini, M.; Ostermann, K.: *Conquering aspects with Caesar*. In: [Aks03], pp. 90-99.

[Meye98] Meyer, B.: *Object-Oriented Software Construction*, Prentice Hall, Upper Saddle River, 2nd edition, 1997.

[MIT01] MIT Technology Review, *Ten emerging technologies that will change the world*, Januar/Februar, 2001.

[MKD03] Masuhara, H.; Kiczales, G.; Dutchyn, C.: *A Compilation and Optimization Model for Aspect-Oriented Programs*, Proceedings of Compiler Construction (CC2003), LNCS 2622, Springer-Verlag, 2003, pp.46-60.

[MSC02] Microsoft Corporation, *Microsoft Visual C++ .NET Language Reference*, Microsoft Press, 2002.

[NCT04] Nishizawa, M.; Chiba, S.; Tatsubori, M.: *Remote Pointcut - A Language Construct for Distributed AOP*, In [Lieb04], pp. 7 – 15.

[NeZd99] Neumann, G.; Zdun, U.: *Enhancing object-based system composition through per-object mixins*. In Proceedings of Asia-Pacific Software Engineering Conference (APSEC), Takamatsu, Japan, December 1999.

[Nord01] Nordberg, M. E. III: *Aspect-Oriented Dependency Inversion*, Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA, Tampa, October, 2001.

[OKK+96] Ossher, H.; Kaplan, M.; Katz, A.; Harrison, W.; Kruskal, V.: *Specifying Subject-Oriented Composition*, TAPOS - Theory and Practice of Object Systems, volume 2, number 3, 1996, Wiley & Sons, pp. 179-202.

[OMG01] Object Management Group (OMG), *Unified Modeling Language (UML) Specification*, Version 1.5, 2003 (OMG Document formal/03-03-01).

[OsTa01] Ossher, H.; Tarr, P.: *Using multidimensional separation of concerns to (re)shape evolving software*. Communication of the ACM, 44 (10), 2001, pp. 43-50.

[Oster94] Ousterhout, J.: *Tcl and the Tk Toolkit*, Addison Wesley, 1994.

[PaJa98] Palsberg, J.; Jay, C.B.: *The Essence of the Visitor Pattern*, 22nd International Computer Software and Applications Conference, August 19-21, 1998, Vienna, Austria. IEEE Computer Society 1998, pp. 9-15.

[Parn72] Parnas, D. L.: *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15, No. 8 (1972), pp. 1053-1058.

[Pern90] Pernici, B.: *Objects with Roles*, in: F.H. Lochovsky, R.B. Allen (Eds.): Proceedings of the Conference on Office Information Systems, SIGOIS Bulletin, vol. 11, no. 2/3, ACM Press, New York, 1990, pp. 205-215.

[PFFT02] Pinto, M.; Fuentes, L.; Fayad, M. E. ;Troya, J.M.: *Separation of Coordination in a Dynamic Aspect Oriented Framework*, Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, April 22-26, 2002, pp. 134 - 140.

[PGA02] Popovici, A.; Gross, T.; Alonso, G.: *Dynamic Weaving for Aspect-Oriented Programming*, In: [Kic02], pp. 141 - 147.

[PGA03] Popovici, A.; Gross, T.; Alonso, G.: *Just in Time Aspects: Efficient Dynamic Weaving for Java,* Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, MA, March 17 - 21, 2003, pp. 100 - 109.

[Pier02] Pierce, B.: *Types and Programming Languages*, MIT Press, 2002.

[POM03] Pichler, R.; Ostermann, K.; Mezini, M.: *On Aspectualizing Component Models.* In Software Practice and Experience, Volume 33, Issue 10, pp. 957-974, Wiley Publishers, 2003, pp. 957 - 974.

[Pras03] Prasad, M. D: *Typecasting As a New Join Point in AspectJ*, 13th Workshop for PhD Students in Object-Oriented Systems at 17th European Conference on Object-Oriented Programming (ECOOP).

[Pree95] Pree, W.: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[PSDF01] Pawlack, R.; Seinturier, L.; Duchien, L. Florin, G.: *JAC: A Flexible Solution for Aspect-Oriented Programming in Java*, Proceedings of Reflection 2001, Kyoto, Japan, September 25-28, 2001, LNCS 2192, Springer 2001, pp. 1-24.

[PUPT02] Prechelt, L.; Unger-Lamprecht, B.; Philippsen, M.; Tichy, W. F.: *Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance.* IEEE Trans. Software Eng. 28(6), 2002, pp. 595-606.

[RaCa03] Rayside, D., Campbell, G. T.: *An Aristotelian understanding of object-oriented programming.* Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000. SIGPLAN Notices 35(10), pp. 337-353.

[RaCh03] Rashid, A.; Chitchyan, R.: *Persistence as an aspect*, In: [Aks03], pp. 120-129.

[Rash02] Rashid, A.: *Weaving Aspects in a Persistent Environment.* SIGPLAN Notices, Volume 37, No. 2, February 2002, pp. 36-44.

[RaSu03] Rajan, H.; Sullivan, K.: *Need for Instance Level Aspect Language with Rich Pointcut Language*, Proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT) held in conjunction with AOSD 2003, Boston, MA, USA, March 18, 2003.

[Riel96] Riel, A. J.: *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

[Riv96] Rivard, F.: *Smalltalk: a reflective language*. In Proceedings of Reflection '96, San Francisco, California, April 21-23, 1996.

[RiZü96] Riehle, D.; Züllighoven, H.: *Understanding and using patterns in software development*. Theory and Practice of Object Systems, 2(1), 1996, pp. 3-13.

[SCT03] Sato, Y; Chiba, S.; Tatsubori, M.: *A Selective Just-in-Time Aspect Weaver*, Proceeding of the Second International Conference on Generative Programming and Component Engineering (GPCE), Erfurt, Germany, September 2003, pp. 189-208.

[SeMo04] Sereni, D.; de Moor, O.: *Static analysis of aspects*, In. [Aks03], pp. 30-39.

[SGC02] Sullivan, K.; Gu, L.; Cai, Y.: *Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ*, In: Kiczales, G. (ed.): Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, April 22-26, ACM-Press, 2002, pp. 19 - 26.

[SHU02] Stein, D.; Hanenberg, S.; Unland, R.: *A UML-based aspect-oriented design notation for AspectJ*, In: [Kic02], pp. 106 - 112.

[SHU04] Stein, D.; Hanenberg, S.; Unland, R.: *Query Models*, In: Baar, Th.; Strohmeier, A.; Moreira, A.; Mellor, St.: Proc. of the 7th International Conference on the Unified Modeling Language (UML 2004), Lisbon, Portugal, October 11-15, 2004, LNCS 3273, Springer-Verlag, 2004, pp. 98-112.

[SHU05] Stein, D.; Hanenberg, S.; Unland, R.: *Visualizing Join Point Selections Using Interaction-Based vs. State-Based Notations*, Appears in: Desel, J.; Frank, U.: Proc. of Workshop on Enterprise Modelling and Information Systems Architectures (EMISA 2005, in conjunction with ER 2005), Klagenfurt, Austria, October, 24-25, 2005, LNI.

[SLL03] Skotiniotis, T.; Lieberherr, K.; Lorenz, D. H.: *Aspect Instances and their Interactions*, Workshop on Software-engineering Properties of Languages for Aspect Technologies held in conjunction with AOSD 2003, Boston, MA, USA, March 18, 2003.

[SmSm77] Smith, J. M.; Smith, D. C. P.: *Database Abstractions: Aggregation and Generalization*, ACM Transactions on Database Systems, Vol. 2, No. 2 (1977), pp. 105-133.

[Snyd86] Snyder, A.: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. Norman K. Meyrowitz (Ed.): Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings. SIGPLAN Notices 21(11), 1986, pp. 38-45.

[SpGS02] Spinczyk, O.; Gal, A.; Schröder-Preikschat, W.: *AspectC++: An Aspect-Oriented Extension to C++*, Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002) , Sydney, Australia, February 18-21, 2002.

[Stee90] Steele, G: *Common Lisp: the Language*, 2nd Edition, Digital Press. 1990.

[StHa05] Störzer, M.; Hanenberg, S.: *A Classification of Pointcut Language Constructs*, Workshop on Software-Engineering Properties of Languages and Aspect

Technologies (SPLAT) in conjunction with 4th International Conference on Aspect-Oriented Software Development (AOSD), Chicago, USA, March 14-18, 2005.

[Stör03] Störzer, M.: *Analytical problems and AspectJ*. In: Bachmendo, B.; Hanenberg, S.; Herrmann, S.; Kniesel, G.: Proceedings of 3rd Workshop on Aspect-Oriented Software Development of the German Informatics Society, Essen, March, 2003, pp. 39-44.

[StSh94] Sterling, L.; Shapiro, E.: *The Art of Prolog: Advanced Programming Techniques*, MIT Press, 2nd edition, 1994.

[StSu78] Steele, G.; Sussman, G. J.: *The Art of the Interpreter of, the Modularity Complex (Parts Zero, One, and Two)*. MIT AI Lab. AI Lab Memo AIM-453. May 1978.

[Sun04a] Sun Microsystems, *JavaBeans Homepage*, http://java.sun.com/products/javabeans/, last access: June 2004.

[Sun04b] Sun Microsystems, *Enterprise JavaBeans Homepage*, http://java.sun.com/products/ejb/, last access: June 2004.

[Sun04c] Sun Microsystems, *JDK™ 5.0 Documentation* , http://java.sun.com/j2se/1.5.0/docs/, last access: March 2004.

[Taiv96] Taivalsaari, A: *On the Notion of Inheritance*. In: ACM Computing Surveys, Vol. 28, No. 3, 1996, pp. 439-479.

[TCIK00] Tatsubori, M.; Chiba, S.; Itano, K.; Killijian, M.: *OpenJava: A Classbased Macro System for Java*, Reflection and Software Engineering, LNCS 1826, Springer-Verlag, 2000, pp.117-133.

[TO00] Tarr, P.; Ossher, H.: *Hyper/J™ User and Installation Manual*, IBM Corporation, 2000.

[TOHS99] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S. M.: *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, In 21st International Conference on Software Engineering (ICSE), 1999, pp. 107–119.

[TuKr03] Tucker, D. B.; Krishnamurthi, S.: *Pointcuts and Advice in Higher-Order Languages*, In: [Aks03], pp. 158-167.

[USE04] Workshop Series on Unanticipated Software Evolution, http://www.cs.uni-bonn.de/~gk/use/, last access: March 2004.

[VeHe03] Veit, M.; Herrmann, S.: *Model-View-Controller and Object Teams: a Perfect Match of Paradigms*, In: [Aks03], pp. 140-149.

[WaVi04] Walker, R.; Viggers, K.: *Implementing protocols via declarative event patterns*. In: Taylor, R. N., Dwyer, M. B. (Eds.): Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004. ACM 2004, pp. 159-169.

[Wegn87] Wegner, P.: *Dimensions of object-based language design*. In: Meyrowitz, N. K. (Ed.): Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), October 4-8, 1987, Orlando, Florida, SIGPLAN Notices 22(12), 1987, pp. 168-182.

[WeZd88] Wegner, P.; Zdonik, S. B.: *Inheritance as an Incremental Modification Mechanism or What Like is and Isn't like*, In: Gjessing, S., K. Nygaard (Eds.): Proceedings of European Conference on Object-Oriented Programming (ECOOP), Oslo, Norway, August 15-17, 1988, LNCS 322, Springer-Verlag, 1988, pp. 55-77.

[WKD02] Wand, M.; Kiczales, G.; Dutchyn, C.: *A Semantics for Advice and Dynamic Join Points in AspectOriented Programming*, Workshop on Foundations Of Aspect-Oriented Languages (FOAL) at AOSD'02, Enschede, The Netherlands, April 22, 2002.

[Wuyt01] Wuyts, R.: *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*, PhD thesis, Vrije Universiteit Brussel, Belgium, 2001.

[Zdun01] Zdun, U.: *Language Support for Dynamic and Evolving Software Architectures*, PhD thesis, Department of Mathematics and Computer Science, University of Essen, Germany, November 2001.

[Zdun04] Zdun, U.: *Pattern language for the design of aspect languages and aspect composition frameworks*. IEE Proceedings Software, 151 (2), April 2004, pp. 67-83.