Utilizing Design Information
in Aspect-Oriented Programming

István Nagy, Lodewijk Bergmans, Wilke Havinga & Mehmet Aksit
TRESE group, Dept. of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
+31-53-489{5682, 4271}
{nagist, bergmans, havingaw, aksit}@cs.utwente.nl

Abstract: In current aspect-oriented languages, pointcut definitions select join points of a program based on lexical information such as explicit names of program elements. However, this reduces the adaptability of software, since it involves too much information that is hard-coded, and often implementation-specific. We claim that this problem can be reduced by referring to program elements through their semantic properties. A semantic property describes for example the behavior of a program element or its intended meaning. In this paper, we analyze four techniques that are regularly used in state-of-the-art object-oriented languages, such as Java or C#, in associating semantic properties with program elements. Also, the usage of semantic properties in the weaving process of aspect-oriented languages is illustrated and their deficiencies are outlined. Accordingly, we formulate requirements for the proper application of semantic properties in aspect-oriented programming. We discuss how to use semantic properties for the superimposition of aspects, and how to apply superimposition to bind semantic properties to program elements. To achieve this, we propose language abstractions that support semantic composition: the ability to compose aspects with the elements of the base program that satisfy certain semantic properties. Based on this proposal, we show how Compose*, a concrete aspect-oriented language, can be extended to support semantic properties. We demonstrate how to apply semantic properties to improve the reusability of aspects. The paper ends with related works, a discussion, and conclusions.

1 Introduction & Motivation

The process of software development generally consists of refinement of conceptual knowledge towards an executable program. During this process, "ideas" or design artifacts are mapped onto implementation artifacts. Typically, the actual implementation contains the artifacts that are necessary for execution. Consequently, certain conceptual knowledge that expresses the intentions of a design is not explicitly represented in the final program. In this paper, we analyze the impact of this information-loss with respect to the pointcut expression.

In aspect-oriented programs (AOPs), a pointcut designator expression specifies a composition interface where the behaviour of a (sub)program can be modified or enhanced, by composing a program that expresses aspectual (crosscutting) behaviour.
Although design information\(^1\) is not necessary for correct execution, it is generally required to avoid fragility of pointcuts with respect to changes in the implementation. The lack of explicit design information in the implementation forces programmers to express the design information in other ways, for example based on syntactic conventions. This paper argues that specifying pointcuts by designating the syntactic proper-ties of artifacts (and perhaps the state) of the program only, can be too restrictive for evolving programs. For this purpose we present a linguistic mechanism that can be used to express design properties in the pointcut specifications. We also evaluate this mechanism with respect to the recent proposals along this direction.

This paper is structured as follows: section 2 presents an analysis of the various ways programmers use to encode design information in a program. Section 3 shows a proposal for attaching design information to the desired places in programs and for referring to this design information in pointcut specifications. In section 4 we present an implementation of the proposed language construct in the aspect-oriented language *Compose*. Section 5 illustrates the applicability of the proposed construct using a number of examples. Section 6 discusses the related work. Finally, section 7 concludes the paper.

## 2 Problem Analysis

Programmers have come to use several means of expressing design intentions in the form of (semantic) information attached to certain program elements. In this section, we present and analyze four commonly used techniques for representing design information in the state-of-the-art object-oriented languages, such as Java or C#. We also illustrate how AOP languages (may) utilize the design information in the pointcut expressions. In the analysis, we show that some of the techniques result in fragile programs especially with respect to evolving requirements. The analysis concludes with a set of requirements for using design information in aspect-oriented languages.

### 2.1 Naming pattern

**Technique:** A common method for expressing design information is to use certain naming conventions or *stylistic naming patterns* [NV02] in the identifiers of a program. A typical example in Java is illustrated by the following code:

```java
public class Customer {
    private String firstName;
    private String lastName;
    private String email;
    public String getFirstName(){ return firstName; }
    public void setFirstName(String fname) { firstName=fname; }
    public String getEmail() { return email; }
    public void setEmail(String nemail) { email=nemail; }
    ...
}
```

*Figure 1: Examples of stylistic naming pattern*

\(^1\)Also called semantic properties.

40
This example shows a very simple convention in Java: a method that queries a given instance variable starts with the `get` prefix, while the updater method has the `set` prefix. There are other well-known naming patterns, such as the `add` and `remove` prefixes for maintaining the items of a collection, or the `test` prefix used by the test fixtures of JUnit [JUnit].

Programmers may use these patterns for the sake of more organized, comprehensible code but there are frameworks, e.g. JavaBeans [Ham97] or JUnit, which in fact rely on these naming patterns for proper operation. In the latter case naming patterns are not only for expressing design information but they also represent explicit dependencies (i.e. they act as ‘hooks’) for the frameworks. More discussion about naming patterns can be found in [NV02].

**Possible use:** The following example shows how naming patterns can be used in combination with wildcards in a pointcut designator expression:

```java
pointcut queryMethods();
    within (Customer) && execution (public * get*(.)));

pointcut updateMethods();
    within (Customer) && execution (public void set*(..));
```

Figure 2: Example of combining pointcuts and naming patterns

The example in Figure 2 shows two pointcuts. The first one designates the execution of each method that starts with the prefix `get` within the `Customer` class. The second pointcut does the same thing with the `set` prefix. The intention of the first pointcut is to designate the execution of methods that query the state of a `Customer` instance, while the second one designates the execution of ‘update’ methods. Note that both pointcuts rely on the naming patterns by using the prefixes.

**Discussion:** In this example, certain properties are hard-wired into the signatures of the base code and the weaving is done based on these signatures. However, programmers need to keep in mind the coding conventions: (a) using the `set` prefix to denote the behavior of the method for all `setter` methods, and (b) avoiding incidental naming ambiguities, such as `settle()` and `settings()` in this case. This phenomenon has been also identified as the arranged pattern problem by Gybels et. al. in [GB03].

Here, the problem stems from the fact that the design information is not separated, but encoded in the structure—more precisely, in the identifiers—of the program. We claim that instead of encoding in the program, design information should be distinctly connected to program units via dedicated language constructs. We will refer to this requirement as separability of properties.

The technique of naming patterns has another deficiency: it is possible that a unit has to participate in more than one pattern. As an example, consider the member variable that stores the email address in Figure 1. Assume that two frameworks, independently from each other, have to be coupled with this variable. The first framework deals with persistence in the whole system, that is, it stores the email address in a database. The second framework is used for encrypting textual data; in this case, it encrypts the email

---

We use AspectJ notation because of its wide use in practice.
address. To express these dependencies we might use a naming pattern with an identifier such as:

```java
private String persistent_encrypted_email;
```

However, this syntactic solution has several problems. The more patterns are applied, the more juxtaposition of textual identifiers is required in the signatures. In addition, the programmers of a framework need to be aware of the fact that more properties may appear in a signature, not only those properties that are specific to their own framework.

As the example shows, it is an important capability to handle not only single but also multiple design information attached to the same program piece. We will refer to this requirement as multiple properties.

### 2.2 Structural Patterns

**Technique:** When using structural patterns, the software engineer adapts the structure of the program, without affecting its behavior, for the sole purpose of attaching design information. For example, a frequently applied technique in Java is to use marker interfaces [NV02]. A marker interface is an interface declaration that does not contain any signature declarations. Consider the following example:

```java
public interface PersistentRoot {}
public class Customer implements PersistentRoot {
}
```

In this example, an (empty) marker interface PersistentRoot is declared, the Customer class then implements this interface. This does not change the behavior of the Customer class, but only "marks" the class as being a "PersistentRoot". Typical examples of marker interfaces in Java are the java.io.Serializable, java.lang.Clona\(b\)le and java.util.Event\(\text{a}\)\(\text{b}\)\(\text{e}\)\(\text{n}\) interfaces.

**Possible use:** Marker interfaces in AOP languages can be designated using this or target pointcuts:

```java
pointcut queryMethods();
    this(PersistentRoot) && execution (public * get());
```

**Discussion:** Other examples of structural patterns that can be used to attach a certain meaning to an element of a program, are dummy methods (i.e. that are not intended to be called), dummy variables and dummy arguments with specific types to indicate a meaning. Such patterns can be used by pointcut designators to identify certain joinpoints within a program.

A difference between structural patterns and naming patterns is that in the latter, the semantic properties are hard-wired purely into the identifiers. This makes them very difficult to maintain, e.g. adding multiple properties is problematic. Although this does not necessarily apply to all structural patterns, for example a class can implement more than one interface, so it is possible to attach multiple properties via marker interfaces. One problem with marker interfaces is that they can be applied only to classes.

A general problem of both structural and naming patterns is that they statically attach design information to classes. This implies that the information will be applied in every application that (re-)uses these classes, whereas this might not be desirable: for example,
PersistentRoot is a property that tends to be specific to an application, not to the characteristics of the class. (That is, a class is not necessarily persistent in every application.) It is also possible that a given property can be used by different frameworks by coincidence, and they interpret the property in different ways. To solve this problem, we think that design information should be dynamically attachable to units and be configurable according to the needs of different applications. We will refer to this requirement as *late binding*.

2.3 Annotations

**Technique:** The .NET platform (supporting various languages) has annotations (*custom attributes*) [ECMA02] to bind design information to a range of language constructs. The metadata facility of Java 1.5 [Sun02] realizes the same technique. Annotations are defined as first class entities, they can have arguments and various constraints can be applied on them. The following example shows a definition of an annotation in Java:

```java
@Target(TYPE);
public @interface PersistentRoot{
    public String tableName() default "unassigned";
}

@Target is a meta-annotation that constrains the type of the unit to which the newly defined annotation can be attached. The value of the argument is TYPE, which means that PersistentRoot can be attached only to classes and interfaces. The definition of PersistentRoot has one String argument called tableName. Since a default value is provided, it is not necessary to fill in the argument when the annotation is used.

```java
@PersistentRoot("CUSTOMERS")
public class Customer {
    @Persistent()
    String firstName;
    ...
    @Persistent()
    String email;

    @Query()
    public String getFirstName(){ return firstName; }

    @Update()
    public void setFirstName(String fname) { firstName=fname; }
    ...
    @Update()
    public void setEmail(String nemail) { email=nemail; }
}
```

**Figure 3:** Custom attributes as design information

Figure 3 illustrates how annotations can be applied to attach design information to class Customer, which was presented in Figure 1. The annotation `@PersistentRoot` (1) is attached to class Customer to indicate that the instances of this class should be persistent. The annotation `@Persistent` (2) denotes that the field `email` of class Customer should be stored as a persistent variable. The annotation `@Query` (3) is attached to those methods that do not change the state of an instance of class Customer, while the annotation `@Update` (4) is attached to those methods that cause state change.
Possible use: For AspectJ, the idea of defining pointcuts based on annotations is not new; it has already been mentioned by G. Kiczales in [Alm03]. However, among the current AOP technologies only JBoss [JBoss], AspectWerkz [AspWerkz] and AspectJ [AT04] support annotations as reference points for designating join points. For example, the execution of the methods that change the state of Customer could be designated by the following pointcut in JBoss:

```xml
<bind pointcut="execution(Customer>Update(..))">
    <interceptor class="" />
</bind>
```

Discussion: Besides the absence of pointcut designators, there are other problems as well; like naming patterns or marker interfaces, annotations are also statically bound to the units that they are attached\(^2\) to.

The second problem with annotations is that they are usually scattered. In other words, it is possible that an annotation is attached to multiple units over the whole application. For example, it is quite common that the annotation `@Author("X.Y.")` is attached to every class within one or more packages. Meta-annotations (i.e. annotations attached to the definition of other annotations), such as the annotation `@Retention`, are often scattered too. We will refer to this requirement as support for scattered properties.

Another related problem is that annotations can be relatively bound to each other. That is, an annotation can be attached to a certain unit, only when a corresponding unit has a certain annotation. To illustrate this, we refer to the annotations attached to class Customer in Figure 3. As a first example, for the member variables of a persistent class we could define the following rule: “if a class has the annotation `@PersistentRoot` for each of its public member variables that has no annotation `@Transient`, it has to be marked with the annotation `@Persistent`”. For the methods of a persistent class we could define similar rules, for instance: “if a class has the annotation `@PersistentRoot` for each of its public methods without annotation, it has to be marked with the annotation `@Query by default`”, or “if a class has the annotation `@PersistentRoot` for each of its constructors, it has to be marked with the annotation `@Create by default`”. Practically, the attachment of a given annotation may trigger the attachment of every other annotation that it depends on.

2.3.1 Superimposing annotations vs. using clear pointcuts

One might ask why it is necessary to attach the annotations `@Persistent` or `@Create` if their places can be designated by the pointcuts that could express the rules above. Using such pointcuts is in fact satisfactory if these annotations have to be known only by the weaver that reads the pointcut that designates the places of the annotations. However, annotations might be read not only by the weaver but by arbitrary non-aspect-oriented frameworks, as well. Thus, every non-AOP framework or component must be able to read such a pointcut to determine if an annotation is attached to a given unit or not. In

---

\(^2\) Recently, newer AspectJ versions have been released (AspectJ 5 milestone 1 and 2 [AT04]), in which annotations are also supported.

\(^3\) If the retention policy of an annotation is SOURCE in Java it is discarded by the compiler and not recorded in the bytecode.
addition, the current AOP tools are not expressive enough to formulate the rules shown in the previous subsection in the form of pointcuts.

2.4 Automatically-derived semantic properties

Technique: As the field of aspect-oriented programming evolves, the need for more ‘expressive’ pointcuts becomes apparent. This is illustrated by G. Kiczales’ keynote in [Kic03], where he argues that the ways to express pointcuts should be as close as possible to the intention of the designer. This naturally leads to proposals for expressing pointcuts that do not directly refer to elements within the program (source), but that refer to those points in the program or the execution of the program that fulfill a certain property. These join points can only be determined by reasoning about the semantics of the program and the adopted programming language.

Possible use: A well-known example is the primitive pointcut in AspectJ named `cflow()`. It selects all the points in the execution of a program that occur between the entry and exit of one or more join points provided as its argument. Clearly, this does not refer to the syntax and structure of the program itself. This means that the patterns in the execution of the program can only be identified by taking into account the semantics of the programming language. Other examples of semantic language patterns are discussed in [GB03]; these are addressed by defining advanced pointcut expressions, which can reason about (the execution of) the program to determine the join points.

All these examples have in common that the pointcut expression is not just referring to the names and structure of the program, but can only be resolved by reasoning about the semantics of the language under consideration.

Discussion: Automatically-derived semantic properties are used to capture the intention of the designer by analyzing the semantic patterns of a program (execution). The two inputs to this analysis process are the program itself, and the semantics/meaning of the programming language involved. However, not all relevant semantic information/intentions can be derived from these sources: certain semantics are defined by the domain and normally not encoded into the program (as it is not required for the execution itself). We have previously mentioned the example of the persistence of individual program elements; this depends solely on the particular requirements of the application; it does not affect the behavior of the program itself, and may apply to both classes and the instance variables individually.

Since certain semantic information cannot be derived, we should be able to attach domain-specific design information as well. We will refer to this requirement as *dealing with domain-specific semantics*.

Note that naming and structural patterns are not suitable for semantic reasoning; however, they can enrich programs with domain-specific semantic properties.

2.5 Summary

In the previous sections, we have presented different ways how design information can be bound to, or derived from, different units of an application. Table 1 summarizes how the presented techniques support the identified requirements:
As the table shows, the only difference between naming and structural patterns is that the latter one can directly deal with multiple properties, although, this also depends on the structure of the applied programming language. Note that naming patterns can handle multiple properties by concatenating the names of the properties, and then, the framework can extract them, when it is necessary. However, as we have shown, this solution has more disadvantages than using structural patterns. Considering the application of annotations in the current AOP languages, none of the implementations except JBoss and AspectWerkz/AspectJ\(^3\) are capable of referencing annotations in pointcuts (indicated respectively by (1) and (2) at the superscripts in the third row). JBoss also supports the late binding of properties and scattered annotations to a limited extent. Note that semantic reasoning can only derive semantic information using the program and semantics of the applied programming languages, while it does not deal with domain and application specific properties.

According to the analysis, the combination of annotations and semantic reasoning seem to be the ideal solution to represent design information in AOP languages. However, there are certain problems that need to be addressed. Consider the following list of requirements:

1. Support explicit pointcut designators that can refer to annotations.
2. There must be means by which scattered annotations can be superimposed, and/or the place of annotations can be derived based on certain rules (e.g. the derivation of annotations can be driven by semantic reasoning).
3. There must be means to ensure the decoupling of design information from the base code; i.e. they must not be always statically bound to the program. Note that the superimposition of annotations is a natural solution to this problem. However, there are also other techniques that can ensure decoupling; we discussed this issue in details in [NBHA05].

### 3 General Approach

In the previous section, we have concluded that there is a need for selecting join points based on design information\(^4\). In this section, we will explain our general approach towards this problem. Figure 4(b) offers an overview of our approach, as compared to the state-of-the-art aspect-oriented approaches shown in Figure 4(a).

---

\(^3\) AspectJ and AspectWerkz have been recently merged into a common platform.

\(^4\) The need for expressing 'semantic pointcuts' was also identified in [THBL00].
In Figure 4(a), the white-colored shapes show the essential concepts of aspect-oriented programming, that are relevant to this paper\(^7\). The large box at the top represents the logical program elements of the (base) program. Program elements can be selected by pointcut specifications. This selection is typically based on the lexical and structural properties of the program elements or on the results of a more advanced program analysis techniques (for example, control flow analysis). The picture illustrates that superimposition is defined based on the specifications of advices\(^8\) and the corresponding pointcut specifications.

In Figure 4 (b), the grey parts illustrate the additional elements and relations that we propose in this paper. The key driver of our approach is, instead of referring directly to the program, new language abstractions are provided to specify pointcuts based on some design information (the grayed part of Figure 4 (b)). The specification of pointcuts incorporate references to the design information. We aim to select program elements based on the annotated design information. This is achieved by the design information reference layer, which allows for querying program elements based on their design information. We use the term semantic composition when the elements of a composition have been ‘collected’ by referring to design information. There are several ways in which design information may be associated with program elements:

- Attach them manually with custom attributes
- Derive them based on the existence of other design information.
- Attach them through superimposition of possibly crosscutting custom attributes (as indicated by the grey dashed arrow between design information and superimposition)

Finally, the figure also illustrates that the connection between superimposition and advice (i.e. the selection of advice), can now be made using design information; as such, also advice can now be selected based on its associated design information.

The key benefit of our approach is that it reduces direct dependencies between the crosscutting concern and the program source. It is realized by introducing a separate abstraction layer in between, which aims at describing the join point through specific

\(^7\) For some elements, we have written in italics the terminology for the corresponding composition-filters [BA04] concepts.

\(^8\) We use ‘advices’ for the plural form, as the term advice is meant as a language unit in this paper.
design information; this approach is considered more resilient to changes in the program and/or requirements.

4 Extending Compose* with Annotations

In this section we outline how design information annotations can be adopted in Compose* [Com][BA04], which is our aspect-oriented language implementation for .NET.

4.1 Definition of Annotations

Annotations have to be defined before they can be attached to a unit. In .NET annotations are defined as classes that extend the System.Attribute class. The following example shows the definition of the annotation Persistent:

```csharp
[AttributeUsage(AttributeTargets.Field)]
public class Persistent : Attribute()
```

*Figure 5:* The definition of the annotation Persistent.

The annotation AttributeUsage is a meta-annotation (i.e. an annotation bound to the definition of an annotation) defined by the .NET framework and it specifies that the annotation Persistent can be attached to only fields.

4.2 Annotation-based Selection

To designate join points based on annotations, pointcuts have to be able to refer to annotations. In Compose*, annotations can be referenced in two ways. Figure 6 illustrates this by an example. The first alternative is to extend the type patterns with annotations in the set of input filters, as it is shown at (1): [@Update *] means that every method with the Update annotation will be matched by the Meta filter. (Without the Update annotation, all methods would match.) For the Meta filter, when a method matches, it is reified and passed as a parameter to the updateAction() method that is executed on a PersistentManager instance.

The second alternative at (2) shows how logic predicates can be used to formulate queries based on matching annotations. This selector will designate all classes that have the annotation PersistentRoot by querying all units in the system and selecting those that match the applied predicates.

In the filtermodules part of the superimposition, the Updating filtermodule is superimposed on each class that is designated by the selector (c.f. query) named persistentClasses. Thus, every instance of these classes will have an instance of the Updating filtermodule superimposed.

```csharp
concern Persistence{
    filtermodule Updating{
        externals
        pm : PersistenceManager;
    }
}
```

*[In this paper, due to lack of space, we explain only the specific features of Compose* that are relevant for this paper. A more detailed discussion about Compose* can be found in [Com].*
inputfilters{
  redirect : Meta =
  (1)  {{@Update *] pm.updateAction();
         dispatch : Dispatch =
         {inner.*});
}
}
superimposition{
  selectors
  persistentClasses =
  (2)  { PersClass| isClass(PersClass),
            hasAnnotationWithName(PersClass,'PersistentRoot');
            filtermodules
            persistentClasses <- Updating;
  }
  implementation in Java; _
}

Figure 6: Definition of the Persistence concern (cf. aspect).

In this way, annotations are referenced two times. First, the intercepted methods are filtered based on the annotations in the filtermodule specification. Secondly, annotations are also referred via predicates, which are used as pointcut designators to superimpose the filtermodules.

4.3 Superimposition of Annotations

In the previous section we have shown how annotations can be referenced in selectors and type patterns within filters (i.e. they are the “pointcuts” of Compose*). In this section we illustrate how annotations can be superimposed\footnote{Using the terminology of AspectJ, superimposition means introduction here.} via selectors. A simple example is shown in Figure 7.

concern AssignAuthor{
  superimposition{
    selectors
    allGUITypes =
    (3)  { GuiType | isNamespaceWithName(GuiNS, 'com.myCompany.gui'),
           contains(GuiNS, GuiType)};
    attributes
    allGUITypes <- [Author("John Smith")];
  }
}

Figure 7: Superimposition of annotations.

The logic predicate (3) at the selector allGUITypes queries for all types (classes and interfaces) that are in the namespace of com.myCompany.gui.

The superimposition specification of Compose* has been extended with a new part called attributes. This part gives place to the specification of binding design information to selectors. In our example, we superimpose the annotation Author(“John Smith”) on allGUITypes at (4). As a result of this, the annotation Author(“John Smith”) will be
attached to every type (classes, interfaces, enumerations, etc.) within the namespace com.myCompany.gui.

5 Application of Semantic Properties

In this section, we will show some possible use of semantic properties, and especially, how semantic properties can be exploited to improve the reusability of aspects.

5.1 Derivation of Properties

Design information annotations can have dependencies that can be derived from the context information of the units they are being attached to. A simple example is presented in Figure 8:

```
class PersistenceView{
    superimposition{
        selectors
        nonTransFields =
            { Field if hasAttributeWithName(Class, 'PersistentRoot'),
              hasField(Class, Field),
              not(hasAttributeWithName(Field, 'Transient')));
        attributes
        nonTransFields <= [Persistent];
    }
}
```

Figure 8: An example of the derivation of attributes.

The selector nonTransFields will designate each field within every class that has the annotation PersistentRoot, excluding fields that have the annotation Transient, since we do not want to attach annotations Persistent and Transient to the same unit at the same time. Finally, we superimpose the annotation Persistent to the non-transient fields in the attributes part.

5.2 Decoupling Pointcuts and Advice

Design information annotations can be attached to filtermodules (i.e. the “advice” of Compose*) as well. Consider the example shown in Figure 9. There are two concern definitions in this example: the first concern is responsible for monitoring accesses to certain resources; the second one is just a simple tracing for debugging purposes. The filtermodules of the concerns realize a sort of monitoring; therefore both have the annotation Monitoring.

In the superimposition specification, the second selector (named monitoringModules) queries every filtermodule that has the ‘annotation Monitoring’. In the filtermodules part, the selector monitoringModules is bound to the selector criticalClasses. This means that every filtermodule that has the annotation Monitoring will be superimposed on the concerns queried by the selector criticalClasses. Therefore, both AccessMonitoring and LoggingModule will be superimposed in this case. The benefit of this approach is that filtermodules are indirectly referred in the superimposition; whenever a new filtermodule is defined with the annotation Monitoring, it will be automatically incorporated in the superimposition.
In general, advices may be decorated with design information as well. In this case, pointcuts should not be bound directly to the advice but must refer to the associated design information of the advice. That is, advices are indirectly referred through their design information in the weaving process. In this way, the aspctual behavior represented by an advice is woven if the advice holds a certain property. In a previous work [NBG+05], we have already presented a more detailed description about designating advices for evolvable weaving specifications. In the same paper, we have also presented an extensive trade-off analysis on combining annotations with advices.

Note that if several advices have the same semantic property, all of them will be woven at every designated join point. Naturally, the problem of shared joint points appears; i.e. join points where multiple advices are superimposed. This has complications e.g. regarding the ordering of advices at such a join point. We have addressed these issues in a separate paper [NBA04].

```java
concern SecurityLog(
  [Monitoring]
  filtermodule AccessMonitorinG{ _ } implementation in Java;
  -
}
concern Debugging(
  [Monitoring]
  filtermodule LoggingModule{ _ }

  superimposition {
    selectors
      criticalClasses = ( AnyRes | isClassWithNam emotE(Res, 'Resource'),
                         inInheritanceTree(Res,AnyRes) );
      monitoringModules ={
                        Module | isFilterModule(Module),
                         hasAnnotationWithNam e(Module,'Monitoring' ));

    filtermodules
      criticalClasses <- monitoringModules;
  }
  implementation in Java;
  -
}
```

Figure 9: An example of attributes on filtermodules

5.3 Defining Reusable Aspects

5.3.1 Problem Description and Example

We will now introduce an example to illustrate how design information can be utilized in the specifications of filters. The following source shows the example concern, Caching:

```java
concern Caching(
  filtermodule CachingModule{
    conditions
      isInvalidCache()
```

Note 1: Note that the full expression power of pointcuts could be used to select advices. This discussion is beyond the scope of this paper.

---

51
methods
getStoredValue();
inputfilters
  /*only for Circle*/
  Update : Meta =
  { isValidCache() =>
    [getPerimeter, getArea] updateStoredValue }
  /*only for Circle*/
  disp : Dispatch =
  { isValidCache() =>
    [getPerimeter, getArea] getStoredValue }
}
filtermodule MaintainCache{
  inputfilters
  /* only for Circle! */
  change : Meta =
  { [setRadius] setInvalidCache }
}

For the sake of efficiency, the computation of certain values is cached. CachingModule intercepts calls on a method which does the computation and instead of performing the method, it returns the previously computed value that is cached (see the Dispatch filter of CachingModule). Also, each call that can change the values used in the computation is intercepted and the state of the cache is set to invalid (see the MaintainCache filtermodule). When the state of the cache is invalid, the new value is computed and stored in the cache again, and the state of the cache becomes valid (see the Meta filter of CachingModule).

In this example code, the concern Caching is currently superimposed on class Circle which has the following interface:

class Circle
  methods that perform computation: getPerimeter, getArea
  the method that changes the values used in the computation: setRadius (only the radius is used in the computation methods)

For each method that returns a cached value (e.g. getPerimeter, getArea) a new instance of CachingModule should be superimposed to provide a stored value and a Boolean variable to indicate the state of the cache. That is, each join point should have a new instance of Caching.

Problem: In this example, the signatures of Circle (in bold) are explicitly referred to in the filters; thus, CachingModule cannot be adopted by a new concern with new computation methods (e.g. the method getVolume() of class Sphere). In general, the adaptability of filtermodules is limited because the signatures always have to be explicitly specified in the type patterns of the specification of the filters.

5.3.2 Solution & Example Revisited

By the application of design information, the concern Caching can be defined in a reusable manner so that programmers can customize it to different applications. The following figure presents a more reusable concern Caching:
concern Caching{
  filtermodule CachingModule{
    conditions
      isInvalidCache()
    methods
      getStoredProcedure();
    inputfilters
      /* for every computation method */
      Update : Meta =
      (1) (isInvalidCache() => [@Computation] updateStoredProcedure )
        /* for every computation method */
        disp : Dispatch =
      (1) (isInvalidCache() => [@Computation] getStoredProcedure )
  }
  filtermodule MaintainCache{
    inputfilters
      /* for every method that changes the input of the computations */
      (2) change : Meta =
        (1) [@ChangeInput] setInvalidCache
        ...
    }
  }
  class Circle{
    /* ============== */
    [Computation] public double getPerimeter(){ return ...; }
    [Computation] public double getArea(){ return ...; }
    [ChangeInput] public void setRadius(double r){ ... }
    ...
  }
}

Figure 10: Reusable Caching with Annotations

The original source of Caching has been changed in two places. At (1) the actual signatures of the methods that perform the computation are replaced by the annotation Computation. Thus, these methods are not directly referred through their names but they are referred through the annotation Computation. Similarly, at (2) the actual signatures of the methods that can change the input values of a computation are replaced by the annotation ChangeInput. Thus, these methods are also indirectly referred through the annotation ChangeInput. Note that these design information, in principle, are intended to denote the design rationale of these methods.

It is also necessary to attach the above-mentioned annotations to the right methods in the base code. A possible solution is to embed “manually” the annotations in the source of the base classes. Thus, in our example (in the right column of Figure 10), the getPerimeter and getArea methods have to be tagged by the annotation Computation, since they perform the computation to be cached. Similarly, the method setRadius has to be labeled by the annotation ChangeInput, since it changes the input values of the computation. Note that embedding annotations in the base code can lead to several problems; instead of this technique, the superimposition of these annotations might provide a better solution. The issue is discussed in details in [NBHA05].

Discussion: Using annotations in the filters instead of using explicit signatures allows for defining reusable filtermodules. Thus, if a filtermodule contains only annotations, it will be free of signatures that are specific to an application. To customize the filtermodules to different applications, the annotations referred in the filtermodules need
to be attached to the necessary units in the base code. By the combination of these mechanisms (using annotations in filtermodules + superimposing annotations on the base code), methods are indirectly intercepted based on their semantic properties. As result of this, concerns can be adopted in a generic way.

In terms of AspectJ, this technique is equivalent to referring to annotations in pointcuts. Note that the same problem could be handled by abstract pointcuts as well. However, by using design information and generic aspects, the customization of aspects is “automatically” managed, as long as design information is used in a disciplined manner.

6 Related Work

AspectWerkz [AspWerkz] is a dynamic aspect-oriented framework for Java that is capable of embedding aspects into the base code through annotations. In other words, there is a set of custom annotations that expresses an AOP language, and one can write his or her aspects by using these custom annotations in the base code. AspectWerkz allows for matching on annotations in pointcuts; however, it does not support introducing annotations (this functionality is mentioned as a future work). Hence the problem of scattered annotations and late binding is also not addressed.

JBOSS AOP [JBOSS][Bur04] is a Java based aspect-oriented framework usable in any programming environment and integrated with the application server of JBOSS [JBOSS]. The framework allows for matching on annotations in pointcuts, as well as introducing annotations to a limited set of units (classes, methods, constructors and fields) by its pointcut language. In this way, the problem of scattered annotations can be handled for these types of units. (However, Java 1.5 allows for more types than only these four; e.g., annotations can be attached to packages, annotations, etc. according to the JSR-175 [Sun02] specification.) Late binding is also possible in the framework; besides inserting the annotations into the bytecode of classes (static binding), the annotation compiler can generate a separate XML file that contains the metadata (i.e., the binding of custom attributes).

JQuery [JV03] is a flexible, query-based source code browser, developed as an Eclipse plug-in. In JQuery, users can define their own queries to select certain elements of a program. The JQuery query language is defined as a set of TyRuBa [Vol98] predicates which operate on facts generated from Eclipse JDT’s abstract syntax tree. The predicates of JQuery are dedicated for Java and the factbase of JQuery is based on Java sources and bytecode files. In Compose* we also use a predicate language to specify selectors as queries. Our predicates are dedicated for Compose* and the factbase is based on the repository model of a Compose* project.

G. Kiczales in [Kic04] proposed a pointcut matching mechanism based on annotations and using introductions with annotations in AspectJ. A. Colyer’s in his blog [Col04] also had a proposal to extend the pointcut language of AspectJ to do matching based on annotations. He also shows a proposal of how annotations can be introduced through a new language construct, called declare annotation. Using type patterns in this construct allows for introducing annotations over multiple units, i.e. it addresses the problem of scattered properties. However, type patterns have a limited expressiveness to designate units that annotations are attached to, as compared to the predicate language we propose.
The difference is that type patterns can designate units based on only the own properties of the units to be designated, while our predicate language can designate units based on the context of units, as well. Typically, relationships with other units (e.g. inheritance, aggregation) and properties of related units make up the context of a unit. Note that the syntax given in [Col04] was for illustrative purposes only. Recently, in the latest version of AspectJ, the above mentioned proposals had been implemented as well.

R. Laddad in [Lad05] investigates the application of metadata in AOP. In this work, he gives practical hints and guidelines how to use and how not to use annotations in combination with AOP, particularly, with AspectJ. In our paper, we also investigated various novel application possibilities of using annotations in AOP, such as providing reusable aspects and evolvable weaving specifications (for instance, by designating advices based on annotations.)

7 Discussion and Conclusion

In this section, we discuss the consequences of the techniques we proposed in this paper, how they can be realized, and finally we summarize the paper and outline some future work.

7.1 Benefits and contribution

The primary contribution of this paper is that it presents an in-depth analysis of the role of design information within the context of aspect-oriented programming. We have shown that there is a need for annotating programs with design information. We also demonstrate that the integration of design information with aspect-oriented composition (‘semantic composition’) mechanisms offers a means of coupling that is both manageable and powerful. The main benefits of design information annotation that we have introduced in this paper are:

- The ability to select join points based on design information (which could never be derived from the program itself).
- The programmer can freely choose what the appropriate locations are to define annotations: within the code, co-located with the code or in an aspect.
- The usage of design information makes aspects, especially pointcut expressions, less vulnerable to changes of the program. The reason is that they avoid dependencies of the pointcut expression upon the structure or the naming conventions of the program.
- The dependencies between program elements can be more precise and easier to understand by referring to the design intentions instead of syntactic structure or naming (if the latter would exactly express the intention, it would be preferred, though).
- As we showed in section 5.3, our proposal supports the definition and customization of reusable aspects.

The related work section discussed several recent programming language implementations that offer --to varying degrees-- implementation techniques that are necessary to deal with design information. However, none of that related work has
pointed out the problems of current ways to deal with semantic properties, or explicitly motivated the need for applying these techniques. Also several useful ways to apply annotations in the context of aspect-oriented programming that we explained in section 5 have not been proposed before. Only the work of Gybels et. al. in [GB03] explicitly discusses these problems. However, the solution they propose is based on automatic derivation of annotations, which cannot always be realized, as we pointed out in section 2.4. A preliminary version of the analysis and the solution presented in this paper was given in [NB04].

7.2 Limitations and suggestions

This section lists a number of limitations and other issues that result from our proposal, together with some suggestions how these can be tackled:

- Disciplined programming is still required to keep the correct design information associated with the appropriate program elements. We believe that through the language abstractions proposed by this paper, the situation can be improved however: first of all; certain semantic properties have to be specified by the software engineer; this is unavoidable. Second, we have illustrated how superimposition and derivation can be used to attach semantic properties. In addition, we plan two ways to address this issue: (1) by investigating design-level support and the automatic derivation of annotation specifications from stereotypes in UML diagrams. (2) By searching for techniques that can automatically derive certain common annotations.

- Similarly, it is important that the software engineers use a consistent and coherent set of design information for each sub domain of an application (whether from a technical/solution domain, or from the application/problem domain). For example, if programmers use terms such as ‘setter’, ‘writer’, or ‘updater’ inconsistently, our approach has limited value. We believe that the development and use of ontologies for identifying a consistent set of semantic properties in a particular domain can be a useful approach.

- The annotated design information may require parameters for passing context information; if the property is superimposed, it is typically hard to include such context information. One alternative to deal with this is by accessing the context through a generic reflective interface.

- Selectors can match only by the type of an annotation; the arguments of an annotation (or the member values of annotation instances) cannot be referred to in the current implementation. This issue is going to be resolved in the subsequent releases of our tool by providing additional predicates to formulate selectors based on that information as well.

- One may consider a case where a semantic property is introduced through superimposition (without derivation rules) and then the occurrences of that same property are used in a pointcut expression to select join points. This is an example

---

12 In fact these appear to be rather technology-driven by the introduction of annotations in Java 5 [Aus04]. However the motivation of introducing annotations in java is formulated as: “One of the primary reasons for adding metadata to the Java platform is to enable development and runtime tools to have a common infrastructure and so reduce the effort required for programming and deployment.” [Aus04].
style that our approach is not aiming at, as this could have been expressed directly in a single pointcut expression.

7.3 Realization

As we indicated in section 6 on related work, with the advent of semantic property annotations (‘attributes’) in Java 1.5, recently several Java-based aspect systems have announced that they offer, or will offer, the integration of annotations with aspects. This means the underlying core technology for achieving our proposal is largely available today. However, in section 6 we discussed some further limitations of most of these approaches.

We have shown the integration of semantic properties with Compose®. The currently released version of the implementation allows only an extremely simple implementation of the pointcut language in the selectors. The predicate-based language we have used in this paper has been prototyped and tested. At the time of writing, this must still be integrated with the semantic property annotations offered by .NET. One of the issues that must be addressed though, is due to the fact that our tools work with objects after they have been compiled into .NET assemblies: this means that a potential compile-time impact of semantic properties that have been superimposed would be ignored in this approach. We intend to address this by partial recompilation of the source after manipulating it, in a –completely language-independent– representation of the source code offered by Visual Studio.

Clearly, the order of superimposition of semantic properties can also be relevant. For example, in section 5 there is an ordering dependency between the properties PersistentRoot and the Persistent: PersistentRoot should be attached first, since the superimposition specification of the annotation Persistent refers to PersistentRoot (see Figure 8).

Assuming that there is no negation in our pointcut language and there are no negative actions, i.e. we only add annotations, not remove or modify them, a solution like in [KCA01] solves this ordering problem. However, as the example at Figure 8 shows, we would like to use negation in our pointcut language. For this reason, one of our future works is to perform an analysis on the superimposition specification of annotations in order to determine a right order for evaluating the predicates and detect possible conflicts (e.g. cyclic dependencies) in the specifications.

7.4 Conclusion

Software that is developed today is making frequent use of design information that is encoded in some way into the source code. In this paper we argue that it is unavoidable to add such design information (a) since not all relevant design information can be derived from the executable source code, and (b) when one wants to refer to program elements based on design intentions, rather than trying to capture the right set of elements by referring to their ‘accidental’ lexical and structural properties.

13 Pronounced as “compose-star”, this is an open-source project that aims at implementing composition filters. A first release is available from http://composestar.sf.net; this implements composition filters on top of the .NET object model (i.e. independent of a specific .NET language).
In section 2 we have analyzed four techniques currently used for binding and deriving design information in traditional object-oriented languages, such as Java or C#. We have illustrated how aspect-oriented software often utilizes design information in its pointcut expressions, and we have presented deficiencies when using any of these four techniques.

To conclude the problem analysis, we have formulated three requirements for adopting design information in aspect-oriented programming:

1. Pointcut expressions need to refer to semantic properties.
2. There is a need to support the superimposition of semantic properties.
3. Design information must not be bound statically but be attached to a unit in the program in a flexible way.

In section 3 we have discussed how design information can be used in the superimposition of aspects and also, how superimposition can be applied to bind design information to the base code. We have also identified and analyzed the language abstractions which can be used to represent design information. Based on this analysis, in section 4 we have shown how a concrete aspect-oriented language (Compose*) can be extended to support modeling design information in such a way that the above mentioned requirements are fulfilled:

1. Section 4.2 has presented how design information can be used to designate join points.
2. Section 4.3 has illustrated how design information can be superimposed or derived.
3. The bindings between the design information and units are expressed by superimposition specifications and localized in a concern, which is the aspectual module of our language (see section 4.3 and [NBHA05]).

In section 5, we have discussed further application of the presented techniques, the most important result is the ability to define generic aspects that can be customized and are flexibly connected to all join points that exhibit the correct semantic properties; as the software evolves or is refactored, this connection is likely to remain valid.

A number of research topics for future research have appeared as the result of this work; some have been mentioned as suggestions for addressing some of the limitations, in section 7.2. Other potential future work is about (a) the ability to apply the notion of semantic composition to more composition techniques than the superimposition mechanism that is the main subject of study in this paper, or (b) the exploitation of semantic composition for the purpose of modeling product lines and variability management.

Finally, we observe that the technology for using design information together with AOP is becoming available, and we believe that this paper can contribute to a better understanding of the importance of using design information and the possible applications.

8 References

[Com] Compose* project: http://composesstar.sf.net
[JBOSS] JBOSS project: http://www.jboss.org
[JUnit] JUNIT project: http://www.junit.org
[Lad05] Laddad, R., AOP and metadata: A perfect match, AOP@Work Series, IBM Technical Library, March 2005


