# Join Point Selectors

Cristiano Breuel Francisco Reverbel
Department of Computer Science
University of São Paulo
{cmbreuel,reverbel}@ime.usp.br

## Abstract

One of the main issues in modern aspect-oriented programming languages and frameworks is the expressiveness of the pointcut language or mechanism. The expressiveness of pointcut languages directly impacts pointcut quality, a property that can be decisive for the effectiveness of aspect implementations. In this paper we propose join point selectors as a simple extension mechanism for enriching current pointcut languages with constructs that play the role of "new primitive pointcuts". Join point selectors allow the creation of pointcuts with greater semantic value. Although similar mechanisms can be found in some existing approaches, the underlying concept has not yet been clearly defined nor fully explored. We present a simple architecture for adding join point selectors to an existing aspect-oriented framework. We show examples of usage of join point selectors to enhance the quality of pointcuts and make aspect development easier. We also show how join point selectors can be used as framework-specific selectors, which allow aspects to cross the boundary of a given framework while still respecting the modularity of that framework.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Aspect-Oriented Programming, Pointcut Languages

***Keywords*** AOP, Semantic Pointcuts, Extensibility, Join Point Selectors

## 1. Introduction

We can think of aspect-oriented programming (AOP) as a way of optimizing the creation of computer programs. The constraints to this optimization are the abilities of humans and computers. Fortunately, these two sets of abilities are complementary. Humans excel at pattern recognition and abstraction, but are very inefficient and prone to error in repetitive tasks. Computers, on the other hand, are extremely efficient and immune to errors in repetitive tasks, but do not have the human capabilities in abstraction and pattern recognition. Thus, the way AOP achieves the optimization is by improving the distribution of tasks: it allows humans to express crosscutting concerns in the form of patterns, and lets the computer apply these patterns in all necessary points.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*Workshop SPLAT '07* March 12-13, 2007 Vancouver, British Columbia, Canada.
Copyright © 2007 ACM 1-59593-656-1/07/03. . . $5.00.

The objective of our work is to further improve such task distribution, by raising the level of abstraction in the creation of pointcuts. The concept we propose, *join point selectors*, is a mechanism for creating pointcuts with greater semantic value. We believe that dealing with higher levels of abstraction enables developers to be more efficient and to reduce errors in pointcut programming. In addition, the open nature of our mechanism makes it possible to use heterogeneous artifacts as input for pointcut expressions.

This paper is organized as follows: Section 2 discusses the problem we want to solve and introduces the notion of pointcut quality; section 3 presents the concept of join point selector; section 4 describes our prototype implementation; section 5 contains examples of join point selector usage; section 6 discusses related work; and section 7 presents our concluding remarks and future work ideas.

## 2. Motivation

The AspectJ language [11] introduced a model for AOP that has been widely accepted and adopted as a reference for many other aspect-oriented languages and frameworks. These include industry projects such as JBoss AOP [9], Spring AOP [10], and AspectWerkz [21]. As pointed out in [13], they all share similar capabilities and semantics, despite the different syntaxes and implementation approaches.

Although the pointcut languages in these tools are sufficiently capable for current practical use, there has been some criticism about their limitations. The most frequent concern is that a pointcut may be "broken" by changes to the base program (e.g. [4, 7, 15, 19]). Another issue is the difficulty or impossibility of expressing some pointcuts clearly and accurately [12].

We have also identified a reason, which in our view has not been extensively discussed before, for making pointcut languages more flexible: the use of meta information about the code as selection criteria for pointcuts. This kind of meta information is very common in modern frameworks (e.g. Hibernate [1], Spring [10], Struts [20]). The only particular case that has been adequately covered and supported by existing aspect-oriented languages is when this meta information is encoded as Java 5 annotations. However, in the most general case when the meta information can be stored in other ways (e.g. comment tags, XML files, proprietary configuration files), conventional aspect-oriented languages are not well-suited. This topic is explored in section 5.5.

### 2.1 Pointcut Quality

To better discuss the issues that arise in comparing pointcut languages, it is useful to define some criteria for comparing pointcut definitions. We define *pointcut quality* as the extent to which a given pointcut meets the following requirements:

- **Resilience**: changes in the base program should not affect the pointcut negatively. More specifically, when a new join point is added to the program or an existing one is modified, the join

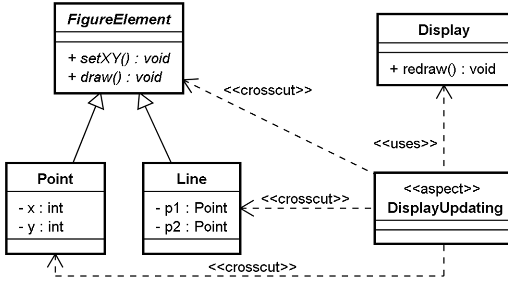


**Figure 1.** The Figure Editor class and aspect structure

point should be included in the set selected by the pointcut if and only if it matches the conditions intended by the pointcut author. This property influences the pointcut's evolvability and the modularity of the aspect implementation. A good methodology for analyzing this characteristic is found in [19].

- **Clarity of purpose**: a pointcut definition should make its intent clear to whoever reads it, and should be expressed in terms that are as close as possible to the problem at hand, to achieve good comprehensibility. That makes changes to it easier, because the changer knows what the pointcut should do.

The ability that a programmer has of creating high quality pointcuts is limited by the expressiveness of the pointcut language. Thus, our objective is to provide tools that allow programmers to create pointcuts of better quality, according to this definition. In the following section, we present an example that illustrates this concept.

### 2.2 Example

One of the most frequent examples regarding AspectJ usage, the figure editor [11], is also the most commonly used to expose its shortcomings (e.g. [4, 5, 14]). It consists of a graphical editor with several types of elements (squares, circles etc.) whose display must be updated every time that the state of the elements changes. The program manages elements in the display as instances of class `FigureElement` and its subclasses (figure 1). We want to create a "display updating" aspect that calls the `Display.redraw()` method when the state of the elements is altered.

The classic solution is an aspect that selects methods based on a naming convention, such as picking all methods in class `FigureElement` and its subclasses whose names start with "set". This is a low quality pointcut, because it is not very resistant to change (for instance, if a method that changes the state and does not start with "set" is implemented, it will not be selected) and does not clearly express its intent (the reader must guess it from the method prefix).

Another solution is to create an annotation that must be associated with updater methods, for example, `@FigureUpdater`. This is an improvement on the clarity of the pointcut, but it is still not very resistant to change, because one can forget to use the annotation. Therefore, the quality of this pointcut is intermediary.

If we could specify a pointcut that explicitly selected all methods that alter fields read by the `Display.redraw()` method, then such pointcut would be of high quality. First, because it states exactly what we intend to capture, and second, because it is resistant to change.

This last type of pointcut is the one that we wish to allow programmers to create.

## 3. Concept

In order to allow programmers to create pointcut definitions with higher semantic value, and therefore enhance pointcut quality, we propose a new construct in aspect-oriented languages – the join point selector[1].

A join point selector is a function that, for a given set of arguments and a joint point, determines whether the join point fulfills a certain condition. When used to compose a pointcut expression, the selector indicates the fitness of the join point as part of the pointcut.

Selectors can operate either at weave time or at run time. A weave time selector may have an associated run-time version, such that the latter will be called if there is not enough information to decide the selection at weave time.

In current aspect-oriented languages and frameworks, the concept of selectors is represented by what is sometimes called "primitive pointcuts" or "primitive pointcut designators", such as "call", "execution" etc. However, the programmer cannot define new selectors, because join point selection algorithms are hard coded into the weaver. In some research papers, similar mechanisms have been proposed with different names, like "pointcut designator" or simply "pointcut". We consider that our proposed naming is important because it distinguishes the algorithm (the selector) from its use in specific instances (the pointcut expressions). The examples below illustrate this distinction:

`call` is a selector.

`call(void *->setSize(..))` is a pointcut expression.

The core characteristics that define join point selectors, and distinguish them from similar mechanisms, are the following:

1. **They can receive arguments.** When used in pointcut expressions, a selector can receive arguments that are taken into account by its algorithm to make a decision.
2. **They can be combined by simple expressions.** Selectors can be combined by simple boolean expressions to create a pointcut definition.
3. **They operate both at weave time and at run time.** A simple and uniform mechanism is used to allow selectors to use weave-time information, run-time information, or both.

These characteristics make selectors a basic unit of functionality. In pointcut expressions, they perform the same role that methods perform in object-oriented languages and advice perform in aspects. Some of these characteristics are found in previously available mechanisms, but the combination of all of them makes join point selectors more expressive and easy to use.

## 4. Implementation

As a proof of concept, we implemented the selector functionality as an extension to the JBoss AOP [9] framework. The choice of JBoss AOP as a basis for the implementation was not made for conceptual reasons, but due to practical factors. Our approach is not limited to this specific framework, and could be implemented in other aspect-oriented languages or frameworks that have similar concepts.

The implemented extension consists of:

- A modification to the pointcut grammar to recognize selectors in pointcut expressions;

[1] In [18], the term "selector language" is used as a general denomination for pointcut languages and other mechanisms for join point selection. We think that the risk of confusion between these terms is low, because they are used in different contexts.

- The addition of new elements to the XML and annotation bindings to allow for the declaration of selectors;
- Changes in the weaver to add calls to run-time selectors where necessary.

### 4.1 Basic Architecture

A selector is an ordinary Java class that implements the interface `org.jboss.aop.selector.Selector`. Listing 1 shows this interface's declaration.

Although we defined the concept of a selector as a function, we decided to implement it as an interface with several methods, one for each type of join point. That was done for practical reasons, since JBoss AOP already had a `Pointcut` interface with similar methods, making it convenient to use a similar approach. We can think of a `Selector` object as a way of grouping related selector functions.

The interface has two groups of methods, one for weave-time selectors and the other for their run-time versions. In each group, there is one method to treat each type of primitive pointcut (method calls, attribute get and set, etc.). Thus, each method has a counterpart in the other group. All methods receive a set of parameters that represent the join point to be evaluated. They also receive a reference to an `Advisor` object, which gives access to some JBoss AOP facilities. Finally, they receive a list of `SelectorParam` objects that encapsulate the arguments given in the pointcut expression where the selector is being used.

In the weave-time selector methods, the types of the parameters that reify join point shadows [8] (e.g. `CtMethod m`, `CtField f`) are part of the Javassist [3] API. Javassist is a framework for structural reflection that reads and manipulates Java bytecode, but provides a high-level API that allows the programmers to deal with elements of the Java language, instead of bytecode details. This framework is used as a basis for all weave-time manipulation in the JBoss AOP framework. Through this API, selector developers have access to the program structure in a more powerful way than the standard Java reflection API would allow. This makes it possible to create selectors that examine the internal structure of constructs such as classes or methods in order to select join points. An example of such a selector will be presented in section 5.2. It would also be possible for the selector programmer to manipulate the bytecode through the Javassist API. In most situations, however, there will be no need for explicit bytecode manipulation by programmer, as the weaver automatically inserts calls to run-time selector methods wherever such calls are necessary.

In our current prototype, selectors do not yet have all the expressiveness that they might have, because the only type of argument that a selector can receive is a `String`. In future versions, we intend to remove this constraint.

Every weave-time selector returns a value of the enumeration type `SelectionValue`, whose definition is given in listing 2. This type can have one of three values: `TRUE` (the join point matches the condition), `FALSE` (the join point does not match the condition) and `CHECK_AT_RUNTIME` (it is not possible to decide without run-time information). If a weave-time selector returns this last value, the weaver is required to insert a call to the respective run-time selector into the join point shadow in the base code. Figure 2 shows this sequence of events.

This architecture is designed to allow selectors to use both weave and run-time information, without complicating the pointcut language. It also relieves the selector programmer from having to use explicit code generation and bytecode manipulation techniques, which can be hard to understand and apply.

```
public enum SelectionValue {
    TRUE,              // Matches
    FALSE,             // Does not match
    CHECK_AT_RUNTIME   // Needs run-time information to decide
}
```

**Listing 2.** The SelectionValue enumeration type

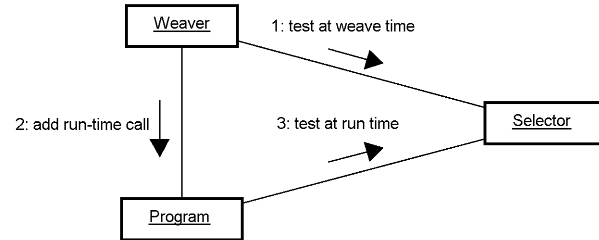


**Figure 2.** The operation of a selector

### 4.2 Declaring a Selector

For the weaver to recognize a new selector, this selector must be declared in one of two ways: in an XML descriptor or through an annotation in the class itself. JBoss AOP support both of these methods for its current features, so we followed the same style. In listings 3 and 4 we see examples of the Java 5 annotation and XML declaration styles, respectively.

The annotation can take only one optional parameter, the selector name to be used in pointcut expressions. If omitted, the selector name will be the class name (without package specification). The XML version needs another parameter, the fully qualified name of the class that implements the selector.

```
@SelectorDef(name="parameterTypeIs")
public class ParameterTypeSelector implements Selector {
...
```

**Listing 3.** A Java 5 annotation selector declaration

```
<selector name="parameterTypeIs"
          class="org.jboss.test.aop.ParameterTypeSelector"/>
```

**Listing 4.** An XML selector declaration

To avoid the need to implement all methods in the `Selector` interface when only some of them are necessary, we created the class `SelectorHelper`. This class implements all selector methods by returning either `SelectionValue.FALSE` or `false`, depending on the method type. To use these default implementations, the programmer can inherit from this class instead of directly implementing the interface.

### 4.3 Usage

A selector is used as a clause in a pointcut expression, combined with other selectors by boolean operators. It has the same syntax as a method call: the selector name, followed by a comma-separated list of arguments, which is enclosed by parentheses. In our current prototype, the only type of argument supported is `String`, so the arguments must be in quotes. Because pointcut expressions in JBoss AOP are specified either as Java annotation arguments or XML tag properties, both of which are already in quotes, the inner quotes must be escaped following each representation's conventions.

```
public interface Selector {

    /* Weave-time selector methods */
    SelectionValue matchesExecution(Advisor adv, CtMethod m, List<SelectorParam> selectorParams);
    SelectionValue matchesExecution(Advisor adv, CtConstructor c, List<SelectorParam> selectorParams);
    SelectionValue matchesConstruction(Advisor adv,CtConstructor c, List<SelectorParam> selectorParams);
    SelectionValue matchesGet(Advisor adv, CtField f, List<SelectorParam> selectorParams);
    SelectionValue matchesSet(Advisor adv, CtField f, List<SelectorParam> selectorParams);
    SelectionValue matchesCall(Advisor callingAdv, MethodCall mc, List<SelectorParam> selectorParams);
    SelectionValue matchesCall(Advisor callingAdv, NewExpr mc, List<SelectorParam> selectorParams);

    /* Run-time selector methods */
    boolean matchesExecution(Advisor adv, Method m, List<SelectorParam> selectorParams);
    boolean matchesExecution(Advisor adv, Constructor c, List<SelectorParam> selectorParams);
    boolean matchesConstruction(Advisor adv, Constructor c, List<SelectorParam> selectorParams);
    boolean matchesGet(Advisor adv, Field f, List<SelectorParam> selectorParams);
    boolean matchesSet(Advisor adv, Field f, List<SelectorParam> selectorParams);
    boolean matchesCall(Advisor adv, AccessibleObject within, Class calledClass, Method calledMethod, Object target,
                        Object[] args, List<SelectorParam> selectorParams);
    boolean matchesCall(Advisor adv, AccessibleObject within, Class calledClass, Constructor calledCon,
                        Object[] args, List<SelectorParam> selectorParams);

}
```

**Listing 1.** The Selector Interface

Listings 5 and 6 show how to use selectors in pointcut expressions defined through Java 5 annotation and XML bindings, respectively.

```
@Bind(pointcut = "parameterTypeIs(\"0\",\"java.lang.Integer\")")
public Object advice(Invocation invocation) throws Throwable {
  ...
```

**Listing 5.** A Java 5 annotation pointcut definition using a selector

```
<bind pointcut=
"parameterTypeIs(&quot;0&quot;,&quot;java.lang.Integer&quot;)"
>
    <interceptor class="com.acme.SomeAspect"/>
</bind>
```

**Listing 6.** An XML pointcut definition using a selector

## 5. Examples

In the following sections, we give examples of how selectors can be used to improve pointcut quality. These examples are meant as a sample of the anticipated uses of join point selectors. Due to the open nature of such constructs, it is not possible to anticipate all of their practical applications.

Most of the listings in the examples are simplified for clarity. We omitted code sections that perform tasks that are not central to this discussion, such as error handling.

### 5.1 Parameter Type

Our first example is borrowed from [4] (with a slight generalization), and consists of a selector that picks methods with a parameter of a specified type in the specified position. For example, it can select all methods whose first parameter is a `String`. The sole purpose of this example is to show how selectors could be employed to solve a problem that does not have a simple solution in conventional aspect-oriented languages, even though it does not seem to have much practical use. Listing 7 shows this selector's implementation.

The first method is the weave-time selector. If the type of the parameter at the specified position is the same or a subtype of

```
@SelectorDef
public class ParameterTypeIs extends SelectorHelper {

 /** Weave-time method execution selector */
 public SelectionValue matchesExecution(Advisor advisor,
   CtMethod m, List<SelectorParam> params) {
  // Gets selector parameters
  int paramIndex = Integer.parseInt(params.get(0).getValue());
  String paramTypeName = params.get(1).getValue();
  // Obtains type of the method parameter
  CtClass parType = m.getParameterTypes()[paramIndex];
  // Obtains CtClass whose name is given by the first selector
  // parameter, which is the desired type
  CtClass argType = ClassPool.getDefault().get(paramTypeName);
  // Tests for compatibility between types. The rule is the
  // same as for Java type casts.
  if (parType.subtypeOf(argType)) {
   return TRUE;
  } else if (argType.subtypeOf(parType)) {
   return CHECK_AT_RUNTIME;
  } else {
   return FALSE;
  }
 }

 /** Run-time method execution selector */
 public boolean matchesExecution(Advisor advisor, Method m,
                            List<SelectorParam> params) {
  // Gets selector parameters
  int paramIndex = Integer.parseInt(params.get(0).getValue());
  String paramTypeName = params.get(1).getValue();
  // Obtains type of the method parameter
  Class parType = m.getParameterTypes()[paramIndex];
  // Obtains Class object for the wanted type
  Class argType = null;
  try {
   argType = Class.forName(paramTypeName);
  } catch (ClassNotFoundException e) {
  }
  // Tests for run-time compatibility between types.
  return (parType.isInstance(argType));
 }
}
```

**Listing 7.** A selector to pick methods with a parameter that is compatible with a specified type

the desired type, the join point is matched. If it is a supertype of the desired type (for example, it is an `Object` while we want an `Integer`), it defers the decision to run time. In any other case, the join point does not match. The second method tests at run time whether the concrete argument given in the call is compatible with the desired one.

This example shows how we can use both weave and run-time information to build a selector algorithm in a simple way. It makes decisions at weave time whenever possible, thus minimizing the impact of run-time checks. This implementation could still be optimized, for example, by caching some objects.

### 5.2 Updater

As seen in section 2.2, the Figure Editor example cannot be implemented in conventional aspect-oriented languages with high pointcut quality. To solve the problem in a better way, it is necessary to identify the methods that alter the state of the figures, which is later read by the redraw method. Here, we propose a specific selector that uses weave-time information to determine which methods could potentially alter the state. This is a simple solution based only on static information, but more elaborate ones could also be developed with the selector mechanism. Figure 8 shows an outline of how the solution could be achieved.

```
@SelectorDef(name="updatesStateReadBy")
public class UpdatesStateReadBy extends SelectorHelper {

 /** Weave-time method execution selector */
 public SelectionValue matchesExecution(Advisor advisor,
   CtMethod m, List<SelectorParam> params) {
  // Gets selector parameters
  String readerTypeName = params.get(0).getValue();
  String readerMethodName = params.get(1).getValue();
  // Obtains the reader method
  CtClass readerType =
   ClassPool.getDefault().get(readerTypeName);
  CtMethod readerMethod =
   readerType.getDeclaredMethod(readerMethodName);
  // Gets the sets of read and updated fields
  Set<CtField> readFields =
   getFieldsReadByMethod(readerMethod);
  Set<CtField> updatedFields =
   getFieldsUpdatedByMethod(m);
  // Compares sets
  boolean result = readFields.removeAll(updatedFields);
  return (result ? TRUE : FALSE);
 }
 /**
  * Finds (possibly a superset of) the set of all fields
  * updated by the given method, including those those updated
  * within calls to other methods (searches recursively).
  *
  * @param m the method to examine.
  * @return a set of fields updated by m.
  */
 private Set<CtField> getFieldsUpdatedByMethod(CtMethod m) {
  ...
 }
 /**
  * Finds (possibly a superset of) the set of all fields
  * read by the given method, including those those read
  * within calls to other methods (searches recursively).
  *
  * @param m the method to examine.
  * @return a set of fields read by m.
  */
 private Set<CtField> getFieldsReadByMethod(CtMethod m) {
  ...
 }
}
```

**Listing 8.** A selector to pick join points that update fields read by a given method

The selector's private methods `getFieldsUpdatedByMethod` and `getFieldsReadByMethod`, which are not shown, make the recursive searches for fields that are updated or read in the control flow of a given method. Because of inheritance and conditionals that might affect the result, these methods work with the worst case scenario. They guarantee that no correct answer will be left out, but might also have false matches. More elaborate solutions have been proposed ([14], [19]), involving the creation of new pointcut languages. We believe that such solutions could be implemented as selectors, without the need for a new language, but we have not validated this hypothesis yet.

The quality of the pointcuts created with this selector will be higher, because the pointcut will be more resilient to changes and its intent will be more clear to the reader than a naming pattern.

### 5.3 Reflective Calls

Method calls using reflection APIs are becoming very common, especially in frameworks and middleware. Conventional aspect-oriented languages make it a difficult task to advise these calls in the caller side, because their pointcut languages do not have enough expressiveness to filter those kinds of indirect calls. The usual solution is to do some filtering in the advices, instead of doing it in the pointcuts, a practice that breaks the intended semantics of these AOP constructs.

Our proposed selector (figure 9) makes creating pointcuts with this kind of call as simple as with conventional calls.

```
@SelectorDef(name="reflectiveCall")
public class ReflectiveCall extends SelectorHelper {

 /** Compile-time method execution selector */
 public SelectionValue matchesCall(Advisor callingAdvisor,
    MethodCall methodCall, List<SelectorParam> params) {
  CtMethod invoke = ClassPool.getDefault()
                     .get("java.lang.reflect.Method")
                     .getDeclaredMethod("invoke");
  // Checks if the called method is Method.invoke(). If so,
  // the joinpont must be checked at run time, otherwise it
  // can be discarded.
  if (methodCall.getMethod().equals(invoke)) {
   return CHECK_AT_RUNTIME;
  } else {
   return FALSE;
  }
 }

 /** Run-time method execution selector */
 public boolean matchesCall(Advisor advisor,
   AccessibleObject within, Class calledClass,
   Method calledMethod, Object target,
   Object[] args, List<SelectorParam> selectorParams) {
  // Pre-process selector parameters
  String methodClassName = selectorParams.get(0).getValue();
  String methodName = selectorParams.get(1).getValue();
  // Obtains Method object for the target method
  Class argType = null;
  try {
   argType = Class.forName(methodClassName);
  } catch (ClassNotFoundException e) {
   return false;
  }
  Method method = getMethod(argType, methodName);
  Method targetMethod = (Method) args[0];
  return targetMethod.equals(method);
 }
 /**
  * Gets the method with the given name in the given class
  */
 private Method getMethod(Class c, String methodName) {
  ...
 }
}
```

**Listing 9.** A selector for reflective method calls

When a call is made through reflection, the called method can be known only at run time. Thus, the weave-time part of this selector simply flags all calls to the `invoke()` method in `java.lang.reflect.Method` as requiring run-time analysis. The run-time part captures the target method and returns `true` if that method is the one whose name was specified as a selector parameter.

### 5.4 Domain-Specific Languages

Many domain-specific languages (DSLs) have been proposed to tailor aspect-oriented programming to specific problem domains. One of the reasons for creating these specific languages is the lack of an adequate level of expressiveness in pointcut languages of conventional AOP approaches. In some cases, it is not possible to express pointcuts with the necessary detail. In others, a solution would be possible by a combination of pointcuts and advice, but it would not be very elegant or practical. We believe that, in many cases, the need for a specific language could be avoided with the use of selectors.

An example is the Doxpects language, proposed in [24]. It is a DSL for processing XML documents in Web Services messages. It includes a pointcut language for selecting specific elements inside messages that are exchanged through SOAP communication. This way, the user does not have to worry about where in the base program the advice should be bound, and how to extract the desired elements from the message.

The language includes two new pointcut designators[2], *header* and *body*, that match the header and the body of a SOAP message. These pointcuts have as arguments XPath queries, which select specific elements inside SOAP messages.

Additionally, the language also defines two new qualifiers for advice, *request* and *response*. These indicate whether the message to be matched is part of a request or a response message. We could also think of these qualifiers as part of the pointcuts, because they help in the filtering of the join point shadows where the advice will be inserted.

In figure 10, we outline a similar solution for SOAP request messages by developing a specific selector. First, at weave time, this selector chooses the appropriate point to insert the run-time checks by returning `CHECK_AT_RUNTIME` when the correct join point shadow is evaluated. This could also be done with a conventional pointcut expression, but the selector hides the details about the framework from the user. In the run-time part of the selector, it uses the XPath expression that was given as an argument to perform the match.

While this solution does not offer all the advantages of the original Doxpects proposal, it fulfills its most important goal, which is creating semantic pointcuts for the processing of SOAP messages. Other features, like the conversion of document elements to Java objects of specific types, could be added in the future (see section 7). The advantage of our solution over the DSL one is that it does not require the user to adopt a specific language.

### 5.5 Frameworks

Many frameworks use meta information to add semantics to a program. In some situations, it may be desirable to use this meta information as a selection criteria to apply aspects to elements of a program. In conventional aspect-oriented languages, the programmer is left with the task of identifying ways of reading and using this meta information. Depending on how the framework stores it, such as annotations or XML, this task can be easy or very hard to ac-

[2] In the terminology that we introduce in this paper, these constructs could be better named selectors, but we kept the author's original terminology in this section.

```
@SelectorDef(name="request")
public class WsRequestSelector extends SelectorHelper {

 /** Weave-time method execution selector */
 public SelectionValue matchesCall(Advisor callingAdvisor,
   MethodCall methodCall, List<SelectorParam> params) {
  if (isWsRequestMethod(methodCall)) {
   return CHECK_AT_RUNTIME;
  } else {
   return FALSE;
  }
 }

 /** Run-time method execution selector */
 public boolean matchesCall(Advisor advisor,
   AccessibleObject within, Class calledClass,
   Method calledMethod, Object target, Object[] args,
   List<SelectorParam> selectorParams) {
  // Gets selector parameters
  String xpathExpression = selectorParams.get(0).getValue();
  // Gets the XML Document
  Document docroot = getWsDocument(target, args);
  // Matches the document to the desired elements, given by
  // the XPath query in the selector parameters
  XPath xpath = XPathFactory.newInstance().newXPath();
  try {
   NodeSet resultNodes = (NodeSet) xpath.evaluate(
    xpathExpression, docroot, XPathConstants.NODESET);
   if (resultNodes != null && resultNodes.getLength() > 0) {
    return true;
   } else {
    return false;
   }
  } catch (XPathExpressionException e) {
   return false;
  }
 }
 /**
  * Determines if the given method call is the appropriate
  * point to insert a run-time check for WS XML request
  * documents. It does so with knowledge of the WS framework
  * and API.
  */
 private boolean isWsRequestMethod(MethodCall methodCall) {
  ...
 }
 /**
  * Gets the DOM document for the Web Services request.
  */
 private Document getWsDocument(Object target,
   Object[] args) {
  ...
 }
}
```

**Listing 10.** A selector for XML elements in Web Services requests

complish. Either way, though, the users will have to create specific code to deal with the framework's meta information representation, which they do not control. Such a scenario leads to fragile pointcuts that may break the modularity of a system, instead of improving it.

This problem could be solved by framework-specific selectors. These selectors could be created as parts of the frameworks, to be used by the programmers. As an example, consider Hibernate [1], a popular framework for object/relational mapping. Figure 11 presents a selector for Hibernate property setters. The selector uses the framework itself to load the meta information. Thus, the modularity is not broken, and the programmer working with the framework gains a high-level mechanism to deal with the framework abstractions.

To illustrate how this selector can improve pointcut quality, we can compare some attempts for creating a pointcut that picks setters of properties of type `java.util.Date`.

The first example, shown in listing 12, uses a naming pattern to match the setters. This pointcut works under two assumptions: (1)

```
@SelectorDef(name = "hibernatePropertySetter")
public class HibernatePersistentPropertySetterSelector
  extends SelectorHelper {

 /** Weave-time method execution selector */
 public SelectionValue matchesExecution(Advisor advisor,
   CtMethod m, List<SelectorParam> params) {
  SelectionValue result = FALSE;
  // Gets the method's class
  Class declaringClass = m.getDeclaringClass().toClass();
  // Gets a Hibernate Session Factory
  SessionFactory sessionFactory =
   new Configuration().configure().buildSessionFactory();
  // Gets the class's persistent properties an iterates
  // over them
  ClassMetadata cmd =
   sessionFactory.getClassMetadata(declaringClass);
  String[] persistentProperties = cmd.getPropertyNames();
  for (String prop : persistentProperties) {
   // Gets the JavaBeans method used to set the property
   PropertyDescriptor pd = new PropertyDescriptor(prop,
     declaringClass);
   Method writeMethod = pd.getWriteMethod();
   // If the methods are the same, we found a match
   if (writeMethod.equals(m.getName())) {
    result = TRUE;
    break;
   }
  }
  return result;
 }
}
```

**Listing 11.** A selector for Hibernate property setters

that all classes under the package `com.acme.someapp` are mapped for persistence, and (2) that all methods starting with "set" in those classes are setters for persistent properties. If any of these assumptions fails, the pointcut will fail. For example, if we have a caching system for the persistent objects that uses a `Date` field to store the time of its last update, the setter for this field would be incorrectly selected by this pointcut.

Another drawback is that a programmer looking at this pointcut will not immediately know that its intent is to capture setters for persistent properties. A comment would have to be added for that to become clear. Therefore, this pointcut has low quality: it is not resistant to change and does not communicate its intent clearly.

```
@Bind(pointcut =
 "execution(void com.acme.someapp.*->set*(java.util.Date))")
```

**Listing 12.** A pointcut based on naming conventions

In the second example, shown in figure 13, we use the specific `hibernatePropertySetter` selector to choose only the methods that are setters for persistent properties. We still have a clause to filter execution of methods in a specific package and with specific parameter and return types, but we do not rely on a naming convention anymore. This pointcut would not match the setter for the last update time field cited previously. Additionally, it is clear to the programmer that we are picking only methods that are setters for Hibernate properties. Therefore, we have enhanced the pointcut quality considerably. Another advantage of this approach is that the access to framework-specific meta information is encapsulated inside the framework classes that are used by the selector. This way, artifact boundaries are crossed in a way that is transparent to the aspect programmer.

A point that must be stressed is that selectors like the one presented here would better be provided as parts of the framework on which they rely. Such an arrangement preserves the modularity of the framework and shields the application/aspect programmer from framework-specific details.

```
@Bind(pointcut =
 "execution(void com.acme.domain.*->*(java.util.Date)) " +
 "AND hibernatePropertySetter()")
```

**Listing 13.** A pointcut based on the hibernatePropertySetter specific selector

## 6. Related Work

Several approaches have been proposed for improving the expressiveness of pointcut languages. Some authors have proposed the use of logic languages as a basis for pointcut languages. In [6], a new Aspect-Oriented language, called Andrew, is proposed. It uses a logic language, similar to Prolog, for the definition of pointcuts. The base language over which the aspects are applied is Prolog, and this language's meta-information facilities are used as a basis for the join point model. In [7], the authors explain what features of their language make it a good fit for defining pointcuts.

In [14], the authors propose the AO language Gamma, which is based on a simplified version of Java, for the base program, and on Prolog for pointcut definition. The main focus of this approach is on dynamic pointcuts. It uses a join point model that is based on a trace of the program execution, with timestamps associated with each point of the execution. This allows for very easy definition of pointcuts that depend on the order of events, like `cflow`. However, this approach has serious limitations for practical use, and the authors regard the overcoming of these as future work. Alpha [19] is a logic-based, language related to Gamma, that uses a less elegant model, but is more tractable in practice. It uses four sources of information: a representation of the program's abstract syntax tree, a representation of its heap, the static type of every expression in the program, and a representation of the program execution trace.

In our view, logic languages are good for expressing the types of pointcuts that are most commonly used today: the ones that use only the basic join point model. However, they would be very hard to use in situations like the one we presented in section 5.5, when other sources of data are necessary besides the basic join point model. By using an imperative language, preferably the same in which the base program is written, users can take advantage of practically any data source they need.

In [5], the authors propose the use of the functional language XQuery as a replacement for current pointcut languages. They use an XML representation of Java bytecode as a base over which to run the queries. They consider that their approach allows for more flexible pointcut definitions, and also that their definitions are clearer. The main shortcoming of their approach is that it only has weaving-time information available. It also requires building the XML bytecode representation, which is an additional step to the weaving process that could create complications. Also, XQuery may not be the most suitable language for defining complex pointcuts, and it requires the programmer to learn another language.

Josh [4] has a lot in common with our approach. It proposes an extension mechanism that is based on the same language as the base program, just as ours. It also uses the Javassist bytecode manipulation framework to obtain weaving-time information about the program. The main difference is that it does not deal with run-time information. If a run-time check is necessary, it must be explicitly inserted into the program through the bytecode manipulation framework. Such a task, not needed in our approach, can be difficult and error-prone.

The AOP part of the Spring framework [10] defines all of its pointcuts through Java classes. It has a mechanism for the combination of weaving-time and run-time checks that is very similar

to ours. However, it does not provide a language to easily combine pointcuts, relying instead on a set of verbose XML definitions. That is a shortcoming that we think is better addressed in our work. Additionally, it does not provide access to a structural reflection framework for join point selection, relying exclusively on the standard Java API. That makes it difficult to use Spring AOP for implementing more powerful selectors, such as the one in section 5.2.

## 7. Conclusion and Future Work

Current pointcut languages are limited in mechanisms to allow the construction of new abstractions. Most of them consist of a fixed set of building blocks that can only be combined by boolean operators. By opening to the programmer the possibility of creating new building blocks with the full power of a general-purpose programming language, higher levels of abstraction can be achieved without sacrificing the simplicity of pointcut creation.

Our work shows that extension mechanisms for pointcut languages can increase pointcut quality. It also enables the creation of new types of pointcuts that were not previously possible, such as those that depend on meta information that is outside of the source code itself. In particular, it lets framework implementors define framework-specific selectors, which allow aspects to cross the boundary of a given framework and its different artifacts while still respecting the modularity of that framework.

Our current prototype still has some limitations, which are not technically difficult to remove. Allowing only `String`s as selector arguments is the most notable of these limitations, whose removal we regard as future work.

Another feature that will make join point selectors even more useful is the possibility of aggregating meta information and making such information available to advice implementers. That would be especially useful in selectors that take advantage of external meta information. This extension would also make it possible to implement a feature of the Doxpects DSL that the example in section 5.4 does not provide: the transformation of XML elements into Java objects, which are made available to advice programmers.

Finally, the combination of selectors could be easier if we made the following changes to their semantics:

- Instead of receiving one join point as an argument, a selector would receive a set of them;
- Instead of returning a boolean, it would return a subset of the set that was received as an argument.

These slight changes would make it possible to use the result of one selector as an argument to another one. For example, the `updatesStateReadBy` selector of section 5.2 could be divided in two parts: one would select all fields read by a given method and the other would select all methods that update any of a given set of fields. That way, both parts could be reused independently.

In order to implement this approach, however, the architecture of the weaver would have to be considerably changed, bringing new performance and functionality challenges. Further research is needed to determine if these challenges can be satisfactorily overcome.

## References


[1] Christian Bauer and Gavin King. *Hibernate in Action*. Manning, 2005.

[2] Johan Brichau and Michael Haupt. Survey of aspect-oriented languages and execution models. Technical Report AOSD-Europe-VUB-01, AOSD-Europe, May 2005.

[3] Shigeru Chiba. Load-time structural reflection in java. In Elisa Bertino, editor, *ECOOP*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, 2000.

[4] Shigeru Chiba and Kiyoshi Nakagawa. Josh: an open AspectJ-like language. In Lieberherr [17], pages 102–111.

[5] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2004.

[6] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In Pascal Costanza, Günter Kniesel, Katharina Mehner, Elke Pulvermüller, and Andreas Speck, editors, *Second Workshop on Aspect-Oriented Software Development of the German Information Society*. Institut für Informatik III, Universität Bonn, February 2002. Technical report IAI-TR-2002-1.

[7] Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In Mehmet Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 60–69. ACM Press, March 2003.

[8] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Lieberherr [17], pages 26–35.

[9] JBoss Inc. *JBoss AOP Reference Documentation*.

[10] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, and Erwin Vervaet. *Spring - Java/J2EE Application Framework Reference Documentation*.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[12] Gregor Kiczales. The fun has just begun. keynote. In *AOSD 2003, Boston*, March 2003.

[13] Mik Kirsten. Aop@work: Aop tools comparison, part 1: Language mechanisms. Technical report, IBM Developer Works, February 2005.

[14] Karl Klose and Klaus Ostermann. Back to the future: Pointcuts as predicates over traces. In Leavens et al. [16].

[15] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[16] Gary T. Leavens, Curtis Clifton, and Ralf Lämmel, editors. *Foundations of Aspect-Oriented Languages*, March 2005.

[17] Karl Lieberherr, editor. *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, March 2004.

[18] Karl J. Lieberherr, Jeffrey Palm, and Ravi Sundaram. Expressiveness and complexity of crosscut languages. In Leavens et al. [16].

[19] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer, 2005.

[20] Apache Struts Project. *http://struts.apache.org/*.

[21] AspectWerkz Project. *http://aspectwerkz.codehaus.org/*.

[22] Dominik Stein, Stefan Hanenberg, and Rainer Unland. An UML-based aspect-oriented design notation. In Gregor Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 106–112. ACM Press, April 2002.

[23] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

[24] Eric Wohlstadter and Kris De Volder. Doxpects: aspects supporting xml transformation interfaces. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 99–108, New York, NY, USA, 2006. ACM Press.