JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 7, No. 9, Special Issue: SPLAT, December 2008

User-Defined Join Point Selectors

An Extension Mechanism for Pointcut Languages

Cristiano Breuel, CS Department, University of São Paulo, Brazil
Francisco Reverbel, CS Department, University of São Paulo, Brazil

One of the main issues in contemporary AOP languages and frameworks is the expressiveness of the pointcut language. This paper proposes an extension mechanism for enriching pointcut languages with constructs that play the role of "new primitive pointcuts" and allow the creation of pointcuts with greater semantic value. *Join point selectors* are a generalization of the primitive pointcuts of current pointcut languages. Existing languages, however, do not allow users to create new join point selectors. We present a simple architecture for supporting user-defined join-point selectors as an extension mechanism implemented atop an existing AOP framework. We show examples of user-defined selectors that enhance the quality of pointcuts and make aspect development easier. Moreover, we show that our extension mechanism supports framework-specific selectors, which let aspects cross the boundary of a given framework while still respecting the modularity of that framework.

1 INTRODUCTION

As programming languages and paradigms evolve, they tend to provide better support for modularity and to let programmers work at higher levels of abstraction. By affording the separation of crosscutting concerns, *aspect-oriented programming* (AOP) increases modularity. Nevertheless, programmers still need to deal with lower-level concepts in order to specify the so-called *pointcuts* — the sets of points at which those concerns crosscut the basic functionality of a program. This work aims at raising the level of abstraction of pointcut specifications. Our proposal allows programmers to define pointcuts in terms of higher-level concepts. It also improves the resiliency of pointcut definitions against changes in the base program. The present text is an updated and extended version of our previous paper [5].

Motivation

The AspectJ language [15] introduced a model for AOP that has been widely accepted and adopted by other aspect-oriented languages and frameworks, including open-source projects such as JBoss AOP [12], Spring AOP [13], and AspectWerkz [2]. In spite of differences in syntax and in implementation approach, all such tools have similar capabilities and semantics [14] and offer similar pointcut languages.

Cite this article as follows: Cristiano Breuel, Francisco Reverbel: *User-Defined Join Point Selectors*, in Journal of Object Technology, vol. 7, no. 9, Special Issue: SPLAT, December 2008, pages 5–24,
http://www.jot.fm/issues/issues_2008_12/article1

Two significant limitations have been identified in current pointcut languages. The most frequent concern is that a pointcut may be “broken” by changes to the base program [10, 7, 19]. This limitation is known as the *fragile pointcut problem* [18]. Another issue is the difficulty or impossibility of expressing some pointcuts clearly and accurately [16].

Pointcut Quality

It is useful to set forth a criterion 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 point should be included in the set selected by the pointcut if and only if it matches the conditions intended by the pointcut author.
- **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. In other words, it should be easy to understand and to modify a pointcut definition.

Example. One of the most frequent examples of AspectJ usage, the figure editor [15], has also been commonly used for exposing the shortcomings of that language [7, 8, 17]. It consists of a graphical editor, with several types of elements (squares, circles etc.), whose display must be updated whenever the state of some element 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 in response to element modifications.

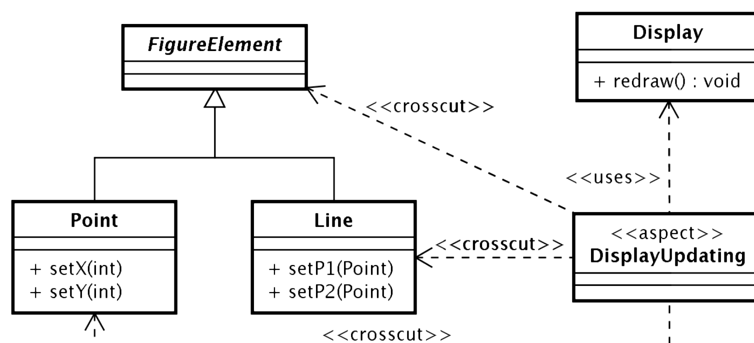


Figure 1: UML diagram for the figure editor example

The classic solution is an aspect that selects methods based on a naming convention, e.g. picking all methods whose names start with “set” defined in class `FigureElement` and its subclasses. A pointcut based on such a naming convention

is a low-quality one, because it is not quite resilient (if someone implements a new method that alters some element but does not start with "set", the new method will not be selected) and does not clearly express its intent (which the reader must guess from the method prefix).

Another solution is to create an annotation that must be associated with updater methods, for example, `@FigureUpdater`. This solution improves the clarity of the pointcut, but still does not meet the resilience requirement (one can forget to use the annotation or accidentally remove it). 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 would state exactly what we intend to capture, and second, because it would be fully tolerant to changes in the base program. This kind of pointcut is what we wish to support.

Problem Statement

Our goal is to provide programmers with the means for defining pointcuts that have high quality, according to the definition in section 1. We want to allow pointcuts that are defined at a higher level of abstraction (closer to the problem at hand) and have greater semantic value (in the sense that they reflect their authors' intents in a more precise way).

Proposed Solution

Since the expressiveness of the pointcut language limits the ability of creating high quality pointcuts, an enrichment to pointcut languages is needed. We propose user-defined *join point selectors* as a simple extension mechanism for enhancing pointcut languages with constructs that play the role of "new primitive pointcuts".

Contributions of this Work

The major contributions of this work are the concept of user-definable join point selector (section 2), a prototype implementation of that concept on an existing AOP framework (section 3), and a set of examples of selector usage (section 4). These examples show that user-defined join point selectors allow aspect programmers to create high-quality pointcuts that were not previously possible. The paper also contains a discussion of related work (section 5) and our concluding remarks and future work ideas (section 6).

2 JOIN POINT SELECTORS

A *join point selector* is a function that, for a given set of arguments and a join point, determines whether the join point should be part of a pointcut. Selectors are

intended to be used as boolean elements of pointcut expressions. Their evaluation starts at weave time and may proceed at run time. Accordingly, a join point selector has a *weave-time part*, which is activated by the weaver, and a *run-time part*, which may or may not act (at the discretion of the weave-time part) when the execution of the aspectized program reaches every candidate join point. Figure 2 shows the operation of a selector. If the weave-time part is unable to reach either a positive decision (the join point should be included in the pointcut) or a negative one (the join point should be excluded from the pointcut), then it specifies that the run-time part should be activated at the appropriate occasions. In other words, the run-time part acts if the information available at weave time is not enough to complete the decision.

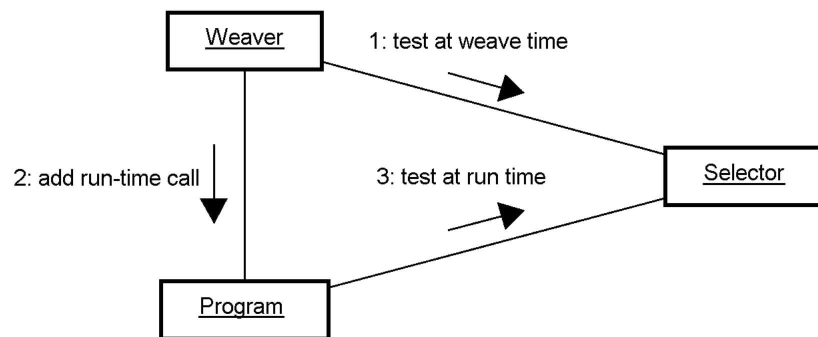


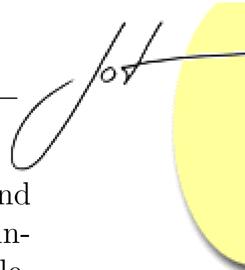
Figure 2: The operation of a selector

In current aspect-oriented languages and frameworks, the concept of selector is represented by the so-called *primitive pointcuts*: "`call`", "`execution`", etc. Nevertheless, the programmer cannot define new selectors, as the algorithms that select join points are hard coded into the weaver. We consider that our proposed naming is important because it makes clear the distinction between a selection algorithm and its usage in pointcut expressions. The examples below illustrate that distinction:

- `call` and `within` are selectors.
- `call(void *->setSize(..)) AND within(com.acme.*)` is a pointcut expression.

In a pointcut language, join point selectors play a role similar to the one of procedures in a procedural language, methods in an object-oriented language and advice in an aspect-oriented language. The core characteristics that define join point selectors, and distinguish them from other extension 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 form composite expressions.** Boolean operators can be used to combine multiple selector occurrences within a pointcut expression.



3. **They operate both at weave time and at run time.** A simple and uniform scheme allows selectors to use weave-time information, run-time information, or both, and to complete the selection decision as early as possible.

These features make selectors a basic unit of functionality. Some of them are found in previous proposals (such as the proposals that we will discuss in section 5), but the combination of all three features makes join point selectors more expressive and easy to use.

3 AN IMPLEMENTATION OF USER-DEFINABLE SELECTORS

As a proof of concept, we have implemented support to user-defined join point selectors as an extension to the JBoss AOP [12] framework. The choice of JBoss AOP as a basis for the implementation was due to practical factors only. Since we had no conceptual reason to employ that specific framework, our approach is not limited to JBoss AOP and could be implemented in other aspect-oriented languages or frameworks that have similar concepts.

Selector Programmer's View

Users of join point selectors (pointcut programmers) view selectors as boolean functions that are applied to candidate join points and take such a join point as an implicit argument. From the viewpoint of a selector programmer, however, the selector is a Java class that implements the interface `org.jboss.aop.selector.Selector` shown in Listing 1. We defined that interface after an existing JBoss AOP interface: `org.jboss.aop.pointcut.Pointcut`, implemented by objects that internally represent pointcuts in that framework. Thanks to the similarity between both interfaces, we were able to consistently integrate selector functionality into the framework without changing a lot of JBoss AOP code.

There are two groups of methods in the `Selector` interface. Each of these groups has a method for every kind of primitive join point supported by the framework[1]: method calls, attribute reads and writes, etc. Therefore, a single selector can be applied to different kinds of join points. One may create a selector that picks heterogeneous join points, in case such a selector is needed.

Weave-time and run-time selector methods. The first group of methods is called by the weaver and corresponds to the weave-time part of the selector. Each of those methods performs selector evaluation at weave time, for a specific kind of join point. It returns a value of the enumeration type `SelectionValue`, which defines three constants: `TRUE` (the join point matches the selector criteria and should be

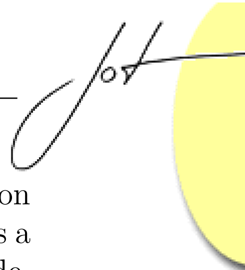
[1] Even though the `Selector` interface has methods for all kinds of join points, the vast majority of selectors implements just a few of these methods (typically one or two) in a non-trivial way. There are pros and cons to this approach, but its main motivation was coherence with the internal structure of JBoss AOP.

```
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 matchesCall(Advisor callingAdv, MethodCall mc,
                               List<SelectorParam> selectorParams);
    SelectionValue matchesCall(Advisor callingAdv, NewExpr mc,
                               List<SelectorParam> selectorParams);
    SelectionValue matchesGet(Advisor adv, CtField f,
                              List<SelectorParam> selectorParams);
    SelectionValue matchesSet(Advisor adv, CtField f,
                              List<SelectorParam> selectorParams);
    // Run-time selector methods:
    boolean matchesExecution(Advisor adv, Method m,
                             Object target, Object[] args,
                             List<SelectorParam> selectorParams);
    boolean matchesExecution(Advisor adv, Constructor c,
                             Object target, Object[] args,
                             List<SelectorParam> selectorParams);
    boolean matchesConstruction(Advisor adv, Constructor c,
                                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);
    boolean matchesGet(Advisor adv, Field f, Object target,
                       List<SelectorParam> selectorParams);
    boolean matchesSet(Advisor adv, Field f, Object target, Object value,
                       List<SelectorParam> selectorParams);
}
```

Listing 1: The Selector Interface

included in the pointcut), FALSE (the join point does not match the selector criteria and should be excluded from the pointcut) and CHECK_AT_RUNTIME (a decision is not possible without run-time information). If a weave-time method returns this last constant, the weaver instruments the join point shadow [11] by inserting into the base code a call to the corresponding run-time method.

The second group of methods is the run-time part of the selector. If a call to a method of the first group determines that the selection decision can only be reached with run-time information, then the corresponding method of the second group will be called at run-time. The run-time calls happen as a result of the code instrumentation performed at weave time.



All selector methods receive a set of parameters that represent the join point upon which the selector is being evaluated. In the weave-time methods, that set contains a single parameter, which reifies the corresponding joint point shadow in the base code. The types of the weave-time parameters that reify join point shadows (`CtMethod`, `CtConstructor`, `MethodCall`, `NewExpr`, and `CtField`) are part of the Javassist API. Javassist [6] is a framework for structural reflection that reads and manipulates Java bytecode, but provides a high-level API that allows programmers to deal with elements of the Java language rather than with bytecode details. Javassist is the basis for all the bytecode manipulation in JBoss AOP. Through its API, selector developers have access to the program structure in a way that is much more effective than standard Java reflection. The power of Javassist makes it possible to create selectors with weave-time methods that examine the internal structure of constructs such as classes or methods. An example of such a selector will be presented in section 4.

Run-time selector methods do not see Javassist types. In those methods, the set of parameters that represent the current join point comprises an instance of a class in `java.lang.reflect` (a `Method`, `Constructor`, or `Field`) that reifies the corresponding join point shadow in the base code, plus additional objects associated with the joint point at run-time (e.g., the target and the arguments of a method call).

Base class for selectors. To avoid the need for implementing all methods of the `Selector` interface when only some of them will be actually used, the convenience class `SelectorBase` implements that interface in a default (and trivial) way. Each selector method has in `SelectorBase` a default implementation that simply returns a constant value, which is either `SelectionValue.FALSE` (in the case of a weave-time method) or `false` (in the case of a run-time method). Selector programmers will typically create classes derived from `SelectorBase`, instead of writing classes that directly implement the `Selector` interface.

Selector declarations. Selectors must be declared in order to be recognized by the weaver. A selector declaration consists of metadata and takes one of the following forms: (i) an annotation in the selector class itself, or (ii) an XML element in a descriptor file. JBoss AOP already supported both declaration styles for all of its features, so we followed the same design choice. Listings 2 and 3 exemplify the two declaration styles by showing alternative declarations for the same selector.

```
@SelectorDef(name="parameterTypeIs")
public class ParameterTypeSelector extends SelectorBase {
    ...
}
```

Listing 2: A selector declaration through a Java annotation

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

Listing 3: A selector declaration through an XML element

In both declaration styles there is a `name` attribute, which specifies the *selector name* to be used in pointcut expressions. The XML version also has a `class` attribute, which specifies the fully qualified name of the selector class.

Pointcut Programmer's View

Selectors are used as boolean clauses in pointcut expressions. A selector clause has the same syntax as a method call: the selector name, followed by a comma-separated list of arguments, which is enclosed by parentheses. Selector clauses may be combined via boolean operators.

Listings 4 and 5 exemplify the usage of the selector named `parameterTypeIs` declared in the previous section. Listing 4 shows a pointcut expression defined in a Java annotation; Listing 5 shows an XML element that defines the same expression. These examples intend to select all executions of methods in class `MyPOJO` (`execution` clause) whose first parameter is an `Integer` (clause that starts with `parameterTypeIs`).

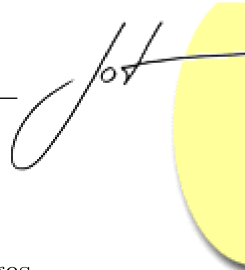
```
@Bind(pointcut="execution(org.jboss.test.aop.selector.MyPOJO->*(..)) AND " +
               "parameterTypeIs(\"0\", \"java.lang.Integer\")")
public Object advice(Invocation invocation) throws Throwable {
    ...
}
```

Listing 4: Selector clause in a pointcut expression defined by a Java annotation

```
<bind pointcut="execution(org.jboss.test.aop.selector.MyPOJO->*(..)) AND
               "parameterTypeIs(&quot;0&quot;, &quot;java.lang.Integer&quot;)">
    <advice name="advice" aspect="TestAspect"/>
</bind>
```

Listing 5: Selector clause in a pointcut expression defined by an XML element

In our current prototype, the arguments of selector clauses must be strings. We plan to support other argument types in a future implementation. Note that the arguments of a selector clause appear within an attribute of an annotation or XML element. Since both the attribute and the selector argument (a string) are enclosed by quotes, the inner quotes must be escaped and thus they appear as “`\"`” within the annotation attribute and as “`"`” within the XML attribute.



Internal View

Our extension to JBoss AOP consists of the following modifications: (*i*) changes in the pointcut grammar of JBoss AOP to recognize selectors within pointcut expressions, (*ii*) the addition of new elements to the XML and annotation bindings to allow for selector declarations, and (*iii*) changes in the JBoss AOP weaver to add calls to run-time selectors where necessary.

4 EXAMPLES OF USER-DEFINED SELECTORS

This section presents examples that demonstrate how our proposal can improve pointcut quality. These examples are meant as a sample of the anticipated uses of join point selectors. Due to the open nature of selectors, it is not possible to anticipate all of their practical applications.

Parameter Type

This example was borrowed from [7], with a slight generalization. It consists of a selector that picks executions of methods with a parameter of a specified type in a specified position. (The pointcut examples of section 3 used such a selector to pick all methods whose first parameter is an `Integer`.) The purpose of this example is to show a very simple use of selectors to solve a problem that is not motivated by a real-world application, but nevertheless does not have an elegant answer in conventional aspect-oriented languages. We consider our solution simpler and more elegant than the one in [7], thanks to the clear separation between the static (weave-time) and dynamic (run-time) parts of the selector.

Declarations for the selector `parameterTypeIs` appeared in Listings 2 and 3. Listing 6 shows the selector class, which is derived from `SelectorBase` and implements two selector methods: the weave-time matcher that picks method executions and the corresponding run-time matcher.

The weave-time matcher looks at the declared type of the parameter at the specified position. If the declared parameter type is the same as the desired type or is a subtype of the desired type, it returns "join point matched". Otherwise, if the declared parameter type is a supertype of the desired type (for example, if the declared type is `Object` when we want an `Integer`), it defers the decision to run time. In all other cases, it returns "join point not matched". The run-time matcher is even simpler. It looks (via standard Java reflection) at the type of the actual parameter passed to the method at run time and checks if that type is compatible with the desired one. Note that there is no run-time penalty unless a dynamic check is truly needed.

Recall our comment on the possibility of creating a selector that picks heterogeneous join points. The selector `parameterTypeIs` could be easily modified to pick

```
@SelectorDef
public class ParameterTypeSelector extends SelectorBase {

    // Weave-time matcher for method executions
    public SelectionValue matchesExecution(Advisor advisor, CtMethod m,
                                           List<SelectorParam> params) {
        // Gets selector parameters
        int paramIndex = Integer.parseInt(params.get(0).getValue());
        String wantedTypeName = params.get(1).getValue();
        // Obtains CtClass for the declared type of the method parameter
        CtClass paramType = m.getParameterTypes()[paramIndex];
        // Obtains CtClass for the desired type
        CtClass wantedType = ClassPool.getDefault().get(wantedTypeName);
        // Tests for compatibility between types
        if (paramType.subtypeOf(wantedType))
            return TRUE;
        else if (wantedType.subtypeOf(paramType))
            return CHECK_AT_RUNTIME;
        else
            return FALSE;
    }

    // Run-time matcher for method executions
    public boolean matchesExecution(Advisor advisor, Method m,
                                    Object target, Object[] args,
                                    List<SelectorParam> params) {
        // Gets selector parameters
        int paramIndex = Integer.parseInt(params.get(0).getValue());
        String wantedTypeName = params.get(1).getValue();
        // Obtains Class object for the desired type
        Class wantedType = Class.forName(wantedTypeName);
        // Gets the actual parameter passed to the method
        Object param = args[paramIndex];
        // Tests for run-time compatibility between types
        if (param == null)
            return true; // null matches any type
        else {
            Class paramType = param.getClass();
            return wantedType.isAssignableFrom(paramType);
        }
    }
}
```

Listing 6: A selector to pick methods with a parameter that is compatible with a specified type

not only method executions, but also method calls, constructor executions, and constructor calls, whenever a method/constructor receives a parameter of a given type in a given position. The enhanced selector would simply have additional weave-time/run-time matcher pairs, which would be implemented similarly to the pair of matcher methods in Listing 6.

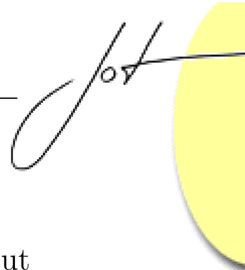


Figure Editor

Let us revisit the figure editor scenario and create a selector that solves the pointcut quality problem identified in section 1. The new selector takes two parameters (a class name and a method name) that specify a *reader method*[2]. It picks all the executions of methods that can possibly update some field read by the reader method. Given such selector, a high-quality pointcut for the figure editor would simply use the following selector clause:

```
updatesStateReadBy("Display", "redraw")
```

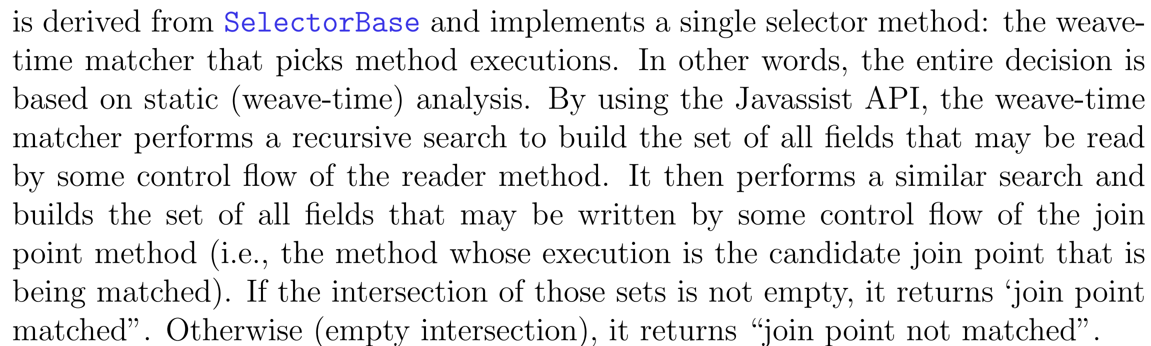
With the new selector, the `DisplayUpdating` aspect (Figure 1) runs only after the executions of methods that alter fields read by the `Display.redraw()` method. Pointcut quality is high, because the pointcut is fully resilient to code changes and its intent is more clear to the reader than a naming pattern.

Listing 7 shows an skeletal implementation of the new selector. The selector class

```
@SelectorDef(name="updatesStateReadBy")
public class StateUpdaterSelector extends SelectorBase {

    // Weave-time matcher for method executions
    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 the sets
        return readFields.removeAll(updatedFields) ? TRUE : FALSE;
    }

    // Recursively finds (possibly a superset of) the set of all fields
    // updated by the method m, including those updated within calls
    // to other methods.
    private Set<CtField> getFieldsUpdatedByMethod(CtMethod m) { ... }

     // Recursively finds (possibly a superset of) the set of all fields read
     // by the method m, including those read within calls to other methods.
    private Set<CtField> getFieldsReadByMethod(CtMethod m) { ... }
}
```

Listing 7: A selector to pick join points that update fields read by a given method

[2]For simplicity, we are not dealing with overloaded methods.



is derived from SelectorBase and implements a single selector method: the weave-time matcher that picks method executions. In other words, the entire decision is based on static (weave-time) analysis. By using the Javassist API, the weave-time matcher performs a recursive search to build the set of all fields that may be read by some control flow of the reader method. It then performs a similar search and builds the set of all fields that may be written by some control flow of the join point method (i.e., the method whose execution is the candidate join point that is being matched). If the intersection of those sets is not empty, it returns 'join point matched". Otherwise (empty intersection), it returns "join point not matched".

Even though the selector performs static analysis only, its implementation is not simple. Both recursive searches take into account the subclasses of any types that may be reached by some sequence of nested calls from the root method (the reader method or the join point method). This is a conservative approach that presumes the worst-case scenario with respect to conditional statements and inheritance. It ensures that no correct match will be left out, but may also produce false matches.

Reflective Calls

Method calls performed via reflective techniques are becoming more and more used by infrastructural software such as frameworks and middleware, which often needs to invoke methods on application objects but has no prior knowledge of the interfaces of those objects. Conventional aspect-oriented languages do not currently offer a simple way of adding caller-side advice to reflective calls. Caller-side advice is important in scenarios that disallow the instrumentation of callee code.

JBoss AOP even provides a helper aspect for intercepting reflective calls to a specified method. That aspect actually intercepts all reflective calls and uses advice code to select just the calls to the specified method. Nevertheless, we think that a solution entirely based on pointcuts would be more appropriate for the following reasons: (*i*) the interception of reflective calls should be performed similarly to the interception of plain (non-reflective) calls, which is based on pointcuts, (*ii*) an additional aspect just to pick reflective calls is unwarranted complexity, and (*iii*) pointcuts are meant for join point selection, so using advice for that purpose is a departure from the intended usage of these AOP constructs.

We propose a specific selector, reflectiveCall, which takes two parameters (a class name and a method name) that specify a method[3] and picks the reflective calls to that method. Listing 8 shows the implementation of the reflectiveCall selector. Again, the selector class is derived from SelectorBase and implements two selector methods: the weave-time matcher that picks method calls and the corresponding run-time matcher. When a call is performed through reflection, the called method can be determined only at run time. Thus the weave-time matcher simply flags every call to the invoke() method of java.lang.reflect.Method as requiring a run-time check. For any other method calls, it returns "join point not matched". The run-

[3]For simplicity, we again disregard the case of overloaded methods.

```
@SelectorDef(name="reflectiveCall")
public class ReflectiveCallSelector extends SelectorBase {

    // Holds a reference to the CtMethod object for Method.invoke()
    private CtMethod invoke;

    public ReflectiveCallSelector() {
        invoke = ClassPool.getDefault().get("java.lang.reflect.Method")
                                       .getDeclaredMethod("invoke");
    }

    // Weave-time matcher for method calls
    public SelectionValue matchesCall(Advisor callingAdvisor,
                                      MethodCall methodCall,
                                      List<SelectorParam> params) {
        if (methodCall.getMethod().equals(invoke))
            return CHECK_AT_RUNTIME;
        else
            return FALSE;
    }

    // Run-time matcher for method calls
    public boolean matchesCall(Advisor advisor, AccessibleObject within,
                               Class calledClass, Method calledMethod,
                               Object target, Object[] args,
                               List<SelectorParam> selectorParams) {
        String methodClassName = selectorParams.get(0).getValue();
        String methodName = selectorParams.get(1).getValue();
        Method targetMethod = (Method) target;
        String targetMethodClassName =
            targetMethod.getDeclaringClass.getName();
        String targetMethodName = targetMethod.getName();
        return (targetMethodClassName.equals(methodClassName) &&
                targetMethodName.equals(methodName));
    }
}
```

Listing 8: A selector for reflective method calls

time matcher looks at the `invoke()` target, which is a `java.lang.reflect.Method`, and checks if this object represents the method specified by the selector parameters.

Selector for Web Services

Various domain-specific languages (DSLs) have been proposed to tailor aspect-oriented programming to specific problem domains. One such language is Doxpects [20], a DSL for processing XML documents in messages exchanged via SOAP. Doxpects allows the definition of pointcuts that specify XML elements within SOAP messages. Moreover, it exposes those elements to advice so that the elements can be processed in a convenient way. Essentially, the language introduces two new kinds

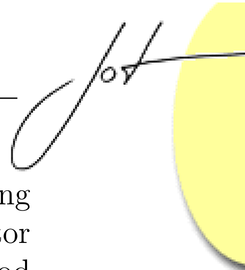
of pointcuts: header and body, which match the header and the body of a SOAP message. A header or body pointcut takes as argument an XPath query that selects a set of elements from the corresponding section (head or body) of the message.

In addition to the new pointcuts, the language also introduces two qualifiers for advice: request and response, which indicate if pointcut (header or body) evaluation will be performed on a request message or on a response message. The authors of Doxpects make a parallel between those qualifiers and AspectJ's before and after modifiers. Even so, a request or response qualifier can be regarded as a part of the pointcut definition, because it helps to determine the join point (the processing of a message, which is either an outgoing request or an incoming response) that will be affected by the advice.

We believe that specific selectors can be used as a substitute for a DSL in some scenarios. A selector-based solution for the processing of SOAP messages would include a selector for SOAP requests and another for SOAP responses. Listing 9 shows a skeletal implementation of a simple selector for SOAP requests. This selector

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

    // Weave-time matcher for method calls
    public SelectionValue matchesCall(Advisor callingAdvisor,
                                      MethodCall methodCall,
                                      List<SelectorParam> params) {
        return (isWsRequestMethod(methodCall)) ? CHECK_AT_RUNTIME : FALSE;
    }

    // Run-time matcher for method calls
    public boolean matchesCall(Advisor advisor, AccessibleObject within,
                               Class calledClass, Method calledMethod,
                               Object target, Object[] args,
                               List<SelectorParam> selectorParams) {
        String xpathExpression = selectorParams.get(0).getValue();
        Document docroot = getWsDocument(target, args);
        XPath xpath = XPathFactory.newInstance().newXPath();
        NodeSet resultNodes =
                (NodeSet) xpath.evaluate(xpathExpression, docroot,
                                         XPathConstants.NODESET);
        return (resultNodes != null && resultNodes.getLength() > 0);
    }

    // Returns true if the given method call is a call to a method that
    // sends out a SOAP request.
    private boolean isWsRequestMethod(MethodCall methodCall) { ... }

    // Gets the DOM document for the SOAP request.
    private Document getWsDocument(Object target, Object[] args) { ... }
}
```

Listing 9: A selector for XML elements in Web Services requests



takes a single parameter, which is an XPath expression, and selects the outgoing requests that contain the XML elements specified by that expression. The selector class implements two selector methods: the weave-time matcher that picks method calls and the corresponding run-time matcher.

Let us assume that the underlying Web services framework has one or more sender methods, which take the actual responsibility for sending out SOAP requests. The weave-time matcher flags every call to a sender method as requiring a run-time check. For any other method calls, it returns "join point not matched". Let us also assume that a sender method receives as a parameter the SOAP request (an XML document) to be sent out. When the run-time matcher intercepts a call to the sender method, it has access to that parameter. It evaluates the XPath expression against the outgoing SOAP request. If the XPath evaluation produces a non-empty set of XML elements, the run-time matcher returns "join point matched". Otherwise, it returns "join point not matched". Note that our `WsRequestSelector` is specific to a given SOAP framework. In order to intercept sender methods and access the outgoing requests, it needs knowledge on certain details of that framework. The methods `isWsRequestMethod` and `getWsDocument` encapsulate that knowledge.

Framework-Specific Selectors

Pointcuts that use framework-specific metadata are a promising application area for selectors. It is often desirable to use framework metadata to decide which elements of a program should be affected by an aspect. In many cases, a selector is the only way for a pointcut to employ framework metadata. Such situations arise, for instance, when the metadata is external to the Java code (e.g., metadata in XML files). In other cases, it would be possible to define a pointcut that accesses framework metadata without resorting to a selector, but the aspect programmer would need framework-specific knowledge in order to write the pointcut. This need would be a burden on the programmer and would generate excessive coupling.

We propose framework-specific selectors that let the programmer use framework concepts and metadata to define pointcuts in a modular way. Such selectors would encapsulate the framework details and would expose to the programmer high-level concepts. In this approach, framework-specific selectors would better be provided as parts of the framework upon which they rely, as a selector library that applications could use as they use the class and procedure libraries distributed with the framework. Such an arrangement respects the modularity of the framework and shields the application/aspect programmer from framework-specific details.

With the goal of validating the proposal outlined above, we have implemented a selector library for Hibernate [4], a popular framework for object/relational mapping. Our library has selectors that pick executions of persistent attribute getters, persistent attribute setters, primary key getters and primary key setters. It also has a `withinPersistent` selector, which picks classes that represent persistent entities. Listing 10 shows a selector for persistent attribute setters. The selector uses the framework itself to load the metadata.

```
@SelectorDef(name = "hibernateSetter")
public class HibernateSetterSelector extends SelectorBase {

    // Hibernate SessionFactory
    private SessionFactory sessionFactory =
        new Configuration().configure().buildSessionFactory();

    // Weave-time matcher for method executions
    public SelectionValue matchesExecution(Advisor advisor,
                                           CtMethod m,
                                           List<SelectorParam> params) {
        // Gets the method's class
        Class declaringClass = m.getDeclaringClass().toClass();
        // Gets the class's persistent properties and iterates over them
        ClassMetadata cmd = sessionFactory.getClassMetadata(declaringClass);
        if (cmd == null)
            return FALSE; // Not a persistent class
        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.getName().equals(m.getName()))
                return TRUE;
        }
        return FALSE;
    }
}
```

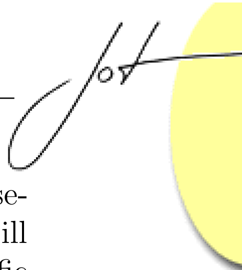
Listing 10: A selector for Hibernate property setters

To show how the `hibernateSetter` selector can improve pointcut quality, we will compare two definitions for a pointcut intended to pick setters of persistent attributes of type `java.util.Date`.

The first pointcut definition, shown in Listing 11, uses a naming pattern to match the setters. This pointcut works under two assumptions: (*i*) that all classes under the package `com.acme.someapp` are mapped for persistence, and (*ii*) that all methods starting with "set" in those classes are setters for persistent attributes. If any of these assumptions fails, the pointcut will fail. Another drawback is that a programmer looking at this pointcut will not immediately know that its intent is to capture setters for persistent attributes. A comment would have to be added for that to become clear. Therefore, the pointcut has low quality: it has low tolerance to changes in the base program and does not communicate its intent clearly.

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

Listing 11: A pointcut based on naming conventions



In the second pointcut definition (Listing 12) we use the `hibernateSetter` selector to choose only the methods that are setters for persistent attributes. 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. Additionally, it is clear to the programmer that we are picking only methods that are setters for persistent attributes managed by Hibernate. Therefore, we have enhanced the pointcut quality considerably. A major advantage of this approach is that the access to framework-specific metadata 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.

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

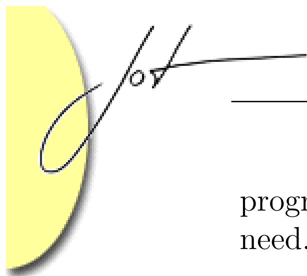
Listing 12: A pointcut based on the hibernateSetter selector

5 RELATED WORK

Several approaches have been proposed for improving the expressiveness of pointcut languages. Some authors have used logic languages as a basis for pointcut languages. In [9], 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 Smalltalk, and this language's meta-information facilities are used as a basis for the join point model. In [10], the authors explain what features of their language make it a good fit for defining pointcuts.

In [17], the authors propose the aspect-oriented 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 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. It uses a less elegant model, but is more tractable in practice. Alpha works with 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 4, 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.

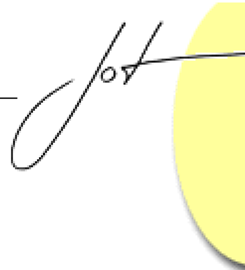
In [8], the authors propose the use of the functional language XQuery as a replacement for current pointcut languages. They run XQuery on an XML representation of Java bytecode. The main shortcoming of their approach is that it employs only weave-time information. It also adds to the weaving process one more step, which builds the XML representation of the bytecode.

Josh [7] 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 weave-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, which can be difficult and error-prone, is not needed in our approach.

The AOP part of the Spring framework [13] defines all of its pointcuts through Java classes. It has a mechanism for the combination of weave-time and run-time checks that is very similar to ours. However, it does not provide a language to easily combine pointcuts. Instead, it relies on verbose XML definitions. Moreover, Spring AOP does not give access to a powerful API for structural reflection that could be used for join point selection: it relies exclusively on standard Java reflection. That makes it difficult to use Spring AOP for implementing more powerful selectors, such as the one described in section 4.

The AspectBench Compiler (*abc*) [3] is a framework for experimentation of novel language features and implementation techniques in AspectJ. It supports extensions on the syntax of AspectJ, on its type system, and on the set of possible join points. *abc* also lets language researchers create new kinds of pointcuts and advice, as well as new semantic checks and optimizations. Its back-end pointcut language partitions the pointcuts in four categories: lexical pointcuts, shadow pointcuts, dynamic pointcuts, and compound pointcuts. This categorization makes it easier to compile new primitive pointcuts into existing ones. *abc* also has a notion of *dynamic residue*, which in our terminology would correspond to the run-time part of a selector. Albeit powerful, *abc* is a complex framework. The complexity of *abc* may be an overkill for users, but that framework could certainly be used as the basis for a simpler extension mechanism, such as ours.

SCoPE [1] is an AspectJ compiler that supports user-defined analysis-based pointcuts. Rather than extending the language, SCoPE lets the programmer write pointcuts that analyze the base program by using a plain `if` pointcut with introspective reflection libraries. It introduced a novel compilation scheme based on a back-patching technique. Unlike our work, SCoPE does not deal with user-defined dynamic pointcuts. Nevertheless, its sophisticated static techniques could be applied to user-defined join point selectors in order to enhance weave-time matching.



6 CONCLUSION AND FUTURE WORK

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 external metadata sources. Moreover, it lets framework implementors define framework-specific selectors, which allow aspects to cross the boundaries of a given framework and of its different artifacts, while still respecting the modularity of that framework.

A feature that will make user-defined join point selectors even more useful is the possibility of aggregating metadata and making such information available to advice implementors. That feature would be especially useful to selectors that take advantage of external metadata. This extension would also make it possible to implement a feature of the Doxpects DSL that the example in section 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 the selector semantics:

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

These changes would make it possible to use the result of one selector as an argument to another one. For example, the `updatesStateReadBy` selector (section 4) could be divided in two selectors: the first one would select all fields read by a given method and the second would select all methods that update any of a given set of fields. Both selectors could then be reused independently.

REFERENCES

[1] Tomoyuki Aotani and Hidehiko Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD 2007)*, pages 161–172. ACM Press, 2007.

[2] AspectWerkz Project. *http://aspectwerkz.codehaus.org/*, 2005.

[3] Pavel Avgustinov et al. abc: An extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 87–98. ACM Press, 2005.

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

[5] Cristiano Breuel and Francisco Reverbel. Join point selectors. In *Proceedings of the 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT 2007)*, pages 14–21. ACM Press, 2007.

[6] Shigeru Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, 2000.

[7] Shigeru Chiba and Kiyoshi Nakagawa. Josh: an open AspectJ-like language. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 102–111. ACM Press, 2004.

[8] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In *Proc. 2nd Asian Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2004.

[9] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. In *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.

[10] Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 60–69. ACM Press, 2003.

[11] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 26–35. ACM Press, 2004.

[12] JBoss AOP Project. *JBoss AOP Reference Documentation (v1.3)*, 2005. http://docs.jboss.com/aop/1.3/aspect-framework/.

[13] Rod Johnson et al. *Spring - Java/J2EE Application Framework Reference Documentation*, 2006. http://static.springframework.org/spring/docs/2.0.0/reference/index.html.

[14] Mik Kersten. AOP@Work: AOP tools comparison, Part 1: Language mechanisms. Technical report, IBM Developer Works, February 2005.

[15] G. Kiczales et al. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, 2001.

[16] Gregor Kiczales. The fun has just begun. Keynote at AOSD 2003, March 2003.

[17] Karl Klose and Klaus Ostermann. Back to the future: Pointcuts as predicates over traces. In *Proc. 4th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, 2005.

[18] 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.

[19] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *Proc. 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer, 2005.

[20] Eric Wohlstadter and Kris De Volder. Doxpects: aspects supporting XML transformation interfaces. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 99–108. ACM Press, 2006.

ABOUT THE AUTHORS

Cristiano Breuel has a Computer Engineering degree from the University of São Paulo, Brazil, and a M.Sc. degree in Computer Science, also from the University of São Paulo. He can be reached at cmbreuel@ime.usp.br.

Francisco Reverbel is a Professor of Computer Science at the University of São Paulo, Brazil. He has E.E. and M.Sc. degrees from the University of São Paulo and a Ph.D. from the University of New Mexico. He can be reached at reverbel@ime.usp.br. See also http://www.ime.usp.br/~reverbel/.