Open Static Pointcuts Through Source Code Templates

Carlos Noguera, Renaud Pawlak

Open Static Pointcuts Through Source Code Templates

Carlos Noguera and Renaud Pawlak
INRIA Futurs - JACQUARD project
LIFL,Université des Sciences et Technologies de Lille
{noguera,pawlak}@lifl.fr

Abstract

Aspect languages define ways to modularize croscutting concerns by means of expressing them as aspects. The expressiveness of an aspect language is very much affected by the expressiveness of the language it uses to describe pointcuts. This is due to the fact that pointcuts define what is crosscutting in a crosscutting concern. We present a mechanism to express type-safe source code templates in pure Java that improves the expressiveness of pointcut languages, and an extension to AspectJ that uses templates to enhance its pointcut designator language.

Keywords AOSD, Pointcut languages, Templates.

1. Introduction

Aspect languages define ways to modularize croscutting concerns by means of expressing them as aspects. Aspect languages usually express, as stated in [1], a set of *joinpoints* that conform to a given model; a way to select a subset of joinpoints, *pointcuts*; and the behavior to execute at selected joinpoints *advice*.

The expressiveness of an aspect language is very much affected by the expressiveness of the language it uses to describe pointcuts. This is due to the fact that pointcuts define what is crosscutting in a crosscutting concern. To provide a certain degree of genericity, pointcut languages usually rely on *wildcards* to specify a family of joinpoints related by name, as is the case with AspectJ. However, wildcard-based pointcut languages have drawbacks.

For example, a common place to insert aspects is whenever a Java bean state changes. The pointcut to match the changes on an object's (of class `A`) state in AspectJ, would then be: **call** (**public** A.set ∗(..)) . This, however, may match unintended joinpoints, and exclude others that do not comply with the naming convention.This is a known problem with current aspect language implementations [4, 2, 7].

In order to alleviate this problem, a more expressive pointcut language is required. We propose to match, not only on the signature, but also on the structure of the method; so that, for the setter example, it would suffice to state that the structure of a setter method is that of, for example, a *method that receives a parameter* `x` *of type* `T` *and assigns that parameter to a field of the a subtype*. Given that the structure of a method is more related with its behavior than the method's name, specifying structure should allow a more expressive pointcut language.

To express this setter pattern it would be interesting to use a *source code template* that defines the structure to look for by defining which parts are fixed, and which parts can be varied. In figure 1, such a structure is presented. In it, parts enclosed in _ are variable while the rest are fixed (_SubType_ is a subtype of _Type_).

```
_Type_ _name_;

public void _methodName_(_SubType_ _x_){
  this._name_ = _x_;
}
```

Figure 1. Setter structure

In this position paper, we propose a way to extend AspectJ's pointcut language with structural constructs in the form of type-safe native Java source code templates. These templates allow the definition of complex pointcuts that are difficult (or impossible) to express using wildcard-based pointcut languages. The rest of the paper is organized as follows: in section 2, source code templates using a framework called Spoon are introduced. The use of templates as Pointcut Descriptors (PCDs) is discussed in section 3.1, an extension of AspectJ that uses pointcuts and some applications are shown in sections 3.2 and 3.3. Finally, how this approach relates to others, and some concluding remarks can be seen in sections 4 and 5

2. Source Code Templates in Spoon

Spoon [6] is a framework for source code processing that we have developed in the context of the Jacquard INRIA project[1]. It provides features such as compile-time reflection, a query system, and Java 5 support.

Spoon implements source code transformations by reifing the target's source code into a typed compile-time (CT) model. Whenever these transformations consist of adding or replacing elements of the model, aside of reflection, we have introduced templates as containers of reusable model elements. Given a template, Spoon is able to use its CT model to add or replace an element in the model. The way these templates defined and used is explained in the following sections.

2.1 Definition of Type-safe templates in Java

Templates in Spoon are normal Java 5 classes that implement the `Template` interface. Each template defines a number of attributes that act as parameters representing the variable parts of the template. These parameters can represent typed expressions, one or

[1] http://www.lifl.fr/jacquard/

many statements blocks, types, and the identifiers of variables and methods. All other attributes and methods in the Template class correspond to the fixed part of the template. Figure 2 shows a complete template[2] which describes a method with a (fixed) name m, and a single parameter with a variable type and name (_T_, _paramName_). The body of the method contains a single `if` statement with its corresponding body.

```
public class IfTemplate<_T_> implements Template{
  TemplateParameter<Boolean> _cond_;
  TemplateParameter<Void> _Tbody_;

  @Parameter Class<_T_> _T_;
  @Parameter String _paramName_;

  public void m(_T_ _paramName_){
    if(_cond.S())
      _TBody_.S();
  }
}
```

Figure 2. Template

Parameters in templates correspond to two main groups: `TemplateParameters`, and `@Parameters`. The first corresponds to expressions, statements, and collections of statements. TemplateParameters are parametrized by the type of the element they represent. Since templates do not use special syntax, they can be compiled by a standard Java 5 compiler. To be able to take advantage of the compiler's type-checker, TemplateParameters have an S() method that has as a return type the type parameter stated on the TemplateParameter's declaration. In Fig 2 this can be seen in the parametrization of the TemplateParameter _cond_. Since _cond_ is parametrized by Boolean, that will be the return type of _cond_.S(). When that expression is used in the body of the **if**, the compiler correctly type-checks it as the expected **boolean**. In general, TemplateParameters parametrized by Void correspond to statements (blocks, ifs, loops).

`@Parameters`, on the other hand, are use to capture types and identifiers; in the example above, _T_ corresponds to the type of the parameter in the method, whereas _paramName_ corresponds to the name of the parameter.

2.2 Template Matching

Templates, as described in the previous section, define a source code model in which some elements are variable. By matching against parts of a program's source code, we are able to find which elements conform to the pattern described in the template. For example, if it was necessary to find all the `if`s that do not have a corresponding `else` or `else if` expression, it would be possible to use the template defined in figure 2, since it describes the desired structure. To this end, we have implemented in Spoon a way to match templates to elements of the program.

The matching is performed by traversing the model of the template and the model of the program in parallel. For each subelement of both models, one of the following cases occur: if they are equal, then it is checked if their children match; if they are not, and the template's current element is a parameter, a match is found; otherwise, the two models do not match. This process is illustrated by figure 3. It is important to note that by comparing the two models, instead of the source code itself, the process can match on expressions that are written differently, but mean the same thing; for example, f = 0 and **this**.f = 0 if f is a field.

[2] a type parameter <_T_> is needed to compile the template

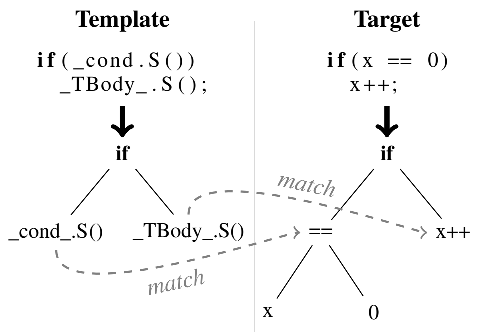

Figure 3. Template model matching

While templates may be useful for describing some code structures in a given program, they may fall short when using them to express, for example, negative matches (when trying to match the places where a structure *does not* occur), optional matches or more complex matching strategies. For these cases we have implemented a way to open up the matching process by allowing the programmer to control how certain template parameters are matched. To this end, the @Parameter annotation takes an optional match parameter that specifies a class of type ParameterMatcher to which the matching engine will delegate the match decision.

In figure 4, a fragment of the template introduced in figure 2 with a custom matcher is shown. In it, access to both the model of the template and the target code to match, as well as the template matching engine are provided.

```
...
  @Parameter(match=TestMatcher.class)
  TemplateParameter<Void> _Tbody_;

...
  public void m(_T_ _paramName_){
    if(_cond.S())
      _TBody_.S();
  }
}

public class TestMatcher
   implements ParameterMatcher{

 boolean match(TemplateMatcher templateMatcher,
             CtElement template,
             CtElement toMatch){
   //Particular match strategy
 }

}
```

Figure 4. Open template parameter

3. Open Static Pointcuts

In this section, a way to use templates as PCDs is discussed. As a proof of concept, we have implemented an extension to AspectJ 5 that includes a limited form of template pointcuts. Finally we show some examples to support the usefulness of template pointcuts.

3.1 Templates as Pointcut Designators

As stated in section 1, templates offer a more expressive way to define certain static pointcuts by declaring the structure of the joinpoints, rather than wildcard-based pointcut languages.

To implement template pointcuts, the aspect tool must check each joinpoint in the program to see if it matches the template, and advice those that do. Template pointcuts can be used in combination with dynamic pointcuts constructs, for example, to advice a call to a given method when it has a certain structure, i.e. matches a template.

3.2 Definition in AspectJ 5

To test the utility of using templates as pointcuts, we have added them to AspectJ. To this end, we base the prototype on the "annotation based development style" (@AspectJ) introduced in AspectJ 5. @AspectJ allows the writing of aspects as normal Java classes that use annotations to define pointcuts, advice and inter-type declarations. Since aspects are valid Java classes that do not use special syntax, we are able to process them using Spoon to add the notion of template pointcuts. Note that in order to do this, the source code of the whole program must be available to Spoon.

Whenever a template pointcut is needed, the AspectJ pointcut is annotated with a @Matches annotation that takes as parameter the class that contains the template. Spoon is then used to process the annotated aspect. It uses the template in the annotation's parameter, and tries to find all the places (methods or classes depending on the template) that correspond. When an element matches the template, it is annotated with a marker annotation derived from the name of the template, and the original @AspectJ pointcut is modified to take into account the marker annotation.

A template may refer to the structure of the method that is being called (target), or it may refer to the structure of the method that produces the call (source). This distinction is taken into account by using @Matches for target methods, and @SourceMatches for source methods. The use of one or the other produces different modifications to the original @AspectJ PCD; @Matches are translated into @annotation(Marker), while @SourceMatches are translated into @withincode(Marker). Other annotations such as @NotMatches are implemented with their respective modifications to the @AspectJ PCD.

An example is presented in figure 5. In it, the template introduced in figure 2 is used to define a "*guarded method*", which stands for a method that only executes if a condition is met. After the aspect has been processed by Spoon, the PCD on the example will be changed to `"call * org.test.A.*(..) && @annotation(Marker_IfTemplate)"`.

```
@Aspect public class Example{

  @Matches(IfTemplate.class)
  @Before("call * org.test.A.*(..)")
  public void advice(){
    Logger.log("call to guarded method");
  }

}
```

Figure 5. @AspectJ and template pointcut

3.3 Applications

Idiom checking. With AspectJ, it is possible to declare that a pointcut corresponds to unwanted patterns in the source code. This is archived by using declare error and declare warning followed by a PCD that matches the unwanted pattern. Using only AspectJ's pointcut syntax, the expressiveness of the errors becomes limited to calls and executions. However, using templates, it is possible to express unwanted idioms in the source code inside methods. For example, given that `String`s in Java are immutable, writing String s = **new** String("x"); creates two strings: one because of the **new** and one for "x". A template containing this idiom can be created and associated with an AspectJ's declare warning to warn whenever string objects are being wasted. Such a template would be written as String _s_ = **new** String("_string_"); (where both _s_ and _string_ are variables) and then used to annotate a declare warning.

Partially weaved aspects. Aspects are normally used to *add* behavior to a base program. If it is the case that the program already (partially) implements it, the programmer must be careful not to inject the behavior twice. This situation can negate the advantages of wildcard-based pointcut languages since the name of the joinpoint alone says nothing of whether the behavior is present or not.

Template pointcuts can aid in this case by filtering out those joinpoints that already address the concern. The @NotMatches annotation can be used in an AspectJ pointcut to include joinpoints that *do not match* a given template. Using this annotation, it is possible to filter out those methods that have a structure similar to that of the corresponding advice. For example, in the aspect on figure 5, all calls to a "guarded method" are logged. It would be interesting not to advice methods that already implement logging within their body. This is achieved by annotating the PCD with an additional @NotMatches(Logged.class) – defined in figure 6. The @NotMatches annotation modifies the PCD by inserting a @annotation(Marker). In the Logged template a special annotation @Statement is used to say that the structure to match is only that of the statement contained inside the dummy method.

```
public class Logged implements Template{

  TemplateParameter<String> _message_;

  @Statement
  public void dummy(){
    Logger.log(_message_.S());
  }

}

@Aspect public class Logging{

  @NotMatches(LoggedTemplate.class)
  @Before( /* AJ pointcut expression */)
  public void log(){
    Logger.log("Some message");
  }
}
```

Figure 6. Template used to filter partially implemented concerns and corresponding aspect

Caller-side patterns Certain tasks in Java are achieved through a well-established protocol; for example reading from text files. A usual way to read from a text file is by using a FileInputStream, and then convert it into a BufferedInputStream so that lines can be extracted from the file. BufferedInputStreams, however, can be used as decorators for other stream sources, such as URL connections. Therefore, to place an advice each time that a line is read from a file by BufferedInputStream#readLine(), it is necessary to filter out those calls that do not come from a FileInputStream. Using templates, this can be achieved by creating a template that describes the pattern (shown in figure 7), and annotating the advice with a @SourceMatches annotation. The _before_ and _after_ template parameters represent multiple statements, that are used to allow the pattern to appear anywhere inside a method.

```
public class FileReaderTemplate
   implements Template {

  @Parameter String _fileName_;
  @Parameter String _fis_;
  TemplateParameter<Void> _whileBlock_;
  TemplateParameterList _before_,_after_;

  @StartBlock
  public void test() throws IOException{
    _before_.S();
    FileInputStream _fis_ =
       new FileInputStream(_fileName_);
    BufferedReader _input_ =
      new BufferedReader(
         new InputStreamReader(_fis_));
    while (_input_.readLine() != null)
      _whileBlock_.S();
    _after_.S();
  }
}
```

Figure 7. Template to match the structure of file reading in Java.

4. Related Work

In [4], Gybels et al. introduce a pointcut language for Smalltalk that addresses the issues presented here. However, they use Prolog predicates as PCDs; these predicates reason about dynamic as well as static pointcuts. By using logic programming and unification, they count with a more expressive way to describe pointcuts. Nevertheless, describing complex static shadows with logic predicates is not as straight forward as using source code templates.

In a similar approach, Kniesel et al [5] propose an extension to the AspectJ language called LogicAJ. They also use Logic assertions to obtain a more expressive pointcut language that carries the same advantages as Gybels' approach. Still, despite the use Java as a base language, they do not offer a notion of type safeness in their approach.

Also, Eichberg et al [3] propose the use of the XQuery functional language on XML representations of Java classes to describe pointcuts. The use of functional language provides advantages over logic-based approaches such as improved composability. Since the pointcuts are described on an XML model of the program, their XQuery expressions can become hard to understand when compared with native source code templates in which the structure they represent is closer to the one of the method.

Josh [2] is an aspect language inspired in AspectJ proposed by Chiba et al. As main feature, Josh proposes open pointcuts. Pointcuts in Josh are defined by implementing a method, in pure Java, that decides if a joinpoint should be adviced or not by inspecting the structure of the joinpoint. Their approach is similar to ours in the use of CT reflection to specify open pointcuts. In the advice, template-like parameters for context exposure are used. Josh's pointcut language, although open, suffers from the lack of declarativeness of XQuery when compared with Spoon templates. Nevertheless, the use of template parameters in the advice permits greater genericity than the one provided on our prototype.

5. Conclusion

We have presented an mechanism to express type-safe source code templates in pure Java, and an extension to AspectJ that uses templates to enhance its PCD language.

The use of templates to extend AspectJ's pointcut language is constrained by its joinpoint model. Given that AspectJ is, mainly, a dynamic aspect language, it can be difficult to exploit the full expressiveness of template-based pointcuts. Nevertheless, the prototype does enhance the static part of AspectJ, proving its usefulness in particular for the declare errors/warnings construct.

It may be argued that the use of templates that rely on a certain code structure that specifies a behavior are more brittle than a pure wildcard-based approach. However, given that templates are defined as pure java classes, they will be taken into account by automatic refactoring tools whenever, for example, renaming or moving entities (classes or methods) in the system. This will be a step towards reducing the gap between the base program and aspects applied to it whenever the system evolves.

While templates provide a way to express patterns that occur in the source code, these patterns may not be generic enough to encompass the diverse ways in which a certain task can be performed (using intermediate variables, for example). For these complex patterns, programmatic checks can be implemented using the *callback* mechanism introduced in section 2.2. Using this, Josh-like open pointcuts can be implemented. However, we believe that using a mix of templates and programatic checks eases the task of writing pointcuts by allowing the programmer to fall back on compile-time reflection only for the places that really warrant them, instead of having to describe the whole structure using reflection alone.

By using template based pointcuts on a fully static AOP language, it will be possible to, for example, provide access to the variables bound during template matching in the advice. This can improve the performance of aspects that need access to the signature of the adviced methods.

References

[1] K. v. Berg, J. M. Conejero, and S. Chitchyan. AOSD ontology 1.0. Technical report, AOSD-Europe Network of Excellence, May 2005.

[2] S. Chiba and K. Nakagawa. Josh: an open AspectJ-like language. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 102–111, Lancaster, UK, 2004. ACM Press.

[3] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as Functional Queries. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004*, pages 366–382, Taipei, Taiwan, November 2004. Springer-Verlag Heidelberg.

[4] K. Gybels and J. Brishau. Arranging Language Features for More Robust Pattern-based Constructs. In *AOSD*, 2003.

[5] G. Kniesel and T. Rho. Generic Aspect Languages - Needs, Options and Challenges, JFDLPA 2005. Sep 2005.

[6] R. Pawlak. Spoon: annotation-driven program transformation — the AOP case. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, pages 1–6. ACM Press, 2005.

[7] T. Tourvé, J. Brishau, and K. Gybels. On the Existence of AOSD-Evolution Paradox. AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, 2003.