

Aspect Oriented Programming (AspectJ) and Meta-Programming Technologies including Java Reflection

David Dodds Open-Meta Computing Inc open-meta.com

underlying AOP and Reflection are ontologies and meta-data

In aspect oriented programming "concerns" crosscut code. Crosscut is a "spatial" term. What is the meaning of the "direction" of the bulk of the code, which has an implied direction? Spatial metaphor here can be used for thinking about programming including "concerns". Machine based reasoning at the high-level of "crosscutting concerns" ("aspects") can be used to derive or generate computer code such as Java and AspectJ.

One way in which the meta-programming system which performs the code generation (which implements the mapping of A into B) might be programmed is to use the template-filling technique such as is available in C++. This might be considered to be the approach where fixed programs have their parameters defined by the meta-programming system which instantiates those fixed programs. A pattern-matching facility such as regular expressions might be used to "detectZ" or match up with text patterns in A, and invoke predefined templates with appropriate template filler data / "parameters". Linguists call this "surface mapping". This means that the text patterns actually present in the input (A) are mapped to predefined output data pattern (B). This technique, while an obvious one is quite weak.

Let's look at Java Reflection next and see why it is useful to meta-programming systems as a source of knowledge about HOW to perform its (spatially oriented) transformations of input code into "aspected" form.

Here is an example of Using Java Reflection:

<http://java.sun.com/developer/technicalArticles/ALT/Reflection/Article>
Using Java Reflection

January 1998

Finding Out About Methods of a Class

One of the most valuable and basic uses of reflection is to find out what methods are defined within a class. To do this the following code can be used:

```
import java.lang.reflect.*;
public class method1 {
private int f1(
Object p, int x) throws NullPointerException
{
if (p == null)
throw new NullPointerException();
return x;
}
public static void main(String args[])
{
try {
Class cls = Class.forName("method1");
Method methlist[]
= cls.getDeclaredMethods();
for (int i = 0; i < methlist.length;
i++) {
Method m = methlist[i];
System.out.println("name
= " + m.getName());
System.out.println("decl class = " +
m.getDeclaringClass());
Class pvec[] = m.getParameterTypes();
for (int j = 0; j < pvec.length; j++)
System.out.println("
param #" + j + " " + pvec[j]);
Class evec[] = m.getExceptionTypes();
for (int j = 0; j < evec.length; j++)
System.out.println("exc #" + j
+ " " + evec[j]);
System.out.println("return type = " +
m.getReturnType());
```

```
System.out.println("-----");
}
}
catch (Throwable e) {
System.err.println(e);
}
}
}
```

A (second order) meta-programming system (m-p) can use Java Reflection at run-time (not compile time) to discover the methods used by classes in the system of code which constitutes the bulk of the total system! The m-p can then use Java Reflection technology to execute those discovered methods. If there is meta-data available to the m-p (perhaps in OWL, Protege or other ontological form, in part) which maps the names of the objects and their methods into a "meaning" representation, one which informs the m-p what the purpose or functionality those names relate to, then the m-p can make informed decisions as to how to translate goals into an organized set of method invocations (which it discovered through Reflection). This in effect allows the planning component of the m-p to decompose a goal (description) into a set of [reflected] method invocations by referencing (representation) as to their effects when executed.

Meta-Computing the Derivation of Aspectual Concerns and Advice Using Ontologies

The AspectJCookBook (O'Reilly) defines "A cross-cutting concern is behaviour, and often data, that is used across the scope of a piece of software." "Logging is a cross-cutting concern.. Logging is potentially applied across many classes, and it is this form of horizontal application of the logging aspect that gives cross-cutting its name. The code that is executed when an aspect is invoked is called advice. Advice contains its own set of rules as to when it is to be invoked in relation to the join point that has been triggered. Join points are simply specific points within the application that may or may not invoke some advice. Pointcuts encapsulate the decision-making logic that is evaluated to decide if a particular piece of advice should be invoked when a join point is encountered."

(The `Containz()` predicate in the DAXSVG spatial ontology is able to recognize spatially located join points and handle advice related to them. Such as `Containz(call MyClass foo)` invokes `Advice(print "Hello World", /"In the advice attached to the call point cut")`.)

Next we see some AspectJ Java code to illustrate what it looks like. The code itself runs vertically down the page. "Concerns" then are conceived as "horizontal". This means that spatial terms such as "before" and "after" have relevant computational as well as spatial metaphorical meaning. In the ontology shown further down we see the spatial ontology term "Before", notice also that AspectJ has a function called "before()." The spatial metaphor given in the definition of "concern" and "advice" by O'Reilly can be computationally acted upon via its spatial metaphor by use of a spatial ontology such as that one given further below (DAX and CYC).

```
package com.oreilly.aspectjcookbook;
public class MyClass
{
public void foo(int number, String name)
{
System.out.println("Inside foo (int, String)");
}
public static void main(String[] args)
{
// Create an instance of MyClass
MyClass myObject = new MyClass();
// Make the call to foo
myObject.foo(1, "Russ Miles");
}
}
package com.oreilly.aspectjcookbook;
/**
 * <p>Title: HelloWorld aspect for Recipe 2.2</p>
 * <p>Description: Simple aspect for demonstrating a very simple
aspect</p>
 * <p>Copyright: Copyright (c) 2003 Russell Miles</p>
 * @author Russell Miles
```

```

* @version 1.0
*/
public aspect HelloWorld
{
/*
Specifies calling advice whenever a method
matching the following rules gets called:
Class Name: MyClass
Method Name: foo
Method Return Type: * (any return type)
Method Parameters: an int followed by a String
*/
pointcut callPointCut() :
call(void com.oreilly.aspectjcookbook.MyClass.foo(int, String));
// Advice declaration
before() : callPointCut()
{
System.out.println(
"Hello World");
System.out.println(
"In the advice attached to the call point cut");
}
}
}

```

Following is a partial listing of Daxsvg ontology, a spatial ontology.

```

<!-- Copyright 2001 - 2005 David Dodds
-->

```

```

<rdf:RDF xml:lang="en"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
<rdfs:Class rdf:ID="SvgEntity">
<rdfs:comment>The class of SVG entities, referenced by their id in
the SVG code.
</rdfs:comment>
<rdfs:subClassOf rdf:resource="http://www.openmeta.
com/2000/01/rdf-schema#Resource"
/>

```

```

</rdfs:Class>
<rdf:Property ID="Near">
<rdfs:comment>has a degree of nearness (by value). g1(x)
</rdfs:comment>
<rdfs:range rdf:resource="#SvgEntity" />
<rdfs:domain rdf:resource="#SvgEntity" />
</rdf:Property>
<rdf:Property ID="Before">
<rdfs:comment>has a degree of precedingness (by value). g30(y1, y2)
</rdfs:comment>
<rdfs:range rdf:resource="#SvgEntity" />
<rdfs:domain rdf:resource="#SvgEntity" />
</rdf:Property>

```

```

public real Before( int y1, int y2 )
{
    if (y1 < y2 )
    {
        return (near(abs(y2 - y1), contexty));
    }
    else
    {
        return 0;
    }
}

```

```

<rdf:Property ID="IsNear">
<rdfs:comment>has a degree of nearness (by value). g1(x)</rdfs:comment>
<rdfs:range rdf:resource="#www.open-meta.com/2001/IsNear" />
<rdfs:domain rdf:resource="#SvgEntity" />
</rdf:Property>

```

g1(centroiddistance)

centroiddistance = $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$

IsNear = g1 = $1 - (1/2 + 1/\text{PI} * \arctan(\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}) - k(1)/k(2))$

(IsNear(x1,y1,x2,y2))

Below are several items from the CYC ontology which convey the meaning of various aspects of “above” (a synonym of “before”). Remember that there is nobody home in a computer and so this pointy bracket verbiage is necessary to provide a modicum of enlightenment to the computer, of stuff which we tacitly know and take for granted everyday.

```
<rdf:Property rdf:ID="above-Directly">
  <rdfs:label xml:lang="en">above - directly</rdfs:label>
  <rdfs:comment>(#$above-Directly ABOVE BELOW) means either that
    (1) the volumetric center of ABOVE is directly above some
    point of BELOW, if ABOVE is smaller than BELOW; or that (2)
    some point of ABOVE is directly above the volumetric center
    of BELOW, if ABOVE is larger than, or equal in size to,
    BELOW.</rdfs:comment>
  <guid>bd58fbde-9c29-11b1-9dad-c379636f7270</guid>
  <rdfs:subPropertyOf rdf:resource="#above-Generally"/>
  <rdfs:domain rdf:resource="#SpatialThing-Localized"/>
  <rdfs:range rdf:resource="#SpatialThing-Localized"/>
</rdf:Property>
<rdf:Property rdf:ID="above-Generally">
  <rdfs:label xml:lang="en">above</rdfs:label>
  <rdfs:comment>(#$above-Generally OBJ1 OBJ2) means that the
    $$SpatialThing-Localized OBJ1 is more or less above the
    $$SpatialThing-Localized OBJ2. To be more precise: if OBJ1
    is within a cone-shaped set of vectors within about 45
    degrees of #Up-Directly pointing up from OBJ2 (see
    #Up-Generally), then (#$above-Generally OBJ1 OBJ2) holds.
    This is a more general predicate than #above-Directly
    (q.v.), but it is a more specialized predicate than
    #above-Higher (q.v.). It probably most closely conforms to
    the English word above.</rdfs:comment>
  <guid>be69c623-9c29-11b1-9dad-c379636f7270</guid>
  <rdfs:subPropertyOf rdf:resource="#above-Higher"/>
  <rdfs:domain rdf:resource="#SpatialThing-Localized"/>
  <rdfs:range rdf:resource="#SpatialThing-Localized"/>
```

</rdf:Property>

<rdf:Property rdf:ID="above-Higher">

<rdfs:label xml:lang="en">above - higher</rdfs:label>

<rdfs:comment>(#\$above-Higher OBJ-A OBJ-B) means that OBJ-A is
`higher up'' than OBJ-B. Since most contexts are
terrestrial (see #TerrestrialFrameOfReferenceMt) `higher
up'' typically means that the
#\$altitudeAboveGround of OBJ-A is greater than that of OBJ-

B.</rdfs:comment>

<guid>bf020f6c-9c29-11b1-9dad-c379636f7270</guid>

<rdfs:subPropertyOf rdf:resource="#spatiallyDisjoint"/>

<rdfs:domain rdf:resource="#SpatialThing-Localized"/>

<rdfs:range rdf:resource="#SpatialThing-Localized"/>

</rdf:Property>

<rdf:Property rdf:ID="above-Overhead">

<rdfs:label xml:lang="en">above - overhead</rdfs:label>

<rdfs:comment>(#\$above-Overhead ABOVE BELOW) means that
ABOVE is

directly above BELOW (see the predicate #above-Directly),
all points of ABOVE are higher than all points of BELOW, and
ABOVE and BELOW do not touch.</rdfs:comment>

<guid>bd58b981-9c29-11b1-9dad-c379636f7270</guid>

<rdfs:subPropertyOf rdf:resource="#above-Directly"/>

<rdfs:domain rdf:resource="#SpatialThing-Localized"/>

<rdfs:range rdf:resource="#SpatialThing-Localized"/>

</rdf:Property>

<rdf:Property rdf:ID="above-Touching">

<rdfs:label xml:lang="en">above - touching</rdfs:label>

<rdfs:comment>(#\$above-Touching ABOVE BELOW) means that
ABOVE is

located over BELOW and they are touching. More precisely,
it implies both (#above-Directly ABOVE BELOW) and that
ABOVE #touches BELOW. Examples: a person sitting on a
chair; coffee in a cup; a boat on water; a hat on a head.

(Note that not every point of ABOVE must be higher than
every point of BELOW.)</rdfs:comment>

<guid>bd58f620-9c29-11b1-9dad-c379636f7270</guid>

```
<rdfs:subPropertyOf rdf:resource="#above-Directly"/>  
<rdfs:subPropertyOf rdf:resource="#touches"/>  
<rdfs:domain rdf:resource="#PartiallyTangible"/>  
<rdfs:range rdf:resource="#PartiallyTangible"/>  
</rdf:Property>
```

The pointcut etc locations in the code are spatial metaphors and can be treated metaphorically as spatial locations / patterns, where position (i.e. sequence location) has meaning and hence terms like AspectJ's "before()" have program usable meaning in both the spatial realm of the meta-program planner and of the Java system's AOP.

open-meta.com