CS 683 Emerging Technologies
Spring Semester, 2003
Doc 3 AspectJ Syntax
Contents

Reference

The AspectJ Programming Guide

Reading

The AspectJ Programming Guide

Section 1 Getting Started with AspectJ
Section 2 The AspectJ Language

AspectJ Syntax
Point Example

```
public class Point
   {
   private double x;
   private double y;

   Point(double x, double y)
      {
      this.x = x;
      this.y = y;
      }

   public void setX(double x)
      {
      this.x = x;
      }

   public void setY(double y)
      {
      this.y = y;
      }

   public void setPolarCoordinates(double theta, double r)
      {
      x = r * Math.cos(theta);
      y = r * Math.sin(theta);
      }
   }
```

```
aspect PointChanged
  {
  private boolean Point.isDirty = false;

  public void Point.makeDirty()
    {
    isDirty = true;
    }

  public boolean Point.isDirty()
    {
    return isDirty;
    }

  public void Point.save()
    {
    isDirty = false;
    // add code to save point
    }

  pointcut fieldChanged(Point changed) :
    call( * Point.set*(..)) && target(changed);

  after(Point changed ) : fieldChanged(changed)
    {
    changed.makeDirty();
    }
  }
```

```
import junit.framework.*;

public class PointTests extends TestCase
  {

  public static void main( String args[])
    {
    junit.swingui.TestRunner.run( PointTests.class);
    }

  public void testPointChange()
    {
    Point sample = new Point( 1, 1);
    assertFalse( sample.isDirty() );
    sample.setX(5);
    assertTrue( sample.isDirty());
    sample.save();
    assertFalse( sample.isDirty() );
    sample.setY(2);
    assertTrue( sample.isDirty());
    sample.save();
    assertFalse( sample.isDirty() );
    }
  }
```

Why Not Just Use?

```
public class Point {
   private double x;
   private double y;
   private boolean isDirty = false;

   Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   public void setX(double x){
      this.x = x;
      isDirty = true;
   }

   public void setY(double y) {
      this.y = y;
      isDirty = true;
   }

   public void setPolarCoordinates(double theta, double r) {
      x = r * Math.cos(theta);
      y = r * Math.sin(theta);
      isDirty = true;
   }

   public boolean isDirty() {
      return isDirty;
   }

   public void save() {
      isDirty = false;
      // add code to save point
   }
}
```

Possible Reasons to Use Aspects over Hand Coding

- Point is an existing class and can't or not allowed to change it
- Fewer bugs

  As the number of methods in Point class increase, the more methods that will need to be modified to mark the object as dirty/clean. A programmer may miss some methods that change the state of the object. When programmers add new methods they may forget to add the code to mark the object dirty/clean. We also have to worry about subclasses. With aspects we only have to write one pointcut.

- Easier to change

  If we change the type of events that are interested in aspects allows us to make the change in one location.

- Scales better with additional complexity

  In some cases we may only be interested in when x changes, or when y changes or when both x & y change in one method. Keeping track of these as separate type of events in the Point class will get a bit tricky. Now imagine a class with 10 fields that can change in more complex ways that doubles. With aspects it will be easier to handle these complex situations.

- Flexibility

  If we change our mind about the type of events we are interested in we change one aspect, rather than search all methods in the Point class and its subclasses

Aspect Terms

- Crosscutting Concern

  Issues that cut cross many modules in a program

  Issues dealt with in many modules in a program

- Join Point

  A well-defined point in the program flow

  Method call
  Accessing a field
  Initialization of an object
  Handling an exception

- Pointcut designators (Pointcuts)

  Selecting join points in a program

```
pointcut fieldChanged(Point changed) :
   call( * Point.set*(..)) && target(changed);
```

Aspect Terms

- Advice

  Code that is run when a pointcut is reached

```
after(Point changed ) : fieldChanged(changed)
  {
  changed.makeDirty();
  }
```

- Introduction

  Modify the static structure of a class

  Add fields and methods

  Change the parent of a class

  Add interfaces to a class

```
private boolean Point.isDirty = false;
```

- Aspect

  Unit of modularity for cross cutting concern

  Contains pointcuts, advice, and introductions

Pointcut Designators

AspectJ Primitive Pointcuts

- **call**(MethodPat)
- **call**(ConstructorPat)
- **execution**(MethodPat)
- **execution**(ConstructorPat)
- **initialization**(ConstructorPat)
- **staticinitialization**(TypePat)
- **get**(FieldPat)
- **set**(FieldPat)
- **handler**(TypePat)
- **within**(TypePat)
- **withincode**(MethodPat)
- **withincode**(ConstructorPat)
- **cflow**(Pointcut)
- **cflowbelow**(Pointcut)
- **if**(Expression)
- **this**(TypePat | Var)
- **target**(TypePat | Var)
- **args**(TypePat | Var , …)

Simple Composing of Pointcuts

Operators that combine pointcuts

- || (or)
- && (and)
- ! (not)

call(int *bar(..)) && target(Foo) && ! target(FooSubclass)

Wildcards in Pointcuts

- *

  Matches anything

  Used in return type, Class name, method name

- (..)

  Matches any argument list

execution(* *(..))

call(* set(..)

execution(int *())

call(* setY(Double))

call(* Point.setY(int))

call(* Point.set*(int))

call(* Po*.set*(..))

call(* Po*.*et*(..))

When Advice can be Run

Before a pointcut

After a pointcut

Instead of a pointcut

Examples

Hello.java

```
public class Hello
   {
   public static void main(String[] args )
      {
      System.out.println("Hello");
      }
   }
```

Before

```
public aspect HelloAspect
  {
  before() : call(void main(String[]))
    {
    System.out.println( "Advice");
    }
  }
```

Output of running java Hello

```
Advice
Hello
```

After

```
public aspect HelloAspect
  {
  after() : call(void main(String[]))
    {
    System.out.println( "Advice");
    }
  }
```

Output from running java Hello

```
Hello
Advice
```

Around

```
public aspect HelloAspect
  {
  void around() : call(void main(String[]))
    {
    System.out.println( "Advice");
    }
  }
```

Output from running java Hello

```
Advice
```

Some Forms of Before, after and around

```
before()
before(int k) : call( int * Point.*(int) ) && args(k)
before(Object k) : call( int * Point.*(Object) ) && args(k)

after ()  : call(int Foo.m(int))
after (int k) : call( int * Point.*(int) ) && args(k)
after () returning : call(int Foo.m(int))
after() returning(int x) : call(int Foo.m(int)) && args( x)
after() throwing : call(int Foo.m(int))
after() throwing (NotFoundException e) : call(int Foo.m(int))

returnType around(arguments) : pointcut
returnType around(arguments) throws ExceptionType : pointcut
```

Pointcut Primitives

Call

- **call**(MethodPat)
- **call**(ConstructorPat)

Separate constructs for methods and constructors

Call & Execution

Call

When a method is called

Execution

When the body of code for an actual method is executes

What is the difference?

Simple Example

```
public class Hello {
   public static void main(String[] args ){
      System.out.println("Hello");
   }
}

public aspect HelloAspect {
   before() : call(void main(String[])) {
      System.out.println( "call-before");
   }

   after() : call(void main(String[])) {
      System.out.println( "call-after");
   }

   before() : execution(void main(String[])) {
      System.out.println( "execution-before");
   }

   after() : execution(void main(String[])) {
      System.out.println( "execution-after");
   }
}
```

Output from running java Hello

```
call-before
execution-before
Hello
execution-after
call-after
```

Why Execution?

A method/constructor can be run but not called

```
public class Hello {
   public Hello() {
      this(5);
      System.out.println("Hello()");
   }

   public Hello(int x) {
      System.out.println("Hello(" + x + ")");
   }

   public static void main(String[] args ) {
      new Hello();
   }
}

public aspect HelloAspect {
   before() : call( Hello.new(..) ) {
      System.out.println( "call-before");
   }

   before() : execution(Hello.new(..)) {
      System.out.println( "execution-before");
   }
}
```

Output from running java Hello

```
call-before
execution-before
Hello(5)
execution-before
Hello()
```

Super and Call

```
public class Hello {
   public void foo() {
      System.out.println( "Hello.foo");
   }
   public static void main(String[] args ) {
      Hello test = new HelloChild();
      test.foo();
   }
 }

public class HelloChild extends Hello {
   public void foo() {
      super.foo();
      System.out.println( "HelloChild.foo");
   }
}

public aspect HelloAspect {
   before() : call( * foo(..) ) {
      System.out.println( "call-before");
   }

   before() : execution(* foo(..)) {
      System.out.println( "execution-before");
   }
}
```

Output from running java Hello

```
call-before
execution-before
execution-before
Hello.foo
HelloChild.foo
```

Reflection and Call

```
import java.lang.reflect.*;

public class Hello {
   public void foo() {
      System.out.println( "Hello.foo");
   }

   public static void main(String[] args ) throws Exception {
      Class helloClass = Hello.class;
      Class[] argumentTypes = { };
      Method foo = helloClass.getMethod( "foo", argumentTypes );
      Object[] arguments = { };
      Hello test = new Hello();
      foo.invoke( test, arguments );
   }
}

public aspect HelloAspect {
   before() : call( * foo(..) ) {
      System.out.println( "call-before");
   }

   before() : execution(* foo(..)) {
      System.out.println( "execution-before");
   }
}
```

Output from running java Hello

```
call-before
execution-before
Hello.foo
```

Accessing Fields with Primitive Pointcuts

get( )

Pointcut for reading a field

get(int Point.x)

Refers to any time int Point.x is read

set()

Pointcut for writing a field

set(!private * Point.*)

Refers to any time a non-private field of Point is set

withincode

withincode(methodOrConstrutor)

Refers to any join point in side the given method or constructor

Examples

```
pointcut fieldChanged(Point changed) : set( private double Point.*)
        && target(changed) && withincode(*  Point.*(..));

pointcut fieldChanged(Point changed) : set( private double Point.*)
        && target(changed) && !withincode( Point.new(..));

pointcut fieldChanged(Point changed) : set( private double Point.*)
        && target(changed) && withincode(*  *.*(..));
```

The above three versions of fieldChanged refer to basically same join points: any method in Point class that changes any private field of type double in a Point object. The third actually handles any method that changes any field of type double in a Point object. But since the fields are private only methods in the Point class can modify them.

Example Using set and withincode

Mark a Point object dirty if x or y changes outside of the constructor

```
aspect PointChanged {
   private boolean Point.isDirty = false;

   public void Point.makeDirty() {
      isDirty = true;
   }

   public boolean Point.isDirty() {
      return isDirty;
   }

   public void Point.save() {
      isDirty = false;
      // add code to save point
   }

   pointcut fieldChanged(Point changed) : set( private double Point.*)
         && target(changed) && !withincode( Point.new(..));

   after(Point changed ) : fieldChanged(changed) {
      System.out.println("Make Dirty");
      changed.makeDirty();
   }
}
```

Tests

```
import junit.framework.*;

public class PointTests extends TestCase
  {
  public static void main( String args[])
    {
    junit.swingui.TestRunner.run( PointTests.class);
    }
  public void testPointChange()
    {
    Point sample = new Point( 1, 1);
    assertFalse( sample.isDirty() );
    sample.setX(5);
    assertTrue( sample.isDirty());
    sample.save();
    assertFalse( sample.isDirty() );
    sample.setY(2);
    assertTrue( sample.isDirty());
    sample.save();
    assertFalse( sample.isDirty() );
    }
  }
```

Warning about Errors in the Example

Platform: Mac OS 10.2.3
Java: jdk 1.3.1 (build 1.3.1-root_1.3.1_020714-12:46)
AspectJ Browser
AspectJ 1.0.6

A bus error occurs when running the example with the definition:

```
pointcut fieldChanged(Point changed) : set( private * Point.*)
   && target(changed) && withincode(*  *.*(..));
```

That is using * for the type of the field in set

AspectJ Browser does not compile the example with the definition:

```
pointcut fieldChanged(Point changed) : set( * double Point.*)
   && target(changed) && withincode(*  *.*(..));
```

That is * is used as the access level for the field in set