Lab 1

Getting going with AspectJ and AJDT
Developing AspectJ code
Exercises and new language features

© Copyright 2004, Gregor Kiczales. All rights reserved.

Outline

- **Get AJDT and AspectJ installed**

A. **Load, build and run shapes example** **(Lab1A)**
B. **Add a new shape class (Circle)**
C. **Use naming patterns in pointcuts**
D. **Make x, y fields private**
E. **Pass changed object to update** **(Lab1E)**
F. **Add a new shape class (Triangle)**
G. **Implement factory enforcement**

To Do: Install AJDT, Open Project

- **Install prepared Eclipse distribution**
  - unzip to appropriate folder
    - /opt under Linux
    - /Developer under Mac OS X
    - arbitrary under Windows
  - startup the Eclipse installation
- **Open Lab1A project**
- **Run target: Test Lab1A**

To Do: Add Circle class

- **Add a new class Circle**
- **The following is a start**

```
class Circle extends Shape {
  private Point center;
  private int   radius;

  ...
}
```

- **Ensure the whole system works properly**
  - Add a new driver DriverB.java
  - Update AllDrivers to call it

To Do: Use Naming Pattern in Pointcuts

- **Modify change() pointcut**
  - use naming pattern rather than enumeration

Wildcards in Pointcuts

weaker than regular expressions!

```
execution(void Point.setX(int))
execution(void Point.getX())
execution(void Point.getY())

execution(* *(..))

execution(public * Point.*(..))
execution(public * Shape+.*(..))
execution(* Shape+.set*(..))
execution(public * *(..))

execution(void Point.get*())

this(Point)
this(graphics.geom.Point)
```

Type Patterns

weaker than regular expressions!

"+" is all sub-types
"*" is wild card
".." is multi-part wild card

```
execution(void Point.setX(int))
execution(void Point.getX())
execution(void Point.getY())

execution(* *(..))

execution(public * Point.*(..))
execution(public * Shape+.*(..))
execution(* Shape+.set*(..))
execution(public * *(..))

execution(void Point.get*())

this(Point)
this(graphics.geom.Point)
```

Signatures

weaker than regular expressions!

"*" is wild card
".." is multi-part wild card

```
execution(void Point.setX(int))
execution(void Point.getX())
execution(void Point.getY())

execution(* *(..))

execution(public * Point.*(..))
execution(public * Shape+.*(..))
execution(* Shape+.set*(..))
execution(public * *(..))

execution(void Point.get*())

this(Point)
this(graphics.geom.Point)
```

Pointcuts

weaker than regular expressions!

"+" is all sub-types
"*" is wild card
".." is multi-part wild card

```
execution(void Point.setX(int))
execution(void Point.getX())
execution(void Point.getY())

execution(* *(..))

execution(public * Point.*(..))
execution(public * Shape+.*(..))
execution(* Shape+.set*(..))
execution(public * *(..))

execution(void Point.get*())

this(Point)
this(graphics.geom.Point)
```

Property-Based Crosscutting

```
package com.xerox.pri
public class C1 {
  …
  public void foo() {
    A.doSomething(…);
    …
  }
  …
}
```

```
package com.xerox.scan;
public class C2 {
  …
  public int frotz()
    A.doSomething(…);
    …
  }
  public int bar() {
    A.doSomething(…);
    …
  }
  …
}
```

```
package com.xerox.copy;
public class C3 {
  …
  public String s1() {
    A.doSomething(…);
    …
  }
  …
}
```

- **Methods with a common property**
  - public/private, return a certain value, in a particular package
- **Logging, debugging, profiling**
  - log on entry to every public method

Property-Based Crosscutting

```
aspect PublicErrorLogging {

  Log log = new Log();

  pointcut publicInterface():
    call(public * org.eclipse..*.*(..));

  after() throwing (Error e): publicInterface() {
    log.write(e);
  }
}
```

Consider code maintenance

- **Another programmer adds a public method**
    - i.e. extends public interface – this code will still work
- **Another programmer reads this code**
    - “what’s really going on” is explicit

What Does *Property-Based* Mean

- **Pointcut stability**
  - what happens as the program evolves
- **Say what matters, not what doesn't**
  - types of args
  - number of args
  - name
  - defining type
- **But be specific enough**
  - don't say call(* set*(..))

Discussion

- **Questions:**
  - is this a good idea?
  - what if you wanted to change it back?
  - what if you wanted to stop using an aspect?
  - a tool for modeling/design/development?

To Do: Make x, y Fields Private

- **x and y fields of Point are package public**
- **Make them private**
- **But first, save a log of running the drivers**

- **There’s something tricky here…**

Other Primitive Pointcuts

`cflow(`*`pointcut designator`*`)`

all join points within the dynamic control flow of any join point in *pointcut designator*

`cflowbelow(`*`pointcut designator`*`)`

all join points within the dynamic control flow below any join point in *pointcut designator*

only top-level changes

DisplayUpdating v4

```
aspect DisplayUpdating {

  pointcut change():
    execution(void Shape.moveBy(int, int)) ||
    execution(void Shape+.set*(*));

  pointcut topLevelchange():
    change() && !cflowbelow(change());

  after() returning: topLevelchange() {
    Display.update();
  }
}
```

To Do: Pass Changed Object to update

- Pass the object that has changed
- Add a parameter to Display.update
- Pass the right object

Values at Join Points

demonstrate first, explain in detail afterwards

- pointcut can explicitly expose certain values
- advice can use explicitly exposed values

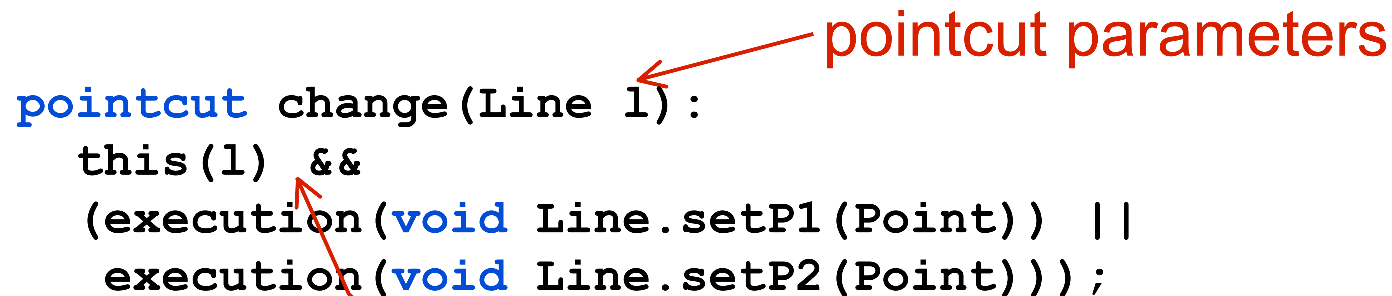
```
pointcut change(Shape shape):
  this(shape) &&
  (execution(void Shape.moveBy(int, int)) ||
   execution(void Shape+.set*(*)));

after(Shape s) returning: change(s) {
  <s is bound to the shape>
}
```

Values at Join Points

of user-defined pointcut designator

- **variable is bound by user-defined pointcut declaration**
  - pointcut supplies value for variable
  - value is available to all users of user-defined pointcut



typed variable in place of type name

```
after(Line line): change(line) {
  <line is bound to the line>
}
```

Values at Join Points

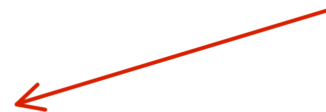
of advice

- **variable is bound by advice declaration**
  - pointcut supplies value for variable
  - value is available in advice body

```
pointcut change(Line l):
  this(l) &&
  (execution(void Line.setP1(Point)) ||
   execution(void Line.setP2(Point)));
```

advice parameters

typed variable in place of type name



```
after(Line line): change(line) {
  <line is bound to the line>
}
```

Values at Join Points

- **value is ‘pulled’**
  - right to left across ‘:’ left side : right side
  - from pointcuts to user-defined pointcuts
  - from pointcuts to advice, and then advice body

```
pointcut change(Line l):
   this(l) &&
   (execution(void Line.setP1(Point)) ||
    execution(void Line.setP2(Point)));
```

```
after(Line line): change(line) {
   <line is bound to the line>
}
```

Values at Join Points

- **value is ‘pulled’**
  - right to left across ‘**:**’   left side : right side
  - from pointcuts to user-defined pointcuts
  - from pointcuts to advice, and then advice body

```
pointcut change(Object changer, Line changee):
  this(changer) && target(changee) &&
  (call(void Line.setP1(..)) ||
   call(void Line.setP2(..)));
```

```
after(change(Object changer, Line changee)): {
  <line is bound to the line>
}
```

this

primitive pointcut

```
this(<typed var (or type)>)
```

does two things:

- **predicate on join points**
    - **any join point at which**
    - **current object is an instanceof type**
    - **dynamic test**
- **exposes current object**

```
this(Point)
this(Line)
this(l)
```

“any join point” means it matches join points of all kinds

target

primitive pointcut

```
target(<typed var (or type)>)
```

does two things:

- **predicate on join points**
  - **any join point at which**
  - **target object is an instanceof type**
  - **dynamic test**
- **exposes target object**

```
target(Point)
target(Line)
target(Shape)
target(t)
```

“any join point” means it matches join points of all kinds

args

primitive pointcut

```
args(<typed var (or type) 1>, <… 2>, …)
```

does two things:

- **predicate on join points**
  - **any join point at which**
  - **arg is of type / args are of types**
  - **dynamic test**
- **exposes arguments**

```
args(Point)
args(int, int)
args(Point, Point)
args(a, b)
```

“any join point” means it matches join points of all kinds

Idiom For…
getting target object in a polymorphic pointcut

```
this(<supertype name>) &&
```

- does not further restrict the join points
- does pick up the target object

```
pointcut change(Shape shape):
   this(shape) &&
   (execution(void Shape.moveBy(int, int)) ||
    execution(void Shape+.set*(*));

after(Shape s) returning: change(s) {
   <s is bound to the shape>
}
```

Pointcuts

can expose values at join points

```
pointcut change(Shape s):
  this(s) &&
  (execution(void Shape.moveBy(int, int)) ||
   execution(void Shape+.set*(*)));

pointcut topLevelchange(Shape s):
  change(s) && !cflowbelow(change(Shape));
```

Other Primitive Pointcuts

`call(void Point.setX(int))`

method/constructor execution join points (actual running method)

`initialization(Point)`

object initialization join points

`staticinitialization(Point)`

class initialization join points (as the class is loaded)

Other Primitive Pointcuts

```
this(<type name>)
within(<class or aspect name>)
withincode(<method/constructor signature>)
```

any join point at which
- **currently executing object is an instance of type or class name**
- **currently executing code is contained within class name**
- **currently executing code is specified method or constructor**

```
get(int Point.x)
set(int Point.x)
```

field reference or assignment join points

Pointcuts

Name	JP Kinds	Static test	Dynamic test
call, execution	method/constructor call/execution	signature matches	none
get, set	field get/set	signature matches	none
handler	exception handler execution	exception matches	none
initialization staticinitialization	class (static)initializer execution	class name matches	
within, withincode	any	shadow is within matching class or member body	none
cflow, cflowbelow	any		within control flow of JP matched by contained pointcut
this	any		current object instanceof type
target	any		target object instance of type
args	any		arguments instanceof type

To Do: Add Triangle class

- **Add a new class Triangle**
- **The following is a start**

```
class Triangle extends Shape {
  private Point p1;
  private Point p2;
  private Point p3;
  ...
}
```

- **Ensure the whole system works properly**
  - Add a new driver DriverF.java

To do: Study Tracing

```
public aspect Tracing {
  private String indent = "";
  after() returning: execution(void Display.update(..)) {
    System.out.print("Display updated.");
  }
  after() returning: execution(Shape+.new(..)) {
    System.out.println(
      thisJoinPoint.getSignature().toShortString());
  }
  void around(): execution(void Shape+.*(..)) {
    System.out.println();
    System.out.print(indent +
      thisJoinPoint.getSignature().toShortString() + " ");
    String oldIndent = indent;
    indent = indent + "  ";
    proceed();
    indent = oldIndent;
  }
}
```

Special Value

reflective* access to the join point

```
thisJoinPoint.
      Signature  getSignature()
      Object[]   getArgs()
      ...
```

available in any advice

thisJoinPoint is abbreviated to ‘tjp’ in these slides to save slide space

* introspective subset of reflection consistent with Java

Using thisJoinPoint

in highly polymorphic advice

```
aspect PointCoordinateTracing {

  before(Point p, int newVal): set(int Point.*) &&
                               target(p) &&
                               args(newVal) {
    System.out.println("At " +
                       tjp.getSignature() +
                       " field is set to " +
                       newVal +
                       ".");
  }
}
```

using thisJoinPoint makes it possible for the advice to recover information about where it is running

To Do: Enforce Factory Pattern

- **Original design had factory pattern**
- **static Shape.make* methods**
- **no other calls to constructors on Shape+**

Contract Checking

simple example of before/after/around

- **pre-conditions**
  - check whether parameter is valid
- **post-conditions**
  - check whether values were set
- **condition enforcement**
  - force parameters to be valid

Advice is

action to take at dynamic join points

- **before** — **before proceeding at join point**

- **after returning** — **a value to join point**
- **after throwing** — **a throwable to join point**
- **after** — **returning to join point either way**

- **around** — **control over when&if JP proceeds**
  - proceed available in around only
  - thisJoinPoint variable available in all above

- **declare error** — **compile-time error**
- **declare warning** — **compile-time warning**
  - two above can only be used with static pointcuts

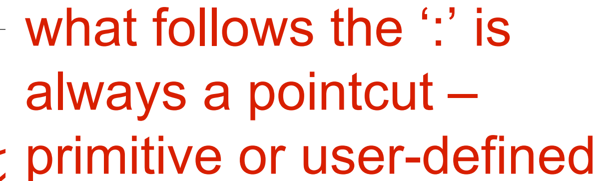
Pre-Condition

using before advice

```
aspect PointBoundsPreCondition {

  before(int newX):
      execution(void Point.setX(int)) && args(newX) {
    check(newX >= MIN_X);
    check(newX <= MAX_X);
  }
  before(int newY):
      execution(void Point.setY(int)) && args(newY) {
    check(newY >= MIN_Y);
    check(newY <= MAX_Y);
  }

  private void check(boolean v) {
    if ( !v )
      throw new RuntimeException();
  }
}
```



Post-Condition

using after advice

```
aspect PointBoundsPostCondition {

  after(Point p, int newX):
      execution(void Point.setX(int))
      && this(p) && args(newX) {
    check(p.getX() == newX);
  }

  after(Point p, int newY):
      execution(void Point.setY(int))
      && this(p) && args(newY) {
    check(p.getY() == newY);
  }

  private void check(boolean v) {
    ...
  }
}
```

Condition Enforcement

using around advice

```
aspect PointBoundsEnforcement {

  void around(Point p, int newX):
      execution(void Point.setX(int))
      && this(p) && args(newX) {
    proceed(p, clip(newX, MIN_X, MAX_X));
  }

  void around(Point p, int newY):
      execution(void Point.setY(int))
      && this(p) && args(newY) {
    proceed(p, clip(newY, MIN_Y, MAX_Y));
  }

  private int clip(int val, int min, int max) {
    return Math.max(min, Math.min(max, val));
  }
}
```

Special Static Method

```
<result type> proceed(arg1, arg2, …)
```

available only in around advice

means “run what would have run if this around advice had not been defined”

result type is result type of advice

Architecture Enforcement

a runtime error

```
class Shape {
  public Line  makeLine(Point p1, Point p2) { new Line...  }
  public Point makePoint(int x, int y)      { new Point... }
  ...
}


aspect FactoryEnforcement {









}
```

ensure that any creation of shapes goes through the factory methods

Architecture Enforcement

a runtime error

```
class Shape {
  public Line  makeLine(Point p1, Point p2) { new Line...  }
  public Point makePoint(int x, int y)      { new Point... }
  ...
}


aspect FactoryEnforcement {





  before(): {
    throw new Error("Call factory to create shapes.");
  }
}
```

ensure that any creation of shapes goes through the factory methods

Architecture Enforcement

a runtime error

```
class Shape {
  public Line  makeLine(Point p1, Point p2) { new Line...  }
  public Point makePoint(int x, int y)      { new Point... }
  ...
}


aspect FactoryEnforcement {

  pointcut newShape():

  pointcut inFactory():

  before(): newShape() && !inFactory(); {
    throw new Error("Call factory to create shapes.");
  }
}
```

ensure that any creation of shapes goes through the factory methods

Architecture Enforcement

a runtime error

```
class Shape {
  public Line  makeLine(Point p1, Point p2) { new Line...  }
  public Point makePoint(int x, int y)      { new Point... }
  ...
}

aspect FactoryEnforcement {

  pointcut newShape(): call(Shape+.new(..));

  pointcut inFactory(): withincode(* Shape.make*(..));

  before(): newShape() && !inFactory(); {
    throw new Error("Call factory to create shapes.");
  }
}
```

ensure that any creation of shapes goes through the factory methods

Architecture Enforcement

a compile-time error

```
class Shape {
  public Line  makeLine(Point p1, Point p2) { new Line...  }
  public Point makePoint(int x, int y)      { new Point... }
  ...
}

aspect FactoryEnforcement {

  pointcut newShape(): call(Shape+.new(..));

  pointcut inFactory(): withincode(* Shape.make*(..));

  declare error: newShape() && !inFactory():
    "Call factory to create shapes.";

}
```

ensure that any creation of shapes goes through the factory methods