Aspect-Oriented Programming with AspectJ®
eclipse.org/aspectj

Andy Clement
IBM Java Technology
AspectJ/AJDT Committer

Mik Kersten
UBC
AspectJ/AJDT Committer

© 2005 IBM Corporation and PARC; made available under the CPL v1.0 | February, 2005 | Burlingame, California
I  AOP and AspectJ overview
   – motivation for AOP
   – problems, basic concepts, context

II  AspectJ tutorial
   – first example
   – language mechanisms
   – using aspects
   – using the eclipse tools

III  Hands on exercises

IV  Wrap up
good modularity

XML parsing

- XML parsing in org.apache.tomcat
  - red shows relevant lines of code
  - nicely fits in one box
good modularity

URL pattern matching

- **URL pattern matching in org.apache.tomcat**
  - red shows relevant lines of code
  - nicely fits in two boxes (using inheritance)
problems like...

logging is not modularized

- where is logging in org.apache.tomcat
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places
the problem of crosscutting concerns

- critical aspects of large systems don’t fit in traditional modules
  - logging, error handling
  - synchronization
  - security
  - memory management
  - performance optimizations

- tangled code has a cost
  - difficult to understand
  - difficult to change
  - increases with size of system
  - maintenance costs are huge

- good programmers work hard to get rid of tangled code
  - the last 10% of the tangled code causes 90% of the development and maintenance headaches
the AOP idea

aspect-oriented programming

• crosscutting is inherent in complex systems
• crosscutting concerns
  – have a clear purpose
  – have a natural structure
    • defined set of methods, module boundary crossings, points of resource utilization, lines of dataflow…
• so, let’s capture the structure of crosscutting concerns explicitly...
  – in a modular way
  – with linguistic and tool support
• aspects are
  – well-modularized crosscutting concerns
• Aspect-Oriented Software Development: AO support throughout lifecycle
this tutorial is about...

- **using AOP and AspectJ to:**
  - improve the modularity of crosscutting concerns
    - design modularity
    - source code modularity
    - development process
- **aspects are two things:**
  - concerns that crosscut [design level]
  - a programming construct [implementation level]
    - enables crosscutting concerns to be captured in modular units
- **AspectJ is:**
  - an aspect-oriented extension to Java™ that supports general-purpose aspect-oriented programming
AspectJ is…

- a small and well-integrated extension to Java™
  - outputs .class files compatible with any JVM
  - all Java programs are AspectJ programs
- a general-purpose AO language
  - just as Java is a general-purpose OO language
- includes IDE support
  - emacs, JBuilder, Forte 4J, Eclipse
- freely available implementation
  - compiler is Open Source
- active user community
  - aspectj-users@eclipse.org
AspectJ applied to a large middleware system

- java code base with 10,000 files and 500 developers
- AspectJ captured logging, error handling, and profiling policies
  - Packaged as extension to Java language
  - Compatible with existing code base and platform

<table>
<thead>
<tr>
<th>existing policy implementations</th>
<th>policies implemented with AspectJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>affect every file</td>
<td>one reusable crosscutting module</td>
</tr>
<tr>
<td>- 5-30 page policy documents</td>
<td>- policy captured explicitly</td>
</tr>
<tr>
<td>- applied by developers</td>
<td>- applies policy uniformly for all time</td>
</tr>
<tr>
<td>affect every developer</td>
<td>written by central team</td>
</tr>
<tr>
<td>- must understand policy document</td>
<td>- no burden on other 492 developers</td>
</tr>
<tr>
<td>repeat for new code assets</td>
<td>automatically applied to new code</td>
</tr>
<tr>
<td>awkward to support variants</td>
<td>easy plug and unplug</td>
</tr>
<tr>
<td>- complicates product line</td>
<td>- simplifies product line issues</td>
</tr>
<tr>
<td>don’t even think about</td>
<td>changes to policy happen in one place</td>
</tr>
</tbody>
</table>
| changing the policy
Part II

tutorial
language mechanisms

• **goal: present basic mechanisms**
  – using one simple example
    • emphasis on what the mechanisms do
    • small scale motivation

• **later**
  – environment, tools
  – larger examples, design and SE issues
basic mechanisms

• **1 overlay onto Java**
  – dynamic join points
    • “points in the execution” of Java programs

• **4 small additions to Java**
  – pointcuts
    • pick out join points and values at those points
      – primitive, user-defined pointcuts
  – advice
    • additional action to take at join points in a pointcut
  – inter-type declarations
  – aspect
    • a modular unit of crosscutting behavior
      – comprised of advice, inter-type, pointcut, field, constructor, and method declarations
a simple figure editor

- **Display**
- **Figure**
  - makePoint(..)
  - makeLine(..)
- **FigureElement**
  - moveBy(int, int)
- **Point**
  - getX()
  - getY()
  - setX(int)
  - setY(int)
  - moveBy(int, int)
- **Line**
  - getP1()
  - getP2()
  - setP1(Point)
  - setP2(Point)
  - moveBy(int, int)

Factory methods and operations that move elements.
a simple figure editor

class Line implements FigureElement{
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
    void moveBy(int dx, int dy) { ... }
}

class Point implements FigureElement {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }
    void moveBy(int dx, int dy) { ... }
}
display updating

- collection of figure elements
  - that move periodically
  - must refresh the display as needed
  - complex collection
  - asynchronous events

we will initially assume just a single display
join points

key points in dynamic call graph

imagine `l.moveBy(2, 2)`
join point terminology

- several kinds of join points
  - method & constructor call
  - method & constructor execution
  - field get & set
  - exception handler execution
  - static & dynamic initialization
join point terminology

key points in dynamic call graph

imagine `l.moveBy(2, 2)`

all join points on this slide are within the control flow of this join point
primitive pointcuts
“a means of identifying join points”

A pointcut is a kind of predicate on join points that:
- can match or not match any given join point and
- optionally, can pull out some of the values at that join point

\texttt{call(void Line.setP1(Point))}

matches if the join point is a method call with this signature
pointcut composition

pointcuts compose like predicates, using &&, || and !

- a "void Line.setP1(Point)" call
- or
- call(void Line.setP1(Point)) || call(void Line.setP2(Point));

- a "void Line.setP2(Point)" call

whenever a Line receives a "void setP1(Point)" or "void setP2(Point)" method call
user-defined pointcuts
defined using the pointcut construct

user-defined (aka named) pointcuts
– can be used in the same way as primitive pointcuts

name parameters

```java
pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));
```

more on parameters and how pointcut can expose values at join points in a few slides
pointcuts

```java
pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));
```

user-defined pointcut

primitive pointcut, can also be:
- `call, execution`      - `this, target`
- `get, set`              - `within, withincode`
- `handler`              - `cflow, cflowbelow`
- `initialization, staticinitialization`
**after advice**

**action to take after computation under join points**

**after advice runs**

“on the way back out”

```java
pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

after() returning: move() {
    <code here runs after each move>
}
```
a simple aspect

DisplayUpdating v1

an aspect defines a special class that can crosscut other classes

```java
aspect DisplayUpdating {

  pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

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

box means complete running code
without AspectJ

DisplayUpdating v1

class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update();
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update();
  }
}

• what you would expect
  – update calls are tangled through the code
  – “what is going on” is less explicit
pointcuts
can cut across multiple classes

pointcut move():
call(void Line.setP1(Point)) ||
call(void Line.setP2(Point)) ||
call(void Point.setX(int))   ||
call(void Point.setY(int));
pointcut move():
    call(void FigureElement.moveBy(int, int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int));
aspect DisplayUpdating { 

    pointcut move(): 
        call(void FigureElement.moveBy(int, int)) ||
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int));

    after() returning: move() { 
        Display.update(); 
    } 
}
using values at join points
demonstrate first, explain in detail afterwards

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

```java
pointcut move(FigureElement figElt):
  target(figElt) &&
  (call(void FigureElement.moveBy(int, int)) ||
   call(void Line.setP1(Point)) ||
   call(void Line.setP2(Point)) ||
   call(void Point.setX(int)) ||
   call(void Point.setY(int)));

after(FigureElement fe) returning: move(fe) {
  <fe is bound to the figure element>
}
```
explaining parameters... 
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

```java
pointcut move(Line l):
    target(l) &&
    (call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)));
```

```java
after(Line line) returning: move(line) {
    <line is bound to the line>
}
```
explaining parameters... of advice

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

```java
pointcut move(Line l):
    target(l) &&
    (call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)));

advice parameters
```
```java
after(Line line) returning: move(line) {
    <line is bound to the line>
}
```
explaining parameters...

- 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

```java
pointcut move(Line l):
  target(l) &&
  (call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point)));

after(Line line) returning: move(line) {
  <line is bound to the line>
}
```
target

primitive pointcut designator

target( TypeName | FormalReference )

does two things:
- exposes target
- predicate on join points - any join point at which target object is an instance of type name (a dynamic test)

target(Point)
target(Line)
target(FigureElement)

“any join point” means it matches join points of all kinds
- method call join points
- method & constructor execution join points
- field get & set join points
- dynamic initialization join points
idiom for...

going target object in a polymorphic pointcut

target( SupertypeName )

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

pointcut move(FigureElement figElt):
  target(figElt) &&
  (call(void Line.setP1(Point)) ||
   call(void Line.setP2(Point)) ||
   call(void Point.setX(int)) ||
   call(void Point.setY(int)));

after(FigureElement fe) returning: move(fe) {
  <fe is bound to the figure element>
}
pointcuts

can expose values at join points

pointcut move(FigureElement figElt):
  target(figElt) &&
  (call(void FigureElement.moveBy(int, int)) ||
   call(void Line.setP1(Point)) ||
   call(void Line.setP2(Point)) ||
   call(void Point.setX(int)) ||
   call(void Point.setY(int)));

© 2005 IBM Corporation and PARC; made available under the CPL v1.0 | February, 2005 | Burlingame, California
aspect DisplayUpdating {
  pointcut move(FigureElement figElt):
    target(figElt) &&
    (call(void FigureElement.moveBy(int, int)) ||
     call(void Line.setP1(Point)) ||
     call(void Line.setP2(Point)) ||
     call(void Point.setX(int)) ||
     call(void Point.setY(int)));

  after(FigureElement fe) returning: move(fe) {
    Display.update(fe);
  }
}
without AspectJ

class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) {
        this.p1 = p1;
    }
    void setP2(Point p2) {
        this.p2 = p2;
    }

    void moveBy(int dx, int dy) { ... }
}

class Point {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) {
        this.x = x;
    }
    void setY(int y) {
        this.y = y;
    }

    void moveBy(int dx, int dy) { ... }
}
without AspectJ

class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) {
        this.p1 = p1;
        Display.update();
    }
    void setP2(Point p2) {
        this.p2 = p2;
        Display.update();
    }
    void moveBy(int dx, int dy) { ... }
}

class Point {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) {
        this.x = x;
    }
    void setY(int y) {
        this.y = y;
    }
    void moveBy(int dx, int dy) { ... }
}
without AspectJ

DisplayUpdating v2

class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) {
        this.p1 = p1;
        Display.update();
    }
    void setP2(Point p2) {
        this.p2 = p2;
        Display.update();
    }
    void moveBy(int dx, int dy) { ... }
}

class Point {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) {
        this.x = x;
        Display.update();
    }
    void setY(int y) {
        this.y = y;
        Display.update();
    }
    void moveBy(int dx, int dy) { ... }
}
without AspectJ

DisplayUpdating v3

```java
class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) {
        this.p1 = p1;
        Display.update(this);
    }
    void setP2(Point p2) {
        this.p2 = p2;
        Display.update(this);
    }
    void moveBy(int dx, int dy) { ... }
}

class Point {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) {
        this.x = x;
        Display.update(this);
    }
    void setY(int y) {
        this.y = y;
        Display.update(this);
    }
    void moveBy(int dx, int dy) { ... }
}
```

- **no locus of “display updating”**
  - evolution is cumbersome
  - changes in all classes
  - have to track & change all callers
with AspectJ

```java
class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) {
        this.p1 = p1;
    }
    void setP2(Point p2) {
        this.p2 = p2;
    }

    void moveBy(int dx, int dy) { ... }
}

class Point {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) {
        this.x = x;
    }
    void setY(int y) {
        this.y = y;
    }

    void moveBy(int dx, int dy) { ... }
}
```
with AspectJ

```java
class Line {
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) {
        this.p1 = p1;
    }
    void setP2(Point p2) {
        this.p2 = p2;
    }
    void moveBy(int dx, int dy) {
    }
}

class Point {
    private int x = 0, y = 0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) {
        this.x = x;
    }
    void setY(int y) {
        this.y = y;
    }
    void moveBy(int dx, int dy) {
    }
}

aspect DisplayUpdating {
    pointcut move():
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point));
    after() returning: move() {
        Display.update();
    }
}
```
with AspectJ

DisplayUpdating v2

class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void setP1(Point p1) {
        this.p1 = p1;
    }
    void setP2(Point p2) {
        this.p2 = p2;
    }

    void moveBy(int dx, int dy) { ... }
}

class Point {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void setX(int x) {
        this.x = x;
    }
    void setY(int y) {
        this.y = y;
    }

    void moveBy(int dx, int dy) { ... }
}

aspect DisplayUpdating {

    pointcut move():
        call(void FigureElement.moveBy(int, int)) ||
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int));

    after() returning: move() {
        Display.update();
    }
}
class Line {
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
  void moveBy(int dx, int dy) { ... }
}

class Point {
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
  void moveBy(int dx, int dy) { ... }
}

aspect DisplayUpdating {
  pointcut move(FigureElement figElt):
    target(figElt) &&
    (call(void FigureElement.moveBy(int, int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)));

  after(FigureElement fe) returning: move(fe) {
    Display.update(fe);
  }
}

• clear display updating module
  – all changes in single aspect
  – evolution is modular
aspects crosscut classes

aspect modularity cuts across class modularity

Display

<table>
<thead>
<tr>
<th>Figure</th>
<th>FigureElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>makePoint(..)</td>
<td>moveBy(int, int)</td>
</tr>
<tr>
<td>makeLine(..)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Point</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>getX()</td>
<td>getP1()</td>
</tr>
<tr>
<td>getY()</td>
<td>getP2()</td>
</tr>
<tr>
<td>setX(int)</td>
<td>setP1(Point)</td>
</tr>
<tr>
<td>setY(int)</td>
<td>setP2(Point)</td>
</tr>
<tr>
<td>moveBy(int, int)</td>
<td>moveBy(int, int)</td>
</tr>
</tbody>
</table>

DisplayUpdating
advice is additional action to take at join points

- **before** before proceeding at join point
- **after returning** a value at join point
- **after throwing** a throwable at join point
- **after** returning at join point either way
- **around** on arrival at join point gets explicit control over when&if program proceeds
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
pre-condition using before advice

aspect PointBoundsPreCondition {

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

what follows the ':' is always a pointcut – primitive or user-defined
aspect PointBoundsPostCondition {

    after(Point p, int newX) returning:
        call(void Point.setX(int)) && target(p) && args(newX) {
            assert p.getX() == newX;
        }

    after(Point p, int newY) returning:
        call(void Point.setY(int)) && target(p) && args(newY) {
            assert p.getY() == newY;
        }
}

post-condition using after advice
aspect PointBoundsEnforcement {

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

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

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

for each around advice with the signature

Return\texttt{Type} around(\texttt{T1} \texttt{arg1}, \texttt{T2} \texttt{arg2}, ...)

there is a special method with the signature

Return\texttt{Type} proceed(\texttt{T1}, \texttt{T2}, ...)

available only in around advice

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

using around advice

```java
aspect PointCaching {

    private MyLookupTable cache = new MyLookupTable();

    Point around(int x, int y):
        call(Point.new(int, int)) && args(x, y) {
            Point ret = cache.lookup(x, y);
            if (ret == null) {
                ret = proceed(x, y);
                cache.add(x, y, ret);
            }
            return ret;
        }
}
```
property-based crosscutting

- crosscuts of 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

```java
package com.parc.print;
public class C1 {
   ...
   public void foo() {
       A.doSomething(…);
   }
   ...
}
package com.parc.scan;
public class C2 {
   ...
   public int frotz() {
       A.doSomething(…);
   }
   public int bar() {
       A.doSomething(…);
   }
}
package com.parc.copy;
public class C3 {
   ...
   public String s1() {
       A.doSomething(…);
   }
}
```
property-based crosscutting

```java
aspect PublicErrorLogging {

    Logger log = Logger.global;

    pointcut publicInterface():
        call(public * com.bigboxco..*.*(..));

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

neatly captures public interface of my packages

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
wildcarding in pointcuts

```java
target(Point)
target(graphics.geom.Point)
target(graphics.geom. *)
target(graphics..*)
```

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

```
call(void Point.setX(int))
call(void Point.setY(*))
call(void Point.set*(*))
call(void set*(*))
```

```
call(Point.new(int, int))
call(new( ..))
```

“*” is wild card
“..” is multi-part wild card

any type in graphics.geom
any type in any sub-package of graphics

any public method on Point
any public method on any type

any setter

any constructor
special value

reflective* access to the join point

thisJoinPoint.

Signature getSignature()

Object[] getArgs()

...

available in any advice

(also thisJoinPointStaticPart with only the statically determinable portions)

* introspective subset of reflection consistent with Java
using thisJoinPoint

in highly polymorphic advice

aspect PublicErrorLogging {

    Logger log = Logger.global;

    pointcut publicInterface():
        call(public * com.bigboxco..*.*(..));

    after() throwing (Error e): publicInterface() {
        log.throwing(
            tjp.getSignature().getDeclaringType().getName(),
            tjp.getSignature().getName(),
            e);
    }
}

please read as
thisJoinPoint

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

\[
\begin{align*}
\text{this} & \quad (\text{TypeName}) \\
\text{within} & \quad (\text{TypeName}) \\
\text{withincode} & \quad (\text{MemberSignature}) \\
\end{align*}
\]

any join point at which
- currently executing object is an instance of type name
- currently executing code is contained within type name
- currently executing code is specified methods or constructors

\[
\begin{align*}
\text{get} & \quad (\text{int} \:\: \text{Point.x}) \\
\text{set} & \quad (\text{int} \:\: \text{Point.x}) \\
\end{align*}
\]

field reference or assignment join points
fine-grained protection

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

aspect FactoryEnforcement {
    pointcut illegalNewFigElt():
        (call(Point.new(..)) || call(Line.new(..)))
        && !withincode(* Figure.make*(..));

    before(): illegalNewFigElt() {
        throw new Error("Use factory method instead.");
    }
}

want to ensure that any creation of figure elements goes through the factory methods
fine-grained protection

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

aspect FactoryEnforcement {
    pointcut illegalNewFigElt():
        (call(Point.new(..)) || call(Line.new(..)))
        && !withincode(* Figure.make*(..));

    declare error: illegalNewFigElt():
        "Use factory method instead."
    }
}
fine-grained protection

a compile-time error

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

aspect FactoryEnforcement {
    pointcut illegalNewFigElt():
        call(FigureElement+.new(..))
        && !withincode(* Figure.make*(..));

    declare error: illegalNewFigElt():
        "Use factory method instead."
;
}

want to ensure that any creation of figure elements goes through the factory methods

must be a “static pointcut”

all subtypes

© 2005 IBM Corporation and PARC; made available under the CPL v1.0 | February, 2005 | Burlingame, California
fine-grained protection

as a static inner aspect

```java
class Line implements FigureElement{
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
    void moveBy(int dx, int dy) { ... }

    static aspect SetterEnforcement {
        declare error: set(Point Line.*) &&
                        !withincode(void Line.setP*(Point))
                        "Use setter method.";
    }
}
```
fine-grained protection
as a static inner aspect

class Line implements FigureElement{
    private Point p1, p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1(Point p1) { this.p1 = p1; }
    void setP2(Point p2) { this.p2 = p2; }
    void moveBy(int dx, int dy) { ... }

    static aspect SetterEnforcement {
        declare error: set(Point Line.*) &&
        !withincode(void Line.setP*(Point))
        "Use setter method, even inside Line class."
    }
}
other primitive pointcuts

execution(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

cflow( Pointcut )
all join points in the dynamic control flow of any
join point picked out by Pointcut

cflowbelow( Pointcut )
all join points in the dynamic control flow below
any join point picked out by Pointcut
aspect DisplayUpdating {

  pointcut move(FigureElement fe):
    target(fe) &&
    (call(void FigureElement.moveBy(int, int)) ||
     call(void Line.setP1(Point)) ||
     call(void Line.setP2(Point)) ||
     call(void Point.setX(int)) ||
     call(void Point.setY(int)));

  pointcut topLevelMove(FigureElement fe):
    move(fe) && !cflowbelow(move(FigureElement));

  after(FigureElement fe) returning: topLevelMove(fe) {
    Display.update(fe);
  }
}
inter-type declarations

- like member declarations...

```c
long l = 37;
void m() { ... }
```
inter-type declarations

- like member declarations, but with a \textit{TargetType}

```c
long \textit{TargetType}.l = 37;
void \textit{TargetType}.m() { ... }
```
one display per figure element

DisplayUpdating v5

aspect DisplayUpdating {

    private Display FigureElement.display;

    static void setDisplay(FigureElement fe, Display d) {
        fe.display = d;
    }

    pointcut move(FigureElement figElt):
        <as before>;

    after(FigureElement fe): move(fe) {
        fe.display.update(fe);
    }
}
field/getter/setter idiom

```java
aspect DisplayUpdating {
    private Display FigureElement.display;

    public static void setDisplay(FigureElement fe, Display d) {
        fe.display = d;
    }

    pointcut move(FigureElement figElt): <as before>
    after(FigureElement fe): move(fe) {
        fe.display.update(fe);
    }
}
```

**the display field**
- is a field in objects of type `FigureElement`, but
- belongs to `DisplayUpdating` aspect
- `DisplayUpdating` should provide getter/setter (called by setup code)
one-to-many

DisplayUpdating v6

aspect DisplayUpdating {

    private List FigureElement.displays = new LinkedList();

    public static void addDisplay(FigureElement fe, Display d) {
        fe.displays.add(d);
    }
    public static void removeDisplay(FigureElement fe, Display d) {
        fe.displays.remove(d);
    }

    pointcut move(FigureElement figElt):
        <as before>;

    after(FigureElement fe): move(fe) {
        Iterator iter = fe.displays.iterator();
        ...
    }
}
inheritance & specialization

- **pointcuts can have additional advice**
  - aspect with
    - concrete pointcut
    - perhaps no advice on the pointcut
  - in figure editor
    - `move()` can have advice from multiple aspects
  - module can expose certain well-defined pointcuts

- **abstract pointcuts can be specialized**
  - aspect with
    - abstract pointcut
    - concrete advice on the abstract pointcut
role types and reusable aspects

```java
abstract aspect Observing {
    protected interface Subject {
    }
    protected interface Observer {
    }

    private List Subject.observers = new ArrayList();
    public void addObserver(Subject s, Observer o) {
    }
    public void removeObserver(Subject s, Observer o) {
    }
    public static List getObservers(Subject s) {
    }

    abstract pointcut changes(Subject s);

    after(Subject s): changes(s) {
        Iterator iter = getObservers(s).iterator();
        while ( iter.hasNext() ) {
            notifyObserver(s, ((Observer)iter.next()));
        }
    }
    abstract void notifyObserver(Subject s, Observer o);
}
```
aspect DisplayUpdating extends Observing {

    declare parents: FigureElement implements Subject;
    declare parents: Display implements Observer;

    pointcut changes(Subject s):
        target(s) &&
        (call(void FigureElement.moveBy(int, int)) ||
        call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int)));

    void notifyObserver(Subject s, Observer o) {
        ((Display)o).update(s);
    }
}
advice precedence

- what happens if two pieces of advice apply to the same join point?

```java
aspect Security {
    before(): call(public *(..)) {
        if (!Policy.isAllowed(tjp))
            throw new SecurityExn();
    }
}

aspect Logging {
    before(): logged() {
        System.err.println("Entering " + tjp);
    }
    pointcut logged():
        call(void troublesomeMethod());
}
```
advice precedence

- order is undefined, unless...
  - in the same aspect,
  - in subaspect, or
  - using declare precedence...

```java
aspect Security {
    before(): call(public *(..)) {
        if (!Policy.isAllowed(tjp))
            throw new SecurityExn();
    }
    declare precedence: Security, *;
}

aspect Logging {
    before(): logged() {
        System.err.println("Entering " + tjp);
    }
    pointcut logged():
        call(void troublesomeMethod());
}
```
**Summary**

**Join Points**
- Method & constructor call
- Execution
- Field
- Get
- Set
- Exception handler execution
- Initialization

**Pointcutes**
- **Primitive**
  - Field
  - Method
  - Call
  - Execution
  - Handler
- **User-defined**
  - Pointcut declaration
  - Initialization
  - Target
  - Args
  - Exception handler
  - Execution
  - Within
  - Within code
  - Cflow

**Advice**
- Before
- After
- Around
- Inter-type decls
  - Type.field
  - Type.method()

**Declare**
- Error
- Parents
- Precedence

**Reflection**
- ThisJoinPoint
- ThisJoinPointStaticPart
Part III

Hands on exercises
Part IV

Wrap up
AspectJ technology

• **AspectJ is a small extension to Java**
  – valid Java programs are also valid AspectJ programs

• **AspectJ has its own compiler, ajc**
  – runs on Java 2 platform (Java 1.3 or later)
  – produces Java platform-compatible .class files
    (Java 1.1 - 1.5)

• **AspectJ tools support**
  – IDE extensions: Emacs, JBuilder, Forte4J, Eclipse
  – ant tasks
  – works with existing debuggers

• **license**
  – compiler, runtime and tools are Open Source and free for any use
when are aspects appropriate?

- is there a concern that:
  - crosscuts the structure of several objects or operations
  - is beneficial to separate out
... crosscutting

- a design concern that involves several objects or operations
- implemented without AOP would lead to distant places in the code that
  - do the same thing
    - e.g. traceEntry("Point.set")
  - do a coordinated single thing
    - e.g. timing, observer pattern
    - harder to find these
… beneficial to separate out

- exactly the same questions as for objects
- does it improve the code in real ways?
  - separation of concerns
    - e.g. think about service without timing
  - clarifies interactions, reduces tangling
    - e.g. all the traceEntry are really the same
  - easier to modify / extend
    - e.g. change the implementation of tracing
    - e.g. abstract aspect reuse
  - plug and play
    - e.g. tracing aspects unpluged but not deleted
expected benefits of using AOP

- good modularity, even in the presence of crosscutting concerns
  - less tangled code, more natural code, smaller code
  - easier maintenance and evolution
    - easier to reason about, debug, change
  - more reusable
    - more possibilities for plug and play
    - abstract aspects
Adopting AspectJ

- exploration enforcement
- auxiliary / infrastructure
- core / business
- AO Analysis, AO Design, AO Strategy

reward vs. time & confidence
The books...
AspectJ on the web

- [http://eclipse.org/aspectj](http://eclipse.org/aspectj)

- **AJDT**
  - [http://eclipse.org/ajdt](http://eclipse.org/ajdt)

- **Contact us:**
  - Andy Clement  clemas@uk.ibm.com
  - Mik Kersten  beatmik@cs.ubc.ca
credits

AspectJ is an* Eclipse project

with notable work by

Ron Bodkin, Andy Clement, Adrian Colyer,
Erik Hilsdale, Jim Hugunin, Wes Isberg, Mik Kersten,
Gregor Kiczales

slides, compiler, tools & documentation are available at
eclipse.org/aspectj

* Originally developed at PARC, with support from NIST and DARPA.