Aspect-Oriented Programming with AspectJ™

Erik Hilsdale
and
Mik Kersten
eclipse.org/aspectj

Copyright is held by the author/owner(s).
OOPSLA ’04, October 24-28, 2004, Vancouver, British Columbia, Canada
2004 ACM 048510

outline

• I AOP and AspectJ overview
  – problems, basic concepts, context
• II AspectJ tutorial
  – first example
  – language mechanisms
  – using aspects
• III examples and demo
• IV conclusion
good modularity

XML parsing

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

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

problems like... 
session expiration is not modularized
problems like...

session tracking is not modularized

HTTPRequest
- getCookies()
- getRequestMethod()
- getSession()
- getRequestedSessionId()

HTTPResponse
- getRequestMethod()
- setContentType(contentType)
- getOutputStream()
- setSessionId(id)

SessionInterceptor
- requestMap(request)
- beforeBody(req, resp)

Session
- getAttribute(name)
- setAttribute(name, val)
- invalidate()

Servlet

the problem of crosscutting concerns

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

- 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

dis t r i b uted i n d e x

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
**language support to...**

**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>
<tr>
<td>changing the policy</td>
<td></td>
</tr>
</tbody>
</table>

**looking ahead**

- problem structure
- examples:
  - crosscutting in the design, and how to use AspectJ to capture that
- AspectJ language

**language mechanisms:**
- crosscutting in the code mechanisms AspectJ provides
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 (aka "open classes")
  - aspect
    - a modular unit of crosscutting behavior
      - comprised of advice, inter-type, pointcut, field, constructor, and method declarations

a simple figure editor

```
Point
-getX()
-getY()
-setX(int)
-setY(int)
-moveBy(int, int)

Figure
-makePoint(..)
-makeLine(..)

FigureElement
-moveBy(int, int)

Display

Operation
-Factory methods
-Operations that move elements
```
a simple figure editor

```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) { ... }
}

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

- other examples
  - session liveness
  - value caching

we will initially assume just a single display
join points

key points in dynamic call graph

imagine `l.moveBy(2, 2)`

- a Line
  - dispatch
  - a method call returning or throwing
- a Point
  - dispatch
  - a method execution returning or throwing
  - a method execution returning or throwing

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

`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
- call(\texttt{void Line.setP1(Point)}) || call(\texttt{void Line.setP2(Point)});
- A "void Line.setP2(Point)" call
- whenever a Line receives a "\texttt{void setP1(Point)}" or "\texttt{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

\begin{verbatim}
pointcut move():
  call(\texttt{void Line.setP1(Point)}) ||
  call(\texttt{void Line.setP2(Point)});
\end{verbatim}

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

**user-defined pointcut**

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

**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**

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

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

**after advice runs “on the way back out”**
A simple aspect

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

```java
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

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

pointcuts can use interface signatures

```java
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));
```
a multi-class aspect

```java
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

- 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

```
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>
}
```

typed variable in place of type name

explaining parameters...

of advice

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

```
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>
}
```

typed variable in place of type name
### 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)));
```

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

### target

**primitive pointcut designator**

```java
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)

```java
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...
getting target object in a polymorphic pointcut

target(SupertypeName) &&

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

pointcut move(FigureElement figEl): target(figEl) &&
(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>
}
context & multiple classes

DisplayUpdating v3

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) { ... }
}

Aspect-Oriented Programming with AspectJ

without AspectJ

DisplayUpdating v1

```java
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

```java
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

```
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

```
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

**DisplayUpdating v1**

```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();
    }
}
```

**DisplayUpdating v2**

```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(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();
    }
}
```
**Aspect-Oriented Programming with AspectJ**

**with 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) {...}
}
```

**aspect DisplayUpdating**

```java
aspect DisplayUpdating {
    pointcut move(FigureElement figEl){
        target(figEl)
            && (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**

```
Figure

makePoint(..)
makeLine(..)

Point

getX()
getY()
setX(int)
setY(int)
moveBy(int, int)

Line

getP1()
getP2()
setP1(Point)
setP2(Point)
moveBy(int, int)

FigureElement

moveBy(int, int)
```

**aspect modularity cuts across class modularity**
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**

```java
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.

**post-condition**

**using after advice**

```java
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;
        }
}
```
condition enforcement

```
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

```
ReturnType around(T1 arg1, T2 arg2, ...)
```

there is a special method with the signature

```
ReturnType proceed(T1, T2, ...)
```

available only in around advice

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

```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
property-based crosscutting

```java
aspect PublicErrorLogging {
    Logger log = Logger.global;

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

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

wildcarding in pointcuts

- `target(Point)`
- `target(graphics.geom.Point)`
- `target(graphics.geom.*)`
- `target(graphics..*)`  
  - any type in `graphics.geom`
  - any type in any sub-package of `graphics`

- `call(void Point.setX(int))`
- `call(public * Point.*(..))`
- `call(public * *(..))`  
  - any public method on `Point`
  - any public method on any type

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

- `call(Point.new(int, int))`
- `call(new(..))`  
  - any constructor
special value

**reflective* access to the join point**

```java
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

```java
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

```java
this( TypeName )
within( TypeName )
withincode( MemberSignature )

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

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

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

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(Point.new(.)) || call(Line.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

all subtypes

must be a “static pointcut”

must be a “static pointcut”
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.";
    }
}
other primitive pointcuts

```java
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

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

```java
cflowbelow(Pointcut)
all join points in the dynamic control flow below
any join point picked out by Pointcut
```
only top-level moves

```java
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...

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

- like member declarations, but with a *TargetType*

```java
long TargetType.l = 37;
void TargetType.m() { ... }
```

one display per figure element

```java
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 <as before>;
    after(FigureElement fe, Display d) {
        // move(FigureElement figElt):
        //    <as before>;
        //    after(FigureElement fe): move(fe) {
        //        Iterator iter = fe.displays.iterator();
        //        ...
        //    }
    }
}
```

- **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**

```java
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-Oriented Programming with AspectJ

**this is the concrete reuse**

```java
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.isAllwed(tjp))
      throw new SecurityExn();
  }
  declare precedence: Security, *
}

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

summary

<table>
<thead>
<tr>
<th>join points</th>
<th>pointcuts</th>
<th>advice</th>
</tr>
</thead>
<tbody>
<tr>
<td>method &amp; constructor call</td>
<td>primitive- call execution handler</td>
<td>before after around</td>
</tr>
<tr>
<td>field get set exception handler execution initialization</td>
<td>-user-defined- pointcut declaration abstract overriding</td>
<td></td>
</tr>
<tr>
<td>aspects</td>
<td>reflection</td>
<td></td>
</tr>
<tr>
<td>crosscutting type</td>
<td>declare</td>
<td></td>
</tr>
<tr>
<td></td>
<td>error parents precedence</td>
<td></td>
</tr>
<tr>
<td></td>
<td>thisJoinPoint thisJoinPointStaticPart</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inter-type decls</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Type.field Type.method()</td>
<td></td>
</tr>
</tbody>
</table>

- primitive- call execution handler
get set initialization this target args within withincode cflow cflowbelow

- user-defined- pointcut declaration abstract overriding
where we have been...
... and where we are going

problem structure

examples:
crosscutting in the design, and
how to use AspectJ to capture that

AspectJ language

language mechanisms:
crosscutting in the code
mechanisms AspectJ provides

using aspects

• **present examples of aspects in design**
  – intuitions for identifying aspects

• **present implementations in AspectJ**
  – how the language support can help
  – putting AspectJ into practice

• **discuss style issues**
  – objects vs. aspects

• **when are aspects appropriate?**
example

plug & play tracing

- **simple tracing**
  - exposes join points and uses very simple advice
- **an unpluggable aspect**
  - core program functionality is unaffected by the aspect

```java
class Point {
    void set(int x, int y) {
        TraceSupport.traceEntry("Point.set");
        this.x = x; this.y = y;
        TraceSupport.traceExit("Point.set");
    }
}
```

```java
class TraceSupport {
    static int TRACELEVEL = 0;
    static protected PrintStream stream = null;
    static protected int callDepth = -1;
    static void init(PrintStream _s) {stream = _s;}
    static void traceEntry(String str) {
        if (TRACELEVEL == 0) return;
        callDepth++;
        printEntering(str);
    }
    static void traceExit(String str) {
        if (TRACELEVEL == 0) return;
        callDepth--;
        printExiting(str);
    }
}
```
a clear crosscutting structure

all modules of the system use the trace facility in a consistent way: entering the methods and exiting the methods.

this line is about interacting with the trace facility

tracing as an aspect

aspect PointTracing {
  pointcut trace():
    within(com.bigboxco.boxes.*) &&
    execution(* *(..));

  before(): trace() {
    TraceSupport.traceEntry(tjp);
  }

  after(): trace() {
    TraceSupport.traceExit(tjp);
  }
}
**plug and debug**

- **plug in:**  
  - `ajc Point.java Line.java`  
  - `TraceSupport.java PointTracing.java`

- **unplug:**  
  - `ajc Point.java Line.java`

- **or...**

```java
// From ContextManager
public void service (Request request, Response response) {
    System.out.println("A");
    request.setContextManager(this);
    request.setResponse(response);
    response.setRequest(request);
    System.out.println("B");
    int status = response.getStatus();
    if (status < 400) {
        status = processRequest(request);
        if (status == 0){
            request.getWrapper().handleRequest(request, response);
        } else {
            // something went wrong
            handleError(request, response, null, status);
        }
    }
    catch (Throwable t) {
        log("Error closing request "+ ex);
    }
    // log("Done with request " + request);
    // System.out.println("C");
}
```

---

**OOPLA '04**
plug and debug

- turn debugging on/off without editing classes
- debugging disabled with no runtime cost
- can save debugging code between uses
- can be used for profiling, logging
- easy to be sure it is off

aspects in the design

- objects are no longer responsible for using the trace facility
  - trace aspect encapsulates that responsibility, for appropriate objects

- if the Trace interface changes, that change is shielded from the objects
  - only the trace aspect is affected

- removing tracing from the design is trivial
  - just remove the trace aspect
## Aspects in the Code

<table>
<thead>
<tr>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object code contains no calls to trace functions</td>
</tr>
<tr>
<td>- trace aspect code encapsulates those calls, for appropriate objects</td>
</tr>
<tr>
<td>If the Trace interface changes, there is no need to modify the object classes</td>
</tr>
<tr>
<td>- only the trace aspect class needs to be modified</td>
</tr>
<tr>
<td>Removing tracing from the application is trivial</td>
</tr>
<tr>
<td>- compile without the trace aspect class</td>
</tr>
</tbody>
</table>

## Tracing: Object vs. Aspect

<table>
<thead>
<tr>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using an object captures tracing support, but does not capture its consistent usage by other objects</td>
</tr>
<tr>
<td>Using an aspect captures the consistent usage of the tracing support by the objects</td>
</tr>
</tbody>
</table>

![Diagram showing object vs. aspect tracing]
Using a library aspect

```java
aspect BigBoxCoTracing {
    pointcut trace():
        within(com.bigboxco.*)
        && execution(* *(.));

    before(): trace() {
        TraceSupport.traceEntry(tjp);
    }

    after(): trace() {
        TraceSupport.traceExit(tjp);
    }
}
```

Example

Context-passing aspects

Workers need to know the caller:
- capabilities
- charge backs
- to customize result
context-passing aspects

workers need to know the caller:
- capabilities
- charge backs
- to customize result

pointcut invocations(Caller c):
    this(c) && call(void Service.doService(String));
context-passing aspects

```java
pointcut invocations(Caller c):
    this(c) && call(void Service.doService(String));

pointcut workPoints(Worker w):
    target(w) && call(void Worker.doTask(Task));

pointcut perCallerWork(Caller c, Worker w):
    cflow(invocations(c)) && workPoints(w);
```
**context-passing aspects**

```java
abstract aspect CapabilityChecking {
    pointcut invocations(Caller c):
        this(c) && call(void Service.doService(String));

    pointcut workPoints(Worker w):
        target(w) && call(void Worker.doTask(Task));

    pointcut perCallerWork(Caller c, Worker w):
        cflow(invocations(c)) && workPoints(w);

    before (Caller c, Worker w): perCallerWork(c, w) {
        w.checkCapabilities(c);
    }
}
```

**a few beginner mistakes**

- **overuse**
- **misunderstanding interactions with reflection**
  - the *call* pointcut captures call join points made from code, not those made reflectively
  - use *execution* to capture reflection
a few beginner mistakes

- not controlling circularity of advice
  - pointcuts sometimes match more than beginners expect
    ```java
    aspect A {
    before(): call(String toString()) {
      System.err.println(tjp);
    }
    }
    ```
  - use within or cflow to control circularity
    ```java
    aspect A {
    before(): call(String toString())
    && !within(A) {
      System.err.println(tjp);
    }
    }
    ```

summary so far

- presented examples of aspects in design
  - intuitions for identifying aspects
- presented implementations in AspectJ
  - how the language support can help
- raised some style issues
  - objects vs. aspects
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")
    - try grep to find these [Griswold]
  - 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 unplugged but not deleted

---

good designs

summary

- capture “the story” well
- may lead to good implementations, measured by
  - code size
  - tangling
  - coupling
  - etc.

learned through experience, influenced by taste and style
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

Part III

examples and demo
Part IV

Conclusion

AOSD

- **language design**
  - more dynamic crosscuts, type system …

- **tools**
  - more IDE support, aspect discovery, refactoring, re-cutting, crosscutting views…

- **software engineering**
  - UML extension, finding aspects, …

- **metrics**
  - measurable benefits, areas for improvement

- **theory**
  - type system for crosscutting, faster compilation, advanced crosscut constructs, modularity principles

- see also aosd.net
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 - 1.4)
- **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

AspectJ on the web

- [eclipse.org/aspectj](http://eclipse.org/aspectj)
  - documentation
  - downloads
  - user mailing list
  - developer mailing list
  - pointers elsewhere...
summary

- **functions → OOP → AOP**
  - handles greater complexity, provides more flexibility...
  - crosscutting modularity
- **AspectJ**
  - incremental adoption package → revolutionary benefits
  - free AspectJ tools
  - community
  - training, consulting, and support for use

credits

AspectJ is now* 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](http://eclipse.org/aspectj)

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