Aspect-Oriented Programming with AspectJ™

AspectJ.org
PARC

Erik Hilsdale, Jim Hugunin, Wes Isberg,
Gregor Kiczales, Mik Kersten

partially funded by DARPA under contract F30602-97-C0246

outline

• I AOP and AspectJ overview
  – problems, basic concepts, context, adoption

• II AspectJ tutorial
  – first example
  – language mechanisms
  – development environment
  – using aspects

• III conclusion
  – futures, references, summary
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

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

session tracking is not modularized

HTTPRequest
- `getCookies()`
- `getRequestURI()`(doc)
- `getSession()`
- `getRequestedSessionId()`
...

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

HTTPResponse
- `getRequest()`
- `setContentType(contentType)`
- `getOutputStream()`
- `setSessionId(id)`
...

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

Servlet

the cost of tangled code

- **redundant code**
  - same fragment of code in many places
- **difficult to reason about**
  - non-explicit structure
  - the big picture of the tangling isn’t clear
- **difficult to change**
  - have to find all the code involved
  - and be sure to change it consistently
  - and be sure not to break it by accident
crosscutting concerns

HttpRequest
- getCookies()
- getRequestURI()
- getSession()
- getRequestedSessionId()
...

HttpResponse
- getRequest()
- setContentType(contentType)
- getOutputStream()
- setSessionId(id)
...

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

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

Servlet

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
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...

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
the context: IT trends

Complex New Technologies
- Mobile Computing
- J2EE
- Components
- Web Services
- XML
- .NET
- P2P

Increasing Pressures
- Variability: Historically, enterprises have had to update these complex relationships by hard-coding single-use applications and relationships - Gartner
- Quality: Requirements for quality have never been higher - Giga
- Integration: This is a crucial need for almost all corporations - Giga
- Agility: Agile application architecture is a critical element of an effective strategy to deal with continuing innovation - Giga

Applications are becoming more complex and more modular – DRW

the unsolved problem

- Each new technology solves specific problems
- But how do you glue it together with flexibility while supporting variability?
  - 10% of code causes 90% of problems
- Structured → Objects → Components → Aspects
  - each offers an additional kind of "modularity technology"
  - each enables significantly more complex software
- Alternatives (many are ad hoc AOP, lacking flexibility, leverage, generality, explicit structure)
  - EJB, servlet deployment descriptors, other XML languages
  - interceptors, proxies, specialized design patterns (command...)
  - code generation/wizards
  - instrumentation (profilers, coverage tools, …)
  - preprocessors (iContract, etc.)
  - meta-object programming: too complex
Aspect-Oriented Programming

- an idea whose time has come
  - enables significantly more complex and flexible software
  - improve quality with consistent policies
  - manage complexity and variability
  - leverage design skills
- research momentum
- luminary endorsements
  - There's something deeper, something that's truly beyond objects... I note subtle signs that point to a marked transformation, a disruptive technology, on the horizon. – Grady Booch, Chief Scientist, Rational
  - Charles Simonyi, Creator of Microsoft Word and Excel
  - Michael Jackson, leader in software engineering for 30 years
  - Linda Northrup, Director SEI Product Line Systems Program

AspectJ smooth adoption curve

- Value
- Development Support for Existing Code
  - 15 minutes
  - 30 lines
  - unpluggable
  - testing
  - tracing
  - performance measurement
- Auxiliary Functionality
  - limited use design
  - error handling
  - standards & contracts
  - monitoring
- Re-factor Core Functionality
  - module design
  - persistence management
  - security
  - feature variations
- Reusable Libraries
- Aspect-Oriented Architecture
  - new services, programming model
  - product lines
  - extend J2EE
  - Web services, mobile, P2P...
### Expected Benefits

<table>
<thead>
<tr>
<th>Add Aspects to Code</th>
<th>Restructure Code for Aspects</th>
<th>Aspect-Oriented Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>• untangle auxiliary concerns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• improve quality</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• reduce code size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• simplify problem solving</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• retire legacy tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• untangle core concerns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• simple, natural code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• improve agility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• leverage experts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• improve analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• support variability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• architectural quality &amp; flexibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• reuse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• extensibility</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### AspectJ Adoption

- **Rapid growth**
- **User mailing list:** 700 active members
- **MIT Tech Review, Java Report, IBM, Java World, CACM...**
- **Production use**
- **Presentations**
  - JavaOne, OOPSLA, Software Development, SIGS
  - Java, ICSE, TOOLS, ECOOP, JAOO, FSE, …

---

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
commercialization

- 1.0 release
- focus on successful customer deployments
- offerings
  - tutorials
  - customized trainings
  - architecture workshops
  - consulting support
  - project reviews
- product development
  - support, joint development, advanced edition
- business planning

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
- user feedback is driving language design
  - users@aspectj.org, support@aspectj.org
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-class declarations (aka "open classes")
  - aspect
    - a modular unit of crosscutting behavior
      - comprised of advice, inter-class, 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)
movesBy(int, int)

Line

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

Figure

makePoint(..) makeLine(..)

FigureElement

moveBy(int, int)

Point

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

Line

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

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)
**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

```java
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

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

or

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

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

user-defined pointcut

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**

```
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**

```
aspect DisplayUpdating {

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

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

- **after advice runs**
  - "on the way back out"

- **a Line**

- **box means complete running code**

---

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
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

```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

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

aspect DisplayUpdating v2

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 value

```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)));

after(Line line): 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)));

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

primitive pointcut designator

tag\(\text{target}(\text{<type name> } | \text{<formal reference>})\)

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)

tag\(\text{target}(\text{Point})\)
tag\(\text{target}(\text{Line})\)
tag\(\text{target}(\text{FigureElement})\)

“any join point” means it matches join points of all kinds
- method & constructor call join points
- method & constructor execution join points
- field get & set join points
- exception handler execution join points
- static & dynamic initialization join points

---

**idiom for...**

getting target object in a polymorphic pointcut

\[ \text{tag}\left(\text{<supertype name>}\right) && \]

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

```java
pointcut move(FigureElement figElt): 
  target(figElt) &&
  (call(\text{void \ Line.setP1(Point)}) ||
   call(\text{void \ Line.setP2(Point)}) ||
   call(\text{void \ Point.setX(int)}) ||
   call(\text{void \ Point.setY(int)}));

after(FigureElement fe): move(fe) { 
  \text{<fe is bound to the figure element>}
}
```
pointcuts
can expose values at join points

```java
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)));
```

context & multiple classes

```java
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): 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;
    }
}
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;
    }
}

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();
    }
}
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;
    }
}
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();
    }
}
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();
    }
}
```

DisplayUpdating v2

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

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;
    }
}

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;
    }
}
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;
    }
}

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

DisplayUpdating v3

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;
    }
}
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;
    }
}

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

<table>
<thead>
<tr>
<th>Point</th>
<th>2</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>getX()</td>
<td></td>
<td>getIP1()</td>
</tr>
<tr>
<td>getY()</td>
<td></td>
<td>getIP2()</td>
</tr>
<tr>
<td>setX(int)</td>
<td></td>
<td>setP1(Point)</td>
</tr>
<tr>
<td>setY(int)</td>
<td></td>
<td>setP2(Point)</td>
</tr>
<tr>
<td>moveBy(int, int)</td>
<td></td>
<td>moveBy(int, int)</td>
</tr>
</tbody>
</table>

Figure

makePoint(..)
makeLine(..)

Point

Line

FigureElement

moveBy(int, int)

DisplayUpdating

Display

advise is

additional action to take at 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 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);
  }

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

what follows the ':' is always a pointcut – primitive or user-defined
post-condition

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

condition enforcement

```java
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

\[ \text{<Tr> around(T1 arg1, T2 arg2, \ldots)} \]

there is a special method with the signature

\[ \text{<Tr> proceed(T1, T2, \ldots)} \]

available only in around advice

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

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 {
    Log log = new Log();

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

    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

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.getX())
call(void Point.getY())
call(void Point.get*)()
call(void get*())
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 getter
any constructor
other primitive pointcuts

\begin{verbatim}
this(<type name>)
within(<type name>)
withincode(<method constructors signature>)

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 method or constructor

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

field reference or assignment join points
\end{verbatim}

fine-grained protection

\begin{verbatim}
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.");
    }
}
\end{verbatim}

want to ensure that any creation of figure elements goes through the factory methods
Aspect-Oriented Programming with AspectJ

**fine-grained protection**

a compile-time error

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

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

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

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

}
```

must be a "static pointcut" (more on this later)

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
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."
    }
}
special value

reflective* access to the join point

thisJoinPoint.

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

available in any advice

* introspective subset of reflection consistent with Java

using thisJoinPoint

in highly polymorphic advice

aspect  PointCoordinateTracing {

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

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

- **execution(\texttt{void Point.setX(int)})**
  - method/constructor execution join points (actual running method)

- **initialization(\texttt{Point})**
  - object initialization join points

- **staticinitialization(\texttt{Point})**
  - class initialization join points (as the class is loaded)

---

other primitive pointcuts

- **cflow(pointcut designator)**
  - all join points within the dynamic control flow of any join point in \texttt{pointcut designator}

- **cflowbelow(pointcut designator)**
  - all join points within the dynamic control flow below any join point in \texttt{pointcut designator}
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);
    }
}
```

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 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

---

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();
        ...
    }
}
```

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
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

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

    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);
}
```
this is the concrete reuse

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);
    }
}

design invariants

aspect FactoryEnforcement {

    pointcut newFigElt():
        call(FigureElement.new(..));

    pointcut inFactory():
        within(Point Figure.make*(..));

    pointcut illegalNewFigElt():
        newFigElt() && !inFactory();

    declare error: illegalNewFigElt():
        "Must call factory method to create figure elements.";
}

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
### summary

**join points**
- method & constructor
- call
- execution
- field
- get
- set
- exception handler
- execution
- initialization

**aspects**
- crosscutting type

**pointcuts**
- **primitive**
  - call
  - execution
  - handler
  - get
  - set
  - initialization
  - this target
  - within
  - within code
  - cflow
  - cflow below
- **user-defined**
  - pointcut declaration
  - abstract
  - overriding

**advice**
- before
- after
- around

**inter-type decls**
- Type.field
- Type.method()

**declare**
- warning
- error
- parents

**reflection**
- thisJoinPoint
- thisJoinPointStaticPart

### 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
tracing without AspectJ

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

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
tracing as an aspect

```java
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...**
plug and debug

```java
public void service(Request rrequest, Response rresponse) {
    // log("New request " + rrequest);
    try {
        rrequest.setContextManager(this);
        rrequest.setResponse(rresponse);
        int status = rresponse.getStatus();
        if (status < 400)
            status = processRequest(rrequest);
        if (status == 0)
            status = authenticate(rrequest, rresponse);
        if (status == 0)
            authorize(rrequest, rresponse);
        if (status == 0) {
            rrequest.getWrapper().handleRequest(rrequest, rresponse);
        } else {
            // something went wrong
            handleError(rrequest, rresponse, null, status);
        }
    } catch (Throwable t) {
        handleError(rrequest, rresponse, t, 0);
    }
    // System.out.print("B");
    try {
        rresponse.finish();
        rrequest.recycle();
        rresponse.recycle();
    } catch (Throwable ex) {
        if (debug > 0) log("Error closing request " + ex);
    }
    // log("Done with request " + rrequest);
    // System.out.print("C");
}
```

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

<table>
<thead>
<tr>
<th>Have these benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objects</strong> are no longer responsible for using the trace facility</td>
</tr>
<tr>
<td>- trace aspect encapsulates that responsibility, for appropriate objects</td>
</tr>
<tr>
<td><strong>If the Trace interface changes, that change is shielded from the objects</strong></td>
</tr>
<tr>
<td>- only the trace aspect is affected</td>
</tr>
<tr>
<td><strong>Removing tracing from the design is trivial</strong></td>
</tr>
<tr>
<td>- just remove the trace aspect</td>
</tr>
</tbody>
</table>

### Aspects in the Code

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

- using an object captures tracing support, but does not capture its consistent usage by other objects
- using an aspect captures the consistent usage of the tracing support by the objects

```java
aspect Tracing {
    abstract pointcut trace();
    before (): trace() {
        TraceSupport.traceEntry(tjp);
    }
    after (): trace() {
        TraceSupport.traceExit(tjp);
    }
}
```

```java
aspect BigBoxCoTracing extends Tracing {
    pointcut trace():
        within(com.bigboxco.*)
        && execution(* *(..));
    before (): trace() {
        TraceSupport.traceEntry(tjp);
    }
    after (): trace() {
        TraceSupport.traceExit(tjp);
    }
}
```
example

layers of functionality

- given a basic telecom operation, with customers, calls, connections
- model/design/implement utilities such as
  - timing
  - consistency checks
  - ...

telecom basic design

These classes define the protocols for setting up calls (includes conference calling) and establishing connections.
Connections and Customers are involved
- well defined protocols among them
- pieces of the timing protocol must be triggered by the execution of certain basic operations. e.g.
  - when connection is completed, set and start a timer
  - when connection drops, stop the timer and add time to customers' connection time
**Aspect-Oriented Programming with AspectJ**

### timing

**an aspect implementation**

```java
aspect Timing {
    private Timer Connection.timer = new Timer();

    private long Customer.totalConnectTime = 0;
    public static long getTotalConnectTime(Customer c) {
        return c.totalConnectTime;
    }

    pointcut startTiming(Connection c): target(c) && call(void c.complete());
    pointcut endTiming(Connection c): target(c) && call(void c.drop());

    after (Connection c) returning : startTiming(c) {
        c.timer.start();
    }

    after (Connection c) returning : endTiming(c) {
        Timer timer = c.timer;
        timer.stop();
        long currTime = timer.getTime();
        c.getCaller().totalConnectTime += currTime;
        c.getReceiver().totalConnectTime += currTime;
    }
}
```

### timing as an object

**Customer**

...  
long getTime()  
void addToTime(long)

**Timer**

void stop()  
void start()  
long getTime()

**Connection**

...  
drop()  
new()

Timing as an object captures timing support, but does not capture the protocols involved in implementing the timing feature
**timing as an aspect**

Timing

```java
long getTime()
void addToTime(long t)

addToTime(timer.getTime())
```

Timer

```java
void stop()
void start()
long getTime()
```

Customer

```java
...
```

Connection

```java
...
drop()
new(...)
```

0..N

0..N

```java
timing as an aspect captures the protocols involved in implementing the timing feature
```

---

**timing**

**interface change**

- Consider a change to the timer interface

```java
Timer

void start()
long stopAndGetTime()
```

- What changes are necessary in the program?

```java
aspect Timing {
    ...
    after(Connection c): endTiming(c) {
        Timer timer = c.timer;
        long currTime = timer.stopAndGetTime();
        c.getCaller().totalConnectTime += currTime;
        c.getReceiver().totalConnectTime += currTime;
    }
}
```
### timing as an aspect

<table>
<thead>
<tr>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>basic objects are not responsible for using the timing facility</strong></td>
</tr>
<tr>
<td>- timing aspect encapsulates that responsibility, for appropriate objects</td>
</tr>
<tr>
<td><strong>if requirements for timing facility change, that change is shielded from the objects</strong></td>
</tr>
<tr>
<td>- only the timing aspect is affected</td>
</tr>
<tr>
<td><strong>removing timing from the design is trivial</strong></td>
</tr>
<tr>
<td>- just remove the timing aspect</td>
</tr>
</tbody>
</table>

### timing with AspectJ

<table>
<thead>
<tr>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>object code contains no calls to timing functions</strong></td>
</tr>
<tr>
<td>- timing aspect code encapsulates those calls, for appropriate objects</td>
</tr>
<tr>
<td><strong>if requirements for timing facility change, there is no need to modify the object classes</strong></td>
</tr>
<tr>
<td>- only the timing aspect class and auxiliary classes needs to be modified</td>
</tr>
<tr>
<td><strong>removing timing from the application is trivial</strong></td>
</tr>
<tr>
<td>- compile without the timing aspect class</td>
</tr>
</tbody>
</table>
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
context-passing aspects

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

context-passing aspects

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

pointcut workPoints(Worker w):
    target(w) && call(void Worker.doTask(Task));
context-passing aspects

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);
    }
}

(c) Copyright 1998-2002 Palo Alto Research Center Incorporated. All Rights Reserved.
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 re-use
  - plug and play
    - 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

conclusion

AOSD future

- **language design**
  - more dynamic crosscuts, type system …
- **tools**
  - more IDE support, aspect discovery, re-factoring, 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 possible features

**continue building language, compiler & tools**

- **user demand driven**
- **specialized support**
  - J2EE (servlet, JSP, EJB, JMS), J2ME
- **flexible weaving**
  - bytecodes
  - intermediate form for aspect libraries
  - load time
- **tools**
  - incremental compiler
  - re-factoring, structure-aware editing, design …
- **language**
  - aspect configuration
  - extensible pointcuts
  - generic types (Java 1.5)
  - structure-shy XML support
  - 2.0?: new dynamic crosscut constructs

## 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 1.2 - 1.4)
  - produces Java platform-compatible .class files (Java 1.1 - 1.4)
- **AspectJ tools support**
  - IDE extensions: Emacs, JBuilder, Forte4J, Eclipse
  - ajdoc to parallel javadoc
  - ant tasks
  - JPDA debugger integration (JSR 45 support)
- **license**
  - compiler, runtime and tools are free for any use
  - compiler and tools are Open Source
## AspectJ on the web

- [aspectj.org](http://aspectj.org)
  - documentation
  - downloads
  - users@aspectj.org
  - support@aspectj.org

## summary

- **OOP $\rightarrow$ AOP**
  - handles greater complexity, provides more flexibility…
  - crosscutting modularity

- **AspectJ**
  - incremental adoption package $\rightarrow$ revolutionary benefits
  - free AspectJ tools
  - community
  - training, consulting, and support for use
credits

AspectJ.org is a PARC project:
Erik Hilsdale, Jim Hugunin, Wes Isberg,
Mik Kersten, Gregor Kiczales

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

partially funded by DARPA under contract F30602-97-C0246