Aspect Oriented Programming

AspectJ.org  
Palo Alto Research Center  
Erik Hilsdale, Jim Hugunin, Wes Isberg, Gregor Kiczales, Mik Kersten

(slightly modified, and some slides added from Martin Giese, Chalmers)

---

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

- 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

---

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
  - get no help from OO tools
concerns

- Sample concerns of a software system:
  - XML parsing
  - UML pattern matching
  - Logging
  - Session management
  -...

Separation of concerns

is a time-honored principle of software design

cross-cutting concerns

- In the motivation,
  - XML parsing and URL pattern matching fit the class hierarchy
  - Logging and Session Management do not.

- A cross-cutting concern is one that needs to be addressed in more than one module

cross-cutting concerns

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

aspects

- Well-modularized crosscutting concerns

A simple figure editor

- Display

- Operation that move elements

- Factory methods

- 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) { ... }
}
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

join point terminology

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

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
- 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

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

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

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

---

**Recall: without AspectJ**

```java
class Line {
    private Point p1, p2;
    Point getPointX() { return p1; }
    Point getPointY() { return p2; }
    void setPointX(int x) {
        p1.x = x;
        Display.update();
    }
    void setPointY(int y) {
        p2.y = y;
        Display.update();
    }
}
```

- **what you would expect**
  - update calls are tangled through the code
  - "what is going on" is less explicit

---

**without AspectJ**

```java
class Line {
    private Point p1, p2;
    Point getPointX() { return p1; }
    Point getPointY() { return p2; }
    void setPointX(int x) {
        p1.x = x;
    }
    void setPointY(int y) {
        p2.y = y;
    }
}
```

---

**AspectJ**

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

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

---

**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

DisplayUpdating

```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(FigureElement fe): move(fe) {
        move(fe);
    }
}
```

without AspectJ

DisplayUpdating v3

```java
class Line {
    private Point p1; // x, y
    private Point p2; // x, y
    private int x; // x
    private int y; // y
    private Point centerP; // x, y
    ...
}

class Point {
    private int x; // x
    private int y; // y
    private Point centerP; // x, y
    ...
}
```

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

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): move(fe) {
    <fe is bound to the figure element>
}
```

context & multiple classes

DisplayUpdating v3

```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

DisplayUpdating v1

```java
class Line {
    private Point p1; // x, y
    private Point p2; // x, y
    private int x; // x
    private int y; // y
    private Point centerP; // x, y
    ...
}

class Point {
    private int x; // x
    private int y; // y
    private Point centerP; // x, y
    ...
}
```
without AspectJ

DisplayUpdating v2

```java
class Line {
    private Point p1, p2;
    private int x1, y1, x2, y2;
    line (p1, p2) { return (x1 = p1.x; y1 = p1.y; x2 = p2.x; y2 = p2.y); }
    line (x1, y1, x2, y2) { return (p1 = new Point(x1, y1); p2 = new Point(x2, y2)); }
    setP1(Point p1) { this.p1 = p1; }
    setP2(Point p2) { this.p2 = p2; }
    getP1() { return this.p1; }
    getp2() { return this.p2; }
    move(int x, int y) { this.p1 = new Point(this.p1.x + x, this.p1.y + y); this.p2 = new Point(this.p2.x + x, this.p2.y + y); }
}
```

class Point {
    private int x, y;
    point (int x, int y) { this.x = x; this.y = y; }
    getx() { return this.x; }
    gety() { return this.y; }
    move(int x, int y) { this.x += x; this.y += y; }
}

without AspectJ

DisplayUpdating v3

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

with AspectJ

DisplayUpdating v1

```java
class Line {
    private Point p1, p2;
    private int x1, y1, x2, y2;
    line (p1, p2) { return (x1 = p1.x; y1 = p1.y; x2 = p2.x; y2 = p2.y); }
    line (x1, y1, x2, y2) { return (p1 = new Point(x1, y1); p2 = new Point(x2, y2)); }
    setP1(Point p1) { this.p1 = p1; }
    setP2(Point p2) { this.p2 = p2; }
    getP1() { return this.p1; }
    getp2() { return this.p2; }
    move():
        Display.Update(p1);
        Display.Update(p2);
        Display.Update(p2);
        Display.Update(p1);
        this.p1 = new Point(this.p1.x + x, this.p1.y + y);
        this.p2 = new Point(this.p2.x + x, this.p2.y + y);
}
```

with AspectJ

DisplayUpdating v2

```java
aspect DisplayUpdating {
    pointcut
        call(void
            Display.Update(figElt))
    returning:
        void
            move():
            Display.Update(p1);
            Display.Update(p2);
            Display.Update(p2);
            Display.Update(p1);
            this.p1 = new Point(this.p1.x + x, this.p1.y + y);
            this.p2 = new Point(this.p2.x + x, this.p2.y + y);
}
```

with AspectJ

DisplayUpdating v3

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

aspect PointBoundsPreCondition {
    before(int newX): call(void Point.setX(int)) && args(newX) { 
        assert(newX >= MIN_X); 
        assert(newX <= MAX_X); 
    }
}

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

aspect PointBoundsEnforcement {
    void around(Point p, int newX): call(void Point.setX(int)) && args(newX) { 
        proceed(p, clip(newX, MIN_X, MAX_X)); 
    }
    void around(Point p, int newY): call(void Point.setY(int)) && args(newY) { 
        proceed(p, clip(newY, MIN_Y, MAX_Y)); 
    }
    private int clip(int val, int min, int max) { 
        return Math.max(min, Math.min(max, val)); 
    }
}

contract checking

- 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); 
    }
}

post-condition

using after advice

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

condition enforcement

using around advice

aspect PointBoundsEnforcement {
    void around(Point p, int newX): call(void Point.setX(int)) && args(newX) { 
        proceed(p, clip(newX, MIN_X, MAX_X)); 
    }
    void around(Point p, int newY): call(void Point.setY(int)) && args(newY) { 
        proceed(p, clip(newY, MIN_Y, MAX_Y)); 
    }
    private int clip(int val, int min, int max) { 
        return Math.max(min, Math.min(max, val)); 
    }
}
wildcarding in pointcuts

```
//                 //
// what goes here?  //
//                 //

aspect DisplayUpdating extends Observing {

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

pointcut changes(Subject s):
  any method on FigureElement
  any method on Point
  any method on any type in graphics.geom
  any method on any type in any sub-package of graphics

| target(Point) | any method on Point |
| target(graphics.geom.Point) | any method on any type in graphics.geom |
| target(graphics.geom.*) | any method on any type in any sub-package of graphics |

| call(void Point.setX(int)) | any setter |
| call(void Point.setY(int)) | any setter |
| call(void Point.get*()) | any getter |
| call(void Point.new(int, int)) | any constructor |
| call(new(..)) | any constructor |
| call(Point.setX(Point)) | any public method on Point |
| call(Point.setY(Point)) | any public method on Point |
| call(Point.get*()) | any public method on any type in graphics.geom |
| call(Point.new*()) | any public method on any type in any sub-package of graphics |
}
```

role types and reusable

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

  public void addObserver(Subject s, Observer o) { ... }
  public void removeObserver(Subject s, Observer o) { ... }

  abstract pointcut changes(Subject s);

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

this is the concrete reuse

```
aspect DisplayUpdating4 extends Observing {

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

pointcut changes(Subject s):
  any method on FigureElement
  any method on Point
  any method on any type in graphics.geom
  any method on any type in any sub-package of graphics

| target(Point) | any method on Point |
| target(graphics.geom.Point) | any method on any type in graphics.geom |
| target(graphics.geom.*) | any method on any type in any sub-package of graphics |

| call(void Point.setX(int)) | any setter |
| call(void Point.setY(int)) | any setter |
| call(void Point.get*()) | any getter |
| call(void Point.new(int, int)) | any constructor |
| call(new(..)) | any constructor |
| call(Point.setX(Point)) | any public method on Point |
| call(Point.setY(Point)) | any public method on Point |
| call(Point.get*()) | any public method on any type in graphics.geom |
| call(Point.new*()) | any public method on any type in any sub-package of graphics |
}
```

element 3 counting bytes

```
interface OutputStream {
  public void write(byte b);
}

/* This SIMPLE aspect keeps a global count of all the bytes ever written to an OutputStream. */
aspect ByteCounting {

  int count = 0;
  int getCount() { return count; }

  // what goes here?
}
```

counting bytes v1 a first attempt

```
aspect ByteCounting {

  int count = 0;
  int getCount() { return count; }

  after() returning: call(void OutputStream.write(byte)) {
      count = count + 1;
  }

  after(byte[] bytes) returning: call(void OutputStream.write(bytes)) {
      count = count + bytes.length;
  }
}
```

counting bytes some stream implementations

```
class SimpleOutputStream implements OutputStream {
  public void write(byte b) { ... }

  public void writeBytes(byte[] b) {
    for (int i = 0; i < b.length; i++) write(b[i]);
  }
}

class OneOutputStream implements OutputStream {
  public void write(byte b) { ... }

  public void writeBytes(byte[] b) { ... }
}
```

Copyright 1998-2002 Palo Alto Research Center. All Rights Reserved.
### Counting Bytes

**Another Implementation**

```java
class OtherOutputStream implements OutputStream {
    public void write(byte b) {
        byte[] bs = new byte[1] { b);
        write(bs);
    }

    public void write(byte[] b) {
    }
}
```

### Other Primitive Pointcuts

- `cflow(pointcut designator)`
  - all join points within the dynamic control flow of any join point in `pointcut designator`
- `cflowbelow(pointcut designator)`
  - all join points within the dynamic control flow below any join point in `pointcut designator`

### Counting Bytes v2

**Using `cflowbelow` for More Robust Counting**

```java
aspect ByteCounting {
    int count = 0;
    int getCount() { return count; }

    pointcut write(): call(void OutputStream.write(byte)) ||
                          call(void OutputStream.write(byte[]));

    pointcut writeCflow(): cflowbelow(write());

    after() returning:
        'writeCflow'(): call(void OutputStream.write(byte)) {
            count++;
        }

    after(byte[] bytes) returning:
        'writeCflow'(): call(void OutputStream.write(byte[])) {
            count = count + bytes.length;
        }
}
```

### AspectJ Technology

- **AspectJ** is a small extension to Java™
  - valid Java programs are also valid AspectJ programs
- **AspectJ** has its own compiler, `ajc`
  - `ajc` runs on Java 2 platform (Java 1.2 – 1.4)
  - `ajc` produces Java platform compatible `.class` files
- **AspectJ tools support**
  - IDE extensions: Emacs, JBuilder 5, Forte4J
  - `ajdoc` to parallel javadoc
  - debugger: command line, GUI, & IDE
- **License**
  - compiler, runtime and tools are free for any use
  - compiler and tools are Open Source

### When Are Aspects Appropriate?

- There is a concern that:
  - crosses 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 [Grisswold]
    - do a coordinated single thing
      - e.g. timing, observer pattern
      - harder to find these
... beneficial to separate out

- **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