Aspect-Oriented Programming with AspectJ™

AspectJ.org
Xerox PARC

Erik Hilsdale
Gregor Kiczales
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:
  - is an aspect-oriented extension to Java™ that supports general-purpose aspect-oriented programming
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

ApplicationSession

StandardSession

ServerSession

SessionInterceptor

StandardManager

ServerSessionManager

import javax.servlet;
import javax.servlet.http;
import java.io.*;
import java.net.*;

private Hashtable values = new Hashtable();

/**
 * @author James Todd [gonzo@eng.sun.com]
 * @author Jason Hunter [jch@eng.sun.com]
 */

* individuals on behalf of the Apache Software Foundation. For more
*
* Redistributions of source code must retain the above copyright
*
* @author James Duncan Davidson [duncan@eng.sun.com]
*
* Core implementation of a server session

synchronized void reap() {
    if (appSession == null && create) {
        return id;
    }

    appSession.validate();

appSession.invalidate();

public void putValue(String name, Object value) {
    throw new IllegalStateException(msg);
}

public void setMaxInactiveInterval(int interval) {
    throw new IllegalArgumentException(msg);
}

String msg = sm.getString("applicationSession.session.ise");

void removeAttribute(String name) {
    stream.writeObject(attributes.get(name));
}

public void putValue(String name, Object value) {
    stream.writeObject(new Boolean(isValid));
}

public void setCheckInterval(int checkInterval) {
    return (this.checkInterval);
}

return (this.checkInterval);

session. A negative time
support.

import java.io.ObjectOutputStream;

Session is associated
View.

Session view.

<code>

*p END IMPLEMENTATION NOTE*

new SessionInterceptor()

// XXX
problems like…

session tracking is not modularized

<table>
<thead>
<tr>
<th>HttpServletRequest</th>
<th>SessionInterceptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCookies()</td>
<td>requestMap(request)</td>
</tr>
<tr>
<td>getRequestURI()</td>
<td>beforeBody(req, resp)</td>
</tr>
<tr>
<td>getSession()</td>
<td>...</td>
</tr>
<tr>
<td>getRequestedSessionId()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HttpServletResponse</th>
<th>Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>getRequest()</td>
<td>getAttribute(name)</td>
</tr>
<tr>
<td>setContentType(contentType)</td>
<td>setAttribute(name, val)</td>
</tr>
<tr>
<td>getOutputStream()</td>
<td>invalidate()</td>
</tr>
<tr>
<td>setSessionId(id)</td>
<td>...</td>
</tr>
</tbody>
</table>

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

```
ServletRequest
getCookies()
getRequestURI(doc)
getSession()
getRequestedSessionId()
...

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

ServletResponse
getRequest()
setContentType(contentType)
getOutputStream()
setSessionId(id)
...

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

Servlet
```
the AOP idea

aspect-oriented programming

crosscutting is inherent in complex systems

crosscutting concerns
  – have a clear purpose
  – have a natural structure
    • defined set of methods, module boundary crossings, points of resource utilization, lines of dataflow…

so, let’s capture the structure of crosscutting concerns explicitly...
  – in a modular way
  – with linguistic and tool support

aspects are
  – well-modularized crosscutting concerns
language support to...
modular aspects

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

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

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

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

Servlet

Aspect-Oriented Programming with AspectJ -- OOPSLA 2001
AspectJ™ is...

- a small and well-integrated extension to Java
- a general-purpose AO language
  - just as Java is a general-purpose OO language
- freely available implementation
  - compiler is Open Source
- includes IDE support
  - emacs, JBuilder, Forte
- user feedback is driving language design
  - users@aspectj.org
  - support@aspectj.org
- currently at 1.0 release
expected benefits of using AOP

• good modularity, even for crosscutting concerns
  – less tangled code
  – more natural code
  – shorter code
  – easier maintenance and evolution
    • easier to reason about, debug, change
  – more reusable
    • library aspects
    • plug and play aspects when appropriate
outline

• I  AOP overview
  – brief motivation, essence of AOP idea
• II  AspectJ language mechanisms
  – basic concepts, language semantics
• III  development environment
  – IDE support, running the compiler, debugging etc.
• IV  using aspects
  – aspect examples, how to use AspectJ to program aspects, exercises to solidify the ideas
• V  related work
  – survey of other activities in the AOP community
looking ahead

problem structure

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

AspectJ mechanisms

Part II:
crosscutting in the code mechanisms AspectJ provides
Part II

Basic Mechanisms of AspectJ
goals of this chapter

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

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

• 1 overlay onto Java
  – 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 pointcuts
      – user-defined pointcuts
  – advice
    • additional action to take at join points in a pointcut
  – introduction
    • additional fields/methods/constructors for classes
  – aspect
    • a crosscutting type
      – comprised of advice, introduction, field, constructor and method declarations
a simple figure editor

class Line implements FigureElement {
    private Point p1, p2;

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

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

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; }
}

display must be updated when objects move
move tracking

- **collection of figure elements**
  - that change periodically
  - must monitor changes to refresh the display as needed
  - collection can be complex
    - hierarchical
    - asynchronous events

- **other examples**
  - session liveness
  - value caching
join points

key points in dynamic call graph

- A method is called and returns or throws
- A method executes and a returns or throws
- Dispatch

a Figure

a Line
join point terminology

key points in dynamic call graph

- several kinds of join points
  - 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
all join points on this slide are within the control flow of this join point

repeated calls to the same method on the same object result in multiple join points
the pointcut construct

names certain join points

each time a Line receives a
“void setP1(Point)” or “void setP2(Point)” method call

ame and parameters

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

or

a “void Line.setP1(Point)” call

or

a “void Line.setP2(Point)” call
pointcut designators

**pointcut** move():

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

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

**user-defined pointcut designator**
after advice

action to take after computation under join points

after advice runs “on the way back out”

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

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

```java
aspect MoveTracking {
  private boolean flag = false;
  public boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;
  }

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

  after() returning: move() {
    flag = true;
  }
}
```

an aspect defines a special class that can crosscut other classes

box means complete running code
without AspectJ

MoveTracking v1

class Line {
    private Point p1, p2;

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

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

class MoveTracking {
    private static boolean flag = false;

    public static void setFlag() {
        flag = true;
    }

    public static boolean testAndClear() {
        boolean result = flag;
        flag = false;
        return result;
    }
}

- what you would expect
  - calls that set the flag are tangled through the code
  - “what is going on” is less explicit
the pointcut construct can cut across multiple classes

pointcut move():
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int));
a multi-class aspect

MoveTracking v2

```java
aspect MoveTracking {
    private boolean flag = false;
    public boolean testAndClear() {
        boolean result = flag;
        flag = false;
        return result;
    }

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

    after() returning: move() {
        flag = true;
    }
}
```
using context in advice
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 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>
}
```

parameter mechanism is being used
context & multiple classes

MoveTracking v3

```java
aspect MoveTracking {
    private Set movees = new HashSet();
    public Set getMovees() {
        Set result = movees;
        movees = new HashSet();
        return result;
    }

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

    after(FigureElement fe) returning: move(fe) {
        movees.add(fe);
    }
}
```
parameters... of user-defined pointcut designator

- variable bound in user-defined pointcut designator
- variable in place of type name in pointcut designator
  - pulls corresponding value out of join points
  - makes value accessible on pointcut

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

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

- variable bound in advice
- variable in place of type name in pointcut designator
  - pulls corresponding value out of join points
  - makes value accessible within advice

```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>
}
```
parameters...

• value is ‘pulled’
  – right to left across ‘:’ left side : right side
  – from pointcut designators to user-defined pointcut designators
  – from pointcut to advice

```java
pointcut moves(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(<type name>)

any join point at which
target object is an instance of type (or class) name

target(Point)
target(Line)
target(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
an idiom for...

getting object in a polymorphic pointcut

target(<supertype name>) &&

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

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

after(FigureElement fe): move(fe) {
  <fe is bound to the figure element>
}
context & multiple classes

```java
aspect MoveTracking {
    private Set movees = new HashSet();
    public Set getMovees() {
        Set result = movees;
        movees = new HashSet();
        return result;
    }

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

    after(FigureElement fe): move(fe) {
        movees.add(fe);
    }
}
```
without AspectJ

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

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

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

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

class Line {
  private Point p1, p2;

  Point getP1() { return _p1; }
  Point getP2() { return _p2; }

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

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;
  }
}

class MoveTracking {
  private static boolean flag = false;

  public static void setFlag() {
    flag = true;
  }

  public static boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;
  }
}
without AspectJ

MoveTracking v2

class Line {
  private Point p1, p2;

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

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

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

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

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

class MoveTracking {
  private static boolean flag = false;

  public static void setFlag() {
    flag = true;
  }

  public static boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;
  }
}
without AspectJ

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

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

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

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

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

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

class MoveTracking {
    private static Set moves = new HashSet();

    public static void collectOne(Object o) {
        moves.add(o);
    }

    public static Set getMoves() {
        Set result = moves;
        moves = new HashSet();
        return result;
    }
}
```

• evolution is cumbersome
  – changes in all three classes
  – have to track all callers
    • change method name
    • add argument

MoveTracking 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;
    }
}
with AspectJ

MoveTracking v1

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 MoveTracking {
    private boolean flag = false;
    public boolean testAndClear() {
        boolean result = flag;
        flag = false;
        return result;
    }

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

    after(): move() {
        flag = true;
    }
}
with AspectJ

MoveTracking 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;
    }
}

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 MoveTracking {
    private boolean flag = false;
    public boolean testAndClear() {
        boolean result = flag;
        flag = false;
        return result;
    }

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

    after(): move() {
        flag = true;
    }
}
```
with AspectJ

MoveTracking v3

```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;
    }
}

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 MoveTracking {
    private Set moves = new HashSet();
    public Set getMovees() {
        Set result = moves;
        moves = new HashSet();
        return result;
    }

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

    after(FigureElement fe): move(fe) {
        moves.add(fe);
    }
}
```

- evolution is more modular
  - all changes in single aspect
advice 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

using after advice

```java
aspect PointBoundsPostCondition {

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

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

    private void assert(boolean v) {
        if ( !v )
            throw new RuntimeException();
    }
}
```
As an assistant, I cannot directly interact with images or documents. However, I can help you understand the code snippet and explain how it works. This snippet is from a document discussing aspect-oriented programming using AspectJ.

### Code Snippet

```java
aspect PointBoundsEnforcement {
    void around(Point p, int newX):
        call(void Point.setX(int)) && target(p) && args(newX) {
            proceed(p, clip(newX, MIN_X, MAX_X));
        }

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

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

### Explanation

The code defines an aspect named `PointBoundsEnforcement` that enforces bounds on the x and y coordinates of a `Point` object. It uses `around` advice to ensure that the `setX` and `setY` methods are called within the specified bounds.

- **`call(...)`** is used to ensure that the `setX` and `setY` methods are called.
- **`target(...)`** checks if the aspect pointcut matches the `Point` object.
- **`args(...)`** verifies if the arguments passed to the advised method are within the specified bounds.
- **`proceed(...)`** allows the method call to proceed with the modified arguments.
- **`clip(...)`** is a private method that clips an integer value to a specified range.

This aspect-oriented approach allows for modular and reusable code to enforce specific behavior across the system without modifying the underlying implementation directly.
special static method

\[ \text{<result type>} \ proceed(\text{arg1}, \text{arg2}, \ldots) \]

available only in around advice

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

this(<type name>)
within(<type name>)
withincode(<method/constructor signature>)

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

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

field reference or assignment join points
using field set pointcuts

```java
aspect PointCoordinateTracing {

    pointcut coordChanges(Point p, int newVal):
        (set(int Point.x) || set(int Point.y)) &&
        target(p) && args(newVal);

    before(Point p, int newVal):
        coordChanges(p, newVal) {
            System.out.println("At " +
                tjp.getSignature() +
                " field is changed to " +
                newVal +
                ".");
        }
}
```
special value

reflective* access to the join point

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

available in any advice

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

* introspective subset of reflection consistent with Java
other primitive pointcuts

execution(void Point.setX(int))
method/constructor execution join points (at actual called method)

initialization(Point)
object initialization join points

staticinitialization(Point)
class initialization join points (as the class is loaded)
context sensitive aspects

MoveTracking v4

```java
aspect MoveTracking {
    List movers = new LinkedList();
    List movees = new LinkedList();
    // ...

    pointcut moveCalls(Object mover, FigureElement movee): this(mover) && target(movee) &&
        (call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int)));

    after(Object mover, FigureElement movee) returning: moveCalls(mover, movee) {
        movers.add(mover);
        movees.add(movee);
    }
}
```
fine-grained protection

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

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

    void setX(int nv) { primitiveSetX(nv); }
    void setY(int nv) { primitiveSetY(nv); }

    void primitiveSetX(int x) { this.x = x; }
    void primitiveSetY(int y) { this.y = y; }
}

aspect PrimitiveSetterEnforcement {
    pointcut illegalSets():
        !(withincode(void Point.primitiveSetX(int)) ||
            withincode(void Point.primitiveSetY(int))) &&
        (sets(int Point.x) || sets(int Point.y));

    before(): illegalSets() {
        throw new Error("Illegal primitive setter call.");
    }
}
fine-grained protection

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

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

    void setX(int nv) { primitiveSetX(nv); }
    void setY(int nv) { primitiveSetY(nv); }

    void primitiveSetX(int x) { this.x = x; }
    void primitiveSetY(int y) { this.y = y; }
}

aspect PrimitiveSetterEnforcement {
    pointcut illegalSets():
        !(withincode(void Point.primitiveSetX(int)) ||
        withincode(void Point.primitiveSetY(int))) &&
        (sets(int Point.x) || sets(int Point.y));

    declare error: illegalSets(): "Illegal setter call";
}
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.
aspect MoveTracking {

    private Set movees = new HashSet();
    public Set getMovees() {
        Set result = movees;
        movees = new HashSet();
        return result;
    }

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

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

    after(FigureElement fe) returning: topLevelMove(fe) {
        movees.add(fe);
    }
}
wildcarding in pointcuts

```
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
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);
    }
}
```

neatly captures public interface of mypackage

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
aspect state

what if you want a per-object log?

```
aspect PublicErrorLogging
    pertarget(PublicErrorLogging.publicInterface()) {
        Log log = new Log();

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

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

one instance of the aspect for each object that ever executes at these points
looking up aspect instances

```java
static Log getLog(Object obj) {
    return (PublicErrorLogging.aspectOf(obj)).log;
}
```

- **static method of aspects**
  - for default aspects takes no argument
  - for aspects of each cflow takes no arguments
  - for aspects of each object takes an Object
- **returns aspect instance**
aspect relations

pertarget(<pointcut>)
perthis(<pointcut>)
  one aspect instance for each object that is ever “this” at the join points

percflow(<pointcut>)
percflowbelow(<pointcut>)
  one aspect instance for each join point in pointcut, is available at all joinpoints in cflow or cflowbelow
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
public class FigureEditor {
    static pointcut move(FigureElement figElt):
        target(figElt) &&
        (call(void Line.setP1(Point)) ||
        call(void Line.setP2(Point)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int)));

    ...
}

aspect MoveTracking {
    after(FigureElement fe) returning:
        FigureEditor.move(fe) {
            ...
        }...
}

a shared pointcut
abstract public aspect RemoteExceptionLogging {

    abstract pointcut logPoint();

    after() throwing (RemoteException e): logPoint() {
        log.println("Remote call failed in: " +
                thisJoinPoint.toString() +
                "(" + e + ").");
    }
}

public aspect MyRMILogging extends RemoteExceptionLogging {
    pointcut logPoint():
        call(* RegistryServer::*.*(..)) ||
        call(private * RMIMessageBrokerImpl::*.*(..));
}
(introduction)

(like “open classes”)

```java
aspect MoveTracking {
    private Set movees = new HashSet();
    public Set getMovees() {
        Set result = movees;
        movees = new HashSet();
        return result;
    }

    private Object FigureElement.lastMovedBy;
    public Object FigureElement.getLastMovedBy() {
        return lastMovedBy;
    }

    pointcut MoveCalls(Object mover, FigureElement movee):
        instanceof(mover) &&
        (lineMoveCalls(movee) || pointMoveCalls(movee));

    pointcut lineMoveCalls(Line ln):
        calls(void ln.setP1(Point)) || calls(void ln.setP2(Point));
    pointcut pointMoveCalls(Point pt):
        calls(void pt.setX(int)) || calls(void pt.setY(int));

    after(Object mover, FigureElement movee):
        MoveCalls(mover, movee) {
            movees.add(movee);
            movee.lastMovedBy = mover;
        }
}
```

(introduction adds members to target type)

(public and private are with respect to enclosing aspect declaration)
summary

join points
- method & constructor calls
- executions
- field
- gets
- sets
- exception handler executions
- initializations

aspects
- crosscutting type
- pertarget
- perthis
- percflow
- percflowbelow

pointcuts
- primitive
  - call
  - execution
  - handler
  - get
  - set
  - initialization
  - this
  - target
  - within
  - within-code
  - cflow
  - cflowbelow

-user-defined-
- pointcut
- declaration
- abstract
- overriding
- static

advice
- before
- after
- around

introduction
declare
where we have been...

... and where we are going

problem structure

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

AspectJ mechanisms

Part II:
crosscutting in the code
mechanisms AspectJ provides
Part III

AspectJ IDE support
programming environment

• AJDE support for
  – emacs, JBuilder, Forte
• also jdb style debugger (ajdb)
• and window-based debugger

• navigating AspectJ code
• compiling
• tracking errors
• debugging

• ajdoc
Part IV

Using Aspects
where we have been...

... and where we are going

problem structure

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

AspectJ mechanisms

Part II:
crosscutting in the code
mechanisms AspectJ provides
goals of this chapter

• present examples of aspects in design
  – intuitions for identifying aspects
• present implementations in AspectJ
  – how the language support can help
• work on implementations in AspectJ
  – putting AspectJ into practice
• raise some style issues
  – objects vs. aspects
• when are aspects appropriate?
example 1

plug & play tracing

• **plug tracing into the system**
  – exposes join points and uses very simple advice

• **an unpluggable aspect**
  – the program’s functionality is unaffected by the aspect

• **uses both aspect and object**
tracing without AspectJ

```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);
    }
}

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

```
```
a clear crosscutting structure

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

TraceSupport

this line is about interacting with the trace facility
tracing as an aspect

aspect TraceMyClasses {

    pointcut tracedMethod():
        within(com.bigboxco.boxes.*) &&
        execution(* *(..));

    before(): tracedMethod() {
        TraceSupport.traceEntry(
            thisJoinPoint.getSignature());
    }

    after(): tracedMethod() {
        TraceSupport.traceExit(
            thisJoinPoint.getSignature());
    }
}

TraceSupport
plug and debug

• plug in:  ajc Point.java Line.java TraceSupport.java MyClassTracing.java

• unplug:   ajc Point.java Line.java

• or...

![AJBuilder Options screenshot]
plug and debug

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

// From ContextManager

```java
// log( "New request " + rrequest );
// System.out.print("A");

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 have these benefits

• 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

have these benefits

• **object code contains no calls to trace functions**
  – trace aspect code encapsulates those calls, for appropriate objects

• **if the trace interface changes, there is no need to modify the object classes**
  – only the trace aspect class needs to be modified

• **removing tracing from the application is trivial**
  – compile without the trace aspect class
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
tracing

- Make the tracing aspect a library aspect by using an abstract pointcut.
- The after advice used runs whether the points returned normally or threw exceptions, but the exception thrown is not traced. Add advice to do so.
refactor TraceMyClasses into a reusable (library) aspect and an extension equivalent to TraceMyClasses

```java
aspect TracingXXX {
    // what goes here?
}

aspect TraceMyClasses extends TracingXXX {
    // what goes here?
}
```
exercise

we now have the Trace class, and two aspects, from a design perspective, what does each implement?

```
abstract aspect TracingProtocol {

    abstract pointcut tracedMethod();

    before(): tracedMethod() {
        TraceSupport.traceEntry(thisJoinPoint.getSignature());
    }

    after(): tracedMethod() {
        TraceSupport.traceExit(thisJoinPoint.getSignature());
    }
}

aspect TraceMyClasses extends TracingProtocol {

    pointcut tracedMethod():
        within(com.bigboxco.boxes.*) &&
        execution(* *(..));
}
```
example 2

CloneablePoint
<<(aspect>>

clone()

ComparablePoint
<<(aspect>>

cmpareTo(o:Object)

EqualablePoint
<<(aspect>>

equals(o:Object)

HashCode
<<(aspect>>

hashCode()
aspect CloneablePoint {

    declare parents: Point implements Cloneable;

    public Object Point.clone() throws CloneNotSupportedException {
        // we choose to bring all fields up to date before cloning
        makeRectangular();       // defined in class Point
        makePolar();            // defined in class Point

        return super.clone();
    }
}

CloneablePoint
roles/views

exercise/discussion

- Write the HashablePoint and ComparablePoint aspects.
- Consider a more complex system. Would you want the HashablePoint aspect associated with the Point class, or with other HashableX objects, or both?
interface OutputStream {
    public void write(byte b);
    public void write(byte[] b);
}

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

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

    //
    // what goes here? //
    //
}

example 3
counting bytes
exercise

complete the code for ByteCounting

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

    int count = 0;
    int getCount() { return count; }
}
```
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 write(byte[] b) {
        for (int i = 0; i < b.length; i++) write(b[i]);
    }
}

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

    public void write(byte[] b) { … }
}
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) { ... }
}
```
counting bytes v2
using cflow 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 withinWrite(): cflowbelow(write());

    after() returning:
        !withinWrite() && call(void OutputStream.write(byte)) {
            count++;
        }

    after(byte[] bytes) returning:
        !withinWrite() && call(void OutputStream.write(bytes)) {
            count = count + bytes.length;
        }
}
```
counting bytes v3

per-stream counting

```java
aspect ByteCounting of each object(write()) {
    int count;

    int getCountOf(OutputStream str) {
        return ByteCounting.aspectOf(str).count;
    }

    ... count++;

    ... count += bytes.length;
}
```
counting bytes

exercises

• How do the aspects change if the method `void write(Collection c)` is added to the OutputStream interface?
• How would you change v2 to handle byte generators:

```java
interface ByteGenerator {
    int getLength();
    void generateTo(OutputStream s);
}
```
example 4

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

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

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

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

pointcut invocations(Caller c):
  target(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);
abstract aspect CapabilityChecking {

    pointcut invocations(Caller c):
        target(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);
    }
}
example 5

properties of interfaces

```java
interface Forest {
    int howManyTrees();
    int howManyBirds();
    ...
}

pointcut forestCall():
    call(* Forest::*(..));

before(): forestCall(): { }
```
aspects on interfaces

a first attempt

```java
aspect Forestry {
  pointcut forestCall():
      call(* Forest.*(..));

  before(): forestCall() {
      System.out.println(tjp.getSignature() + 
          " is a Forest-Method.");
  }
}
```
aspects on interfaces

interface Forest includes methods from Object, such as toString()
aspects on interfaces

```
aspect Forestry {
    pointcut forestCall():
        call(* Forest.*(..)) 
        && !call(* Object.*(..));

    before(): forestCall() {
        System.out.println(thisJoinPoint.methodName + 
            " is a Forest-method. ");
    }
}
```
Aspects on Interfaces

- In this example you needed to constrain a pointcut because of undesired inheritance. Think of an example where you would want to capture methods in a super-interface.

- Constraining a pointcut in this way can be seen as an aspect *idiom*. What other idioms have you seen in this tutorial?
example 6

client reactions to failures:
- abort
- try another server
a TimeServer design
the TimeService

```java
public interface TimeService extends Remote {

    /**
     * What's the time?
     */
    public Date getTime() throws RemoteException;

    /**
     * Get the name of the server
     */
    public String getName() throws RemoteException;

    /**
     * Exported base name for the service
     */
    public static final String nameBase = "TimeService";
}
```
the TimeServer

```java
public class TimeServer extends UnicastRemoteObject
    implements TimeService {

    /**
     * The remotely accessible methods
     */
    public Date getTime() throws RemoteException {
        return new Date();
    }

    public String getName() throws RemoteException {
        return toString();
    }

    /**
     * Make a new server object and register it
     */
    public static void main(String[] args) {
        TimeServer ts = new TimeServer();
        Naming.bind(TimeService.nameBase, ts);
    }

    /**
     * Exception pointcuts. Code is not complete without advice on them.
     */
    pointcut create():
        within(TimeServer) && call(TimeServer.new());

    pointcut bind(): within(TimeServer) && call(void Naming.bind(String,..));
    pointcut bindName(String name): args(name, ..) && bind();
}
```
aspect AbortMyServer {
    TimeServer around(): TimeServer.create() {
        TimeServer result = null;
        try {
            result = proceed();
        } catch (RemoteException e) {
            System.out.println("TimeServer err: " + e.getMessage());
            System.exit(2);
        }
        return result;
    }
    declare soft: RemoteException: TimeServer.create();

    void around(String name): TimeServer.bindName(name) {
        try {
            proceed(name);
            System.out.println("TimeServer: bound name.");
        } catch (Exception e) {
            System.err.println("TimeServer: error " + e);
            System.exit(1);
        }
    }
    declare soft: Exception: TimeServer.bind();
}
aspect RetryMyServer {  
  TimeServer around(): TimeServer.create() {  
    TimeServer result = null;  
    try { result = proceed(); }  
    catch (RemoteException e){  
      System.out.println("TimeServer error."); e.printStackTrace();  
    }  
    return result;  
  }  
  declare soft: RemoteException: TimeServer.create();  
  
  void around(String name): TimeServer.bindName(name) {  
    for (int tries = 0; tries < 3; tries++) {  
      try {  
        proceed(name + tries);  
        System.out.println("TimeServer: Name bound in registry.");  
        return;  
      }  
      catch (AlreadyBoundException e) {  
        System.err.println("TimeServer: name already bound");  
      }  
      System.err.println("TimeServer: Giving up."); System.exit(1);  
    }  
    declare soft: Exception: TimeServer.bind();  
  }
}
the Client

```java
public class Client {
    TimeService server = null;
    /**
     * Get a server and ask it the time occasionally
     */
    void run() {
        server = (TimeService) Naming.lookup(TimeService.nameBase);
        System.out.println("nRemote Server=", server.getName() + "n\n");
        while (true) {
            System.out.println("Time: " + server.getTime());
            pause();
        }
    }
    /**
     * Exception pointcuts. Code is not complete without advice on them.
     */
    pointcut setup(): call(Remote Naming.lookup(..));
    pointcut setupClient(Client c): this(c) && setup();

    pointcut serve(): call(* TimeService.*(..));
    pointcut serveClient(Client c, TimeService ts):
        this(c) && target(ts) && serve();

    ... other methods ... 
}
```
aspect AbortMyClient {
  Remote around(Client c): Client.setupClient(c) {
    Remote result = null;
    try {
      result = proceed(c);
    } catch (Exception e) {
      System.out.println("Client: No server. Aborting.");
      System.exit(0);
    }
    return result;
  }
  declare soft: Exception: Client.setup();

  Object around(Client c, TimeService ts): Client.serveClient(c, ts) {
    Object result = null;
    try {
      result = proceed(c, ts);
    } catch (RemoteException e) {
      System.out.println("Client: Remote Exception. Aborting.");
      System.exit(0);
    }
    return result;
  }
  declare soft: RemoteException: Client.serve();
}
aspect RetryMyClient {

   Remote around(Client c): Client.setupClient(c) {
      Remote result = null;
      try { result = proceed(c); }
      catch (NotBoundException e) {
         System.out.println("Client: Trying alternative name...");
         result = findNewServer(TimeService.nameBase, c.server, 3);
         if (result == null) System.exit(1); /* No server found */
      } catch (Exception e2) { System.exit(2); }
      return result;
   }
   declare soft: Exception: Client.setup();

   Object around(Client c, TimeService ts): Client.serveClient(c,ts) {
      try { return proceed(c,ts); }
      catch (RemoteException e) { /* Ignore and try other servers */ }
      c.server = findNewServer(TimeService.nameBase, c.server, 3);
      if (c.server == null) System.exit(1); /* No server found */
      try { return thisJoinPoint.runNext(c, c.server); }
      catch (RemoteException e2) { System.exit(2); }
      return null;
   }
   declare soft: RemoteException: Client.serve();

   static TimeService findNewServer(String baseName, Object currentServer, int nservers) { ... }
}
building the client

• abort mode:

  \texttt{ajc Client.java TimeServer_Stub.java AbortMyClient.java}

• retry mode:

  \texttt{ajc Client.java TimeServer_Stub.java RetryMyClient.java}

• switch to different failure handling modes without editing
• no need for subclassing or delegation
• reusable failure handlers
Write another exception handler that, on exceptions, gives up the remote mode and instantiates a local TimeServer.

How would this client look like if the exception handling were not designed with aspects? Can you come up with a flexible OO design for easily switching between exception handlers?

Compare the design of exception handlers with aspects vs. with your OO design.
example 7

layers of functionality

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

Aspect-Oriented Programming with AspectJ -- OOPSLA 2001
telecom basic design

These classes define the protocols for setting up calls (includes conference calling) and establishing connections.
timing

store total connection time

Customer

Call

Connection

caller
receiver

Local

LongDistance

time each connection

entities

126

Aspect-Oriented Programming with AspectJ -- OOPSLA 2001
connection dropped: add time

connection made: start timing
connection dropped: stop timing
long getTime()  

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

invoke when connection drops  
set and invoke upon new connection

Customer  
1  
1  

Call  
0..N

0

caller
receiver

Connection

0..N

Local  
LongDistance  

additional design elements  
timing
Write an aspect representing the timing protocol.
timing

what is the nature of the crosscutting?

• connections and calls 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 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): startTiming(c) {
        c.timer.start();
    }

    after(Connection c): endTiming(c) {
        Timer timer = c.timer;
        timer.stop();
        long currTime = timer.getTime();
        c.getCaller().totalConnectTime += currTime;
        c.getReceiver().totalConnectTime += currTime;
    }
}
timing as an object captures timing support, but does not capture the protocols involved in implementing the timing feature
timing as an aspect

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

has these benefits

• basic objects are not responsible for using the timing facility
  – timing aspect encapsulates that responsibility, for appropriate objects

• if requirements for timing facility change, that change is shielded from the objects
  – only the timing aspect is affected

• removing timing from the design is trivial
  – just remove the timing aspect
timing with AspectJ

has these benefits

- **object code contains no calls to timing functions**
  - timing aspect code encapsulates those calls, for appropriate objects

- **if requirements for timing facility change, there is no need to modify the object classes**
  - only the timing aspect class and auxiliary classes needs to be modified

- **removing timing from the application is trivial**
  - compile without the timing aspect class
How would you change your program if the interface to Timer objects changed to

```java
Timer

void start()
long stopAndGetTime()
```

What changes would be necessary without the aspect abstraction?
telecom, continued

layers of functionality: consistency

• ensure that all calls and connections are being shut down in the simulation
consistency checking

Customer
Call startCall(Customer)
void pickupCall(Call)
void mergeCalls(Call, Call)
void hangupCall(Call)

Call
Customer getCaller()
void pickup()
void merge(Call)
void hangup()

1

0..N

0..N

0..N

store each new call
remove calls that
• hangup
• merge

Customer getCaller()
Customer getReceiver()
void complete()
void drop()

check at the end
of simulation

List of Connections
List of Calls

store each new connection
remove connections that drop

store each new call
remove calls that

Connection
caller
receiver

Local
LongDistance
consistency checking

```java
aspect ConsistencyChecker {
    Vector calls = new Vector(), connections = new Vector();
    
    /* The lifecycle of calls */
    after(Call c): target(c) && call(Call.new(..)) {
        calls.addElement(c);
    }
    after(Call c): target(c) && call(* Call.hangup(..)) {
        calls.removeElement(c);
    }
    after(Call other): args(other) && (void Call.merge(Call)) {
        calls.removeElement(other);
    }
    
    /* The lifecycle of connections */
    after(Connection c): target(c) && call(Connection.new(..)) {
        connections.addElement(c);
    }
    after(Connection c): target(c) && call(* Connection.drop(..)) {
        connections.removeElement(c);
    }
    after(): within(TelecomDemo) && executions(void main(..)) {
        if (calls.size() != 0) println("ERROR on calls clean up.");
        if (connections.size() != 0) println("ERROR on connections clean up.");
    }
}
```
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

- **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

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

summary

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 V

References, Related Work
AOP and AspectJ on the web

- aspectj.org
- www.parc.xerox.com/aop
Workshops

- ECOOP’97
  - http://wwwtrese.cs.utwente.nl/aop-ecoop97
- ICSE’98
- ECOOP’98
- ECOOP’99
- OOPSLA’99
- ECOOP’00
  - http://trese.cs.utwente.nl/Workshops/adc2000/
- OOPSLA’00
  - http://trese.cs.utwente.nl/Workshops/OOPSLA2000/
- ECOOP’01
growing interest in separation of crosscutting concerns

- **aspect-oriented programming**
  - composition filters @ U Twente
    - [Aksit]
  - adaptive programming @ Northeastern U
    - [Lieberherr]
- **multi-dimensional separation of concerns @ IBM**
  - [Ossher, Tarr]
- **assessment of SE techniques @ UBC**
  - [Murphy]
- **information transparency @ UCSD**
  - [Griswold]
- ...

Aspect-Oriented Programming with AspectJ -- OOPSLA 2001
AOP future – idea, language, tools

- **objects are**
  - code and state
  - “little computers”
  - message as goal
  - hierarchical structure

- **languages support**
  - encapsulation
  - polymorphism
  - inheritance

- **tools**
  - browser, editor, debuggers
    - preserve object abstraction

- **aspects are**
  - crosscutting structure

- **languages support**
  - crosscutting

- **tools**
  - preserve aspect abstraction
AOP future

- **language design**
  - more dynamic crosscuts, type system …
- **tools**
  - more IDE support, aspect discovery, re-factoring, re-cutting…
- **software engineering**
  - finding aspects, modularity principles, …
- **metrics**
  - measurable benefits, areas for improvement
- **theory**
  - type system for crosscutting, fast compilation, advanced crosscut constructs
AspectJ & the Java platform

- **AspectJ is a small extension to the Java programming language**
  - all valid programs written in the Java programming language are also valid programs in the AspectJ programming language

- **AspectJ has its own compiler, ajc**
  - ajc runs on Java 2 platform
  - ajc is available under Open Source license
  - ajc produces Java platform compatible .class files
AspectJ status

• **release status**
  – 3 major, ~18 minor releases over last year (1.0alpha is current)
  – tools
    • IDE extensions: Emacs, JBuilder 3.5, JBuilder 4, 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

• **aspectj.org**
  – May 1999: 90 downloads/mo, 20 members on users list
  – Feb 2001: 600 downloads/mo, 600 members on users list

• **tutorials & training**
  – 3 tutorials in 1999, 8 in 1999, 12 in 2000
AspectJ future

continue building language, compiler & tools

• 1.0
  – minor language tuning
  – incremental compilation, compilation to bytecodes

• 1.1
  – faster incremental compiler (up to 5k classes)
  – source of target classes not required
  – at least one more IDE

• 2.0
  – new dynamic crosscut constructs

commercialization decision after 1.0
AspectJ.org is a Xerox PARC project:
Bill Griswold, Erik Hilsdale, Jim Hugunin, Vladimir Ivanovic, Mik Kersten, Gregor Kiczales, Jeffrey Palm

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

partially funded by DARPA under contract F30602-97-C0246