Aspect-Oriented Programming
- *A New Form of Separation*

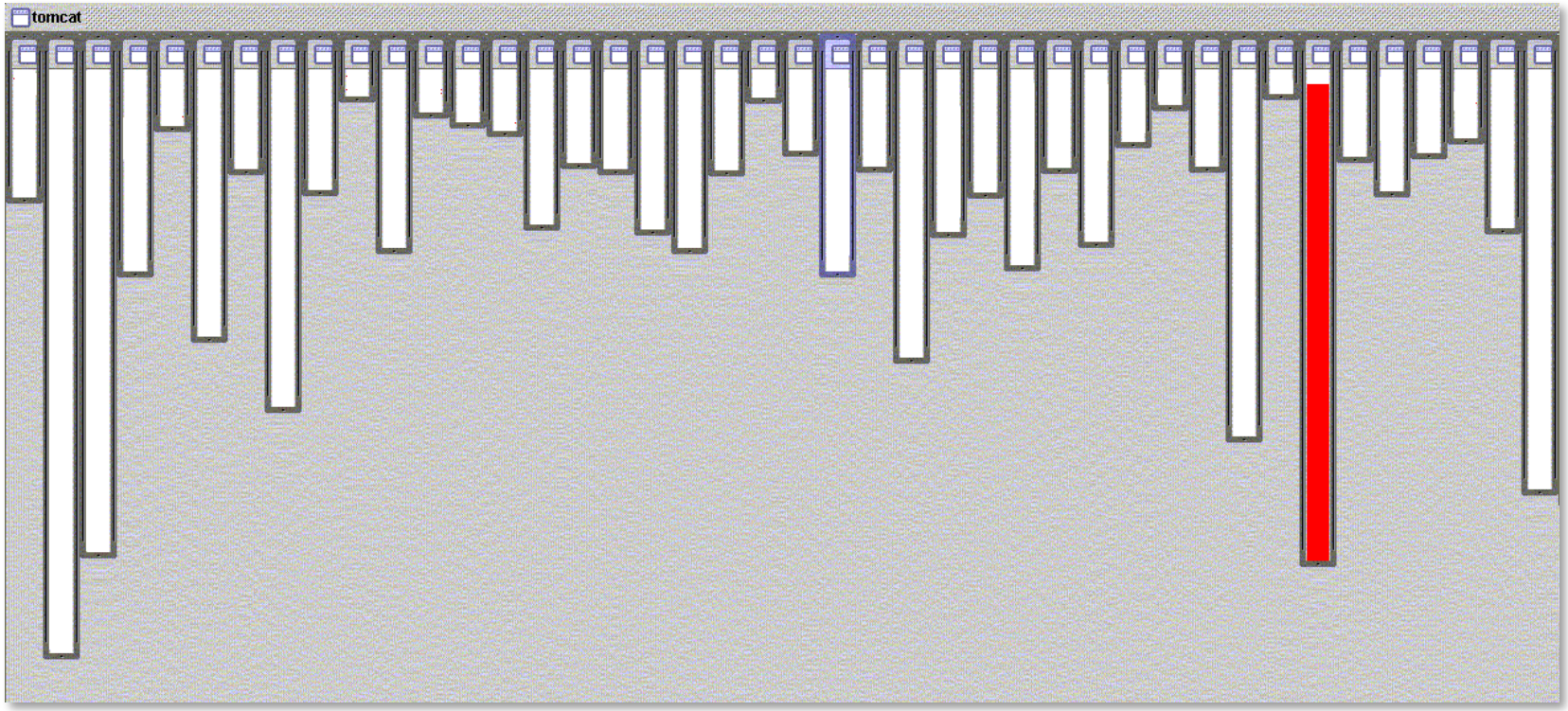
CS580 Advanced Software Engineering
http://cs580.yusun.io
October 13, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

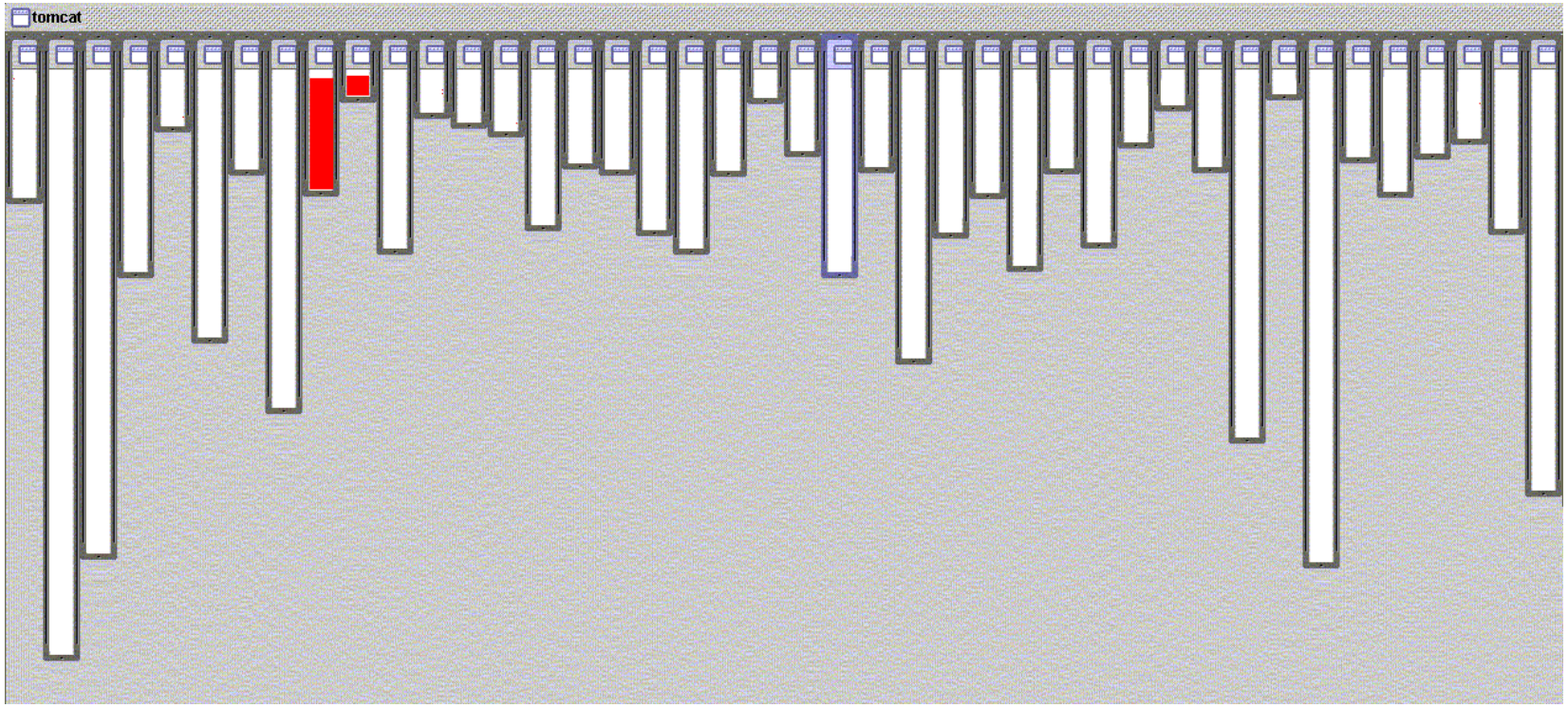
Good Modularity

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



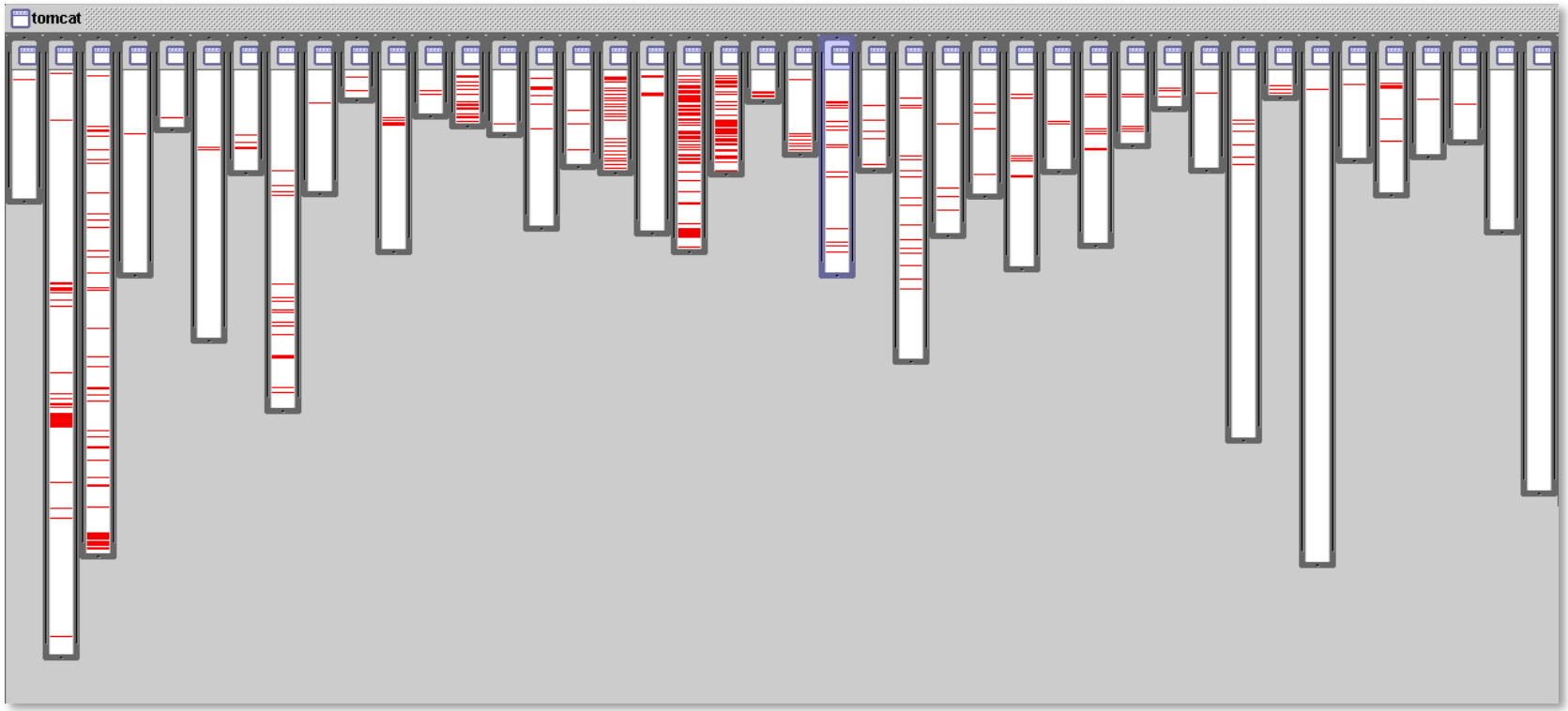
Good Modularity

- URL pattern matching in org.apache.tomcat
  - Red shows relevant lines of code
  - Nicely fits in two boxes (using inheritance)



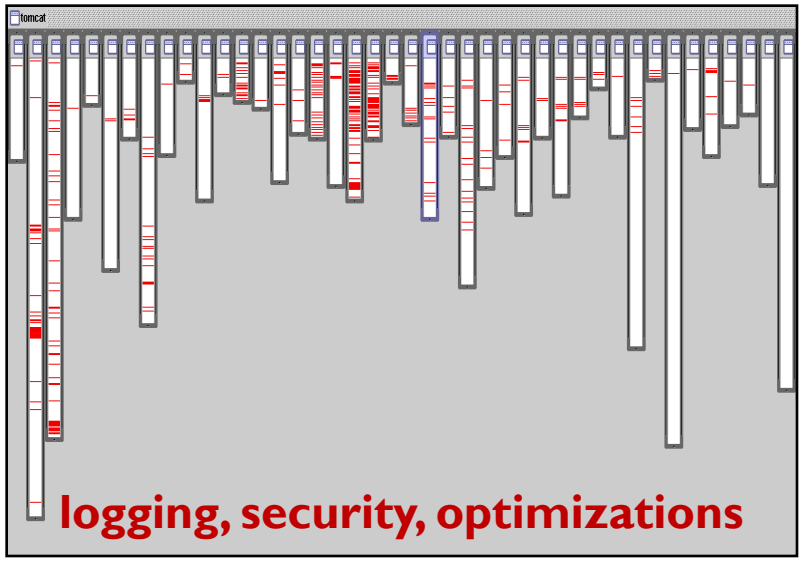
Logging is not Modularized

- Where is logging in org.apache.tomcat
  - Red shows lines of code that handle logging
  - Not in just one place
  - Not even in a small number of places



The Problem of Crosscutting Concerns

- Critical aspects of large systems don't fit in traditional modules
  - Logging, error handling
  - Synchronization
  - Security
  - Memory management
  - Performance optimizations



Crosscutting Concerns

- Two concerns "crosscut" when the modularization of one concern forces the second concern to be captured in an "unclean" manner
  - "Tyranny of the dominant decomposition" – [Tarr et al., 99]
- The result of crosscutting concerns is "tangled code"

The Cost of Tangled Code

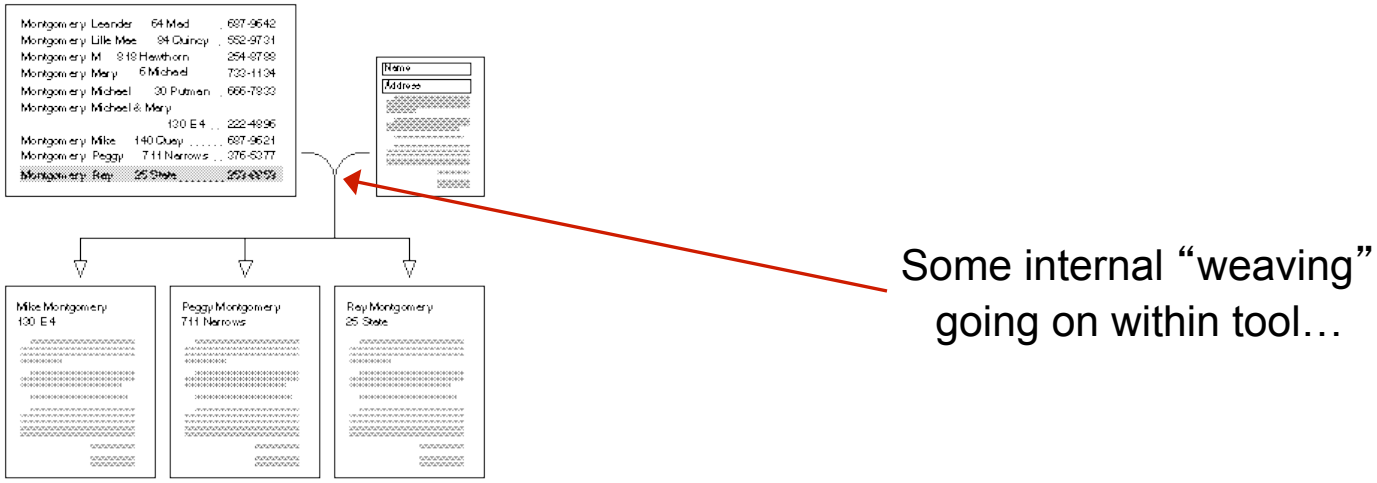
- Redundant code
  - Same fragment of code in many places
- Difficult to reason about
  - Non-explicit structure
  - The big picture of the tangling isn't clear
- Difficult to change
  - Have to find all the code involved
  - And be sure to change it consistently
  - And be sure not to break it by accident

The Aspect-Oriented Programming 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

Mail Merge

- Separation of form from instance
- For example
  - 15 different documents
  - 6 different participants with specific contact information
  - Numerous places within each document where parts of contact information is to be inserted



Style Sheets

- Separation of content from presentation style

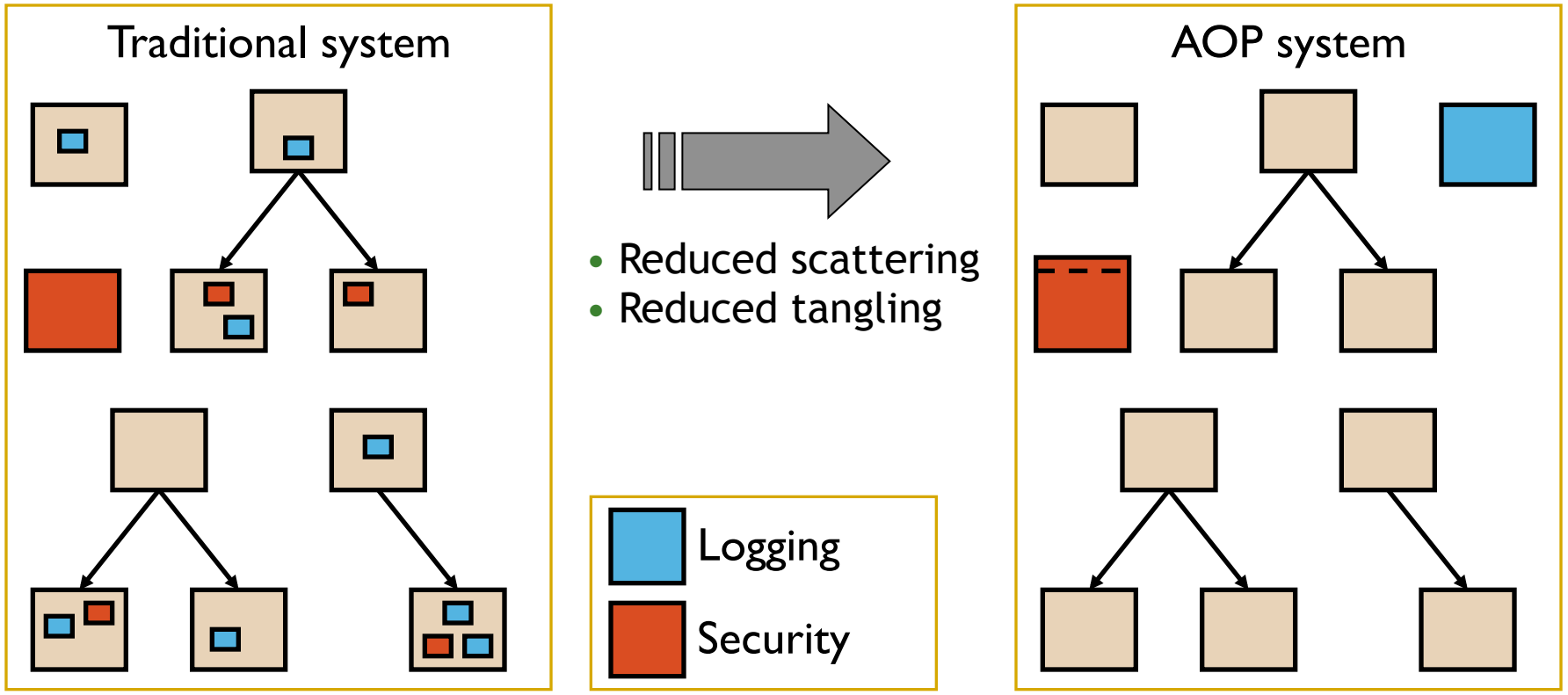
HTML code (content)

```
<html>
<head><title>Chaos Magnet</title></head>
<body>
<div>
   <h1>Chaos Magnet</h1>
   <p>Murphy never had it like this.</p>
</div>
</body>
</html>
```

CSS code (style)

```
<style type="text/css">
   *     { font-family: arial }
   h1    { color: red }
   div   { border: 1px solid black;
           padding: 10px; }
</style>
```

What's the Difference?



Eliminating Tangling

```
try {
  if (!removed) entityBean.ejbPassivate();
  setState( POOLED );
} catch (RemoteException ex ) {
  FFDCEngine.processException(
    ex,"EBean.passivate()","237",
    this);
  destroy();
  throw ex;
} finally {
  if (!removed &&
    statisticsCollector != null ) {
    statisticsCollector.
       recordPassivation();
  }
  removed = false;
  beanPool.put( this );
  if (Logger.isEnabled) {
    Logger.exit(tc,"passivate");
  }
}
```

Before

```
try {
  if (!removed) entityBean.ejbPassivate();
  setState( POOLED );
} catch (RemoteException ex ) {
  destroy();
  throw ex;
} finally {
  removed = false;
  beanPool.put( this );
}
```

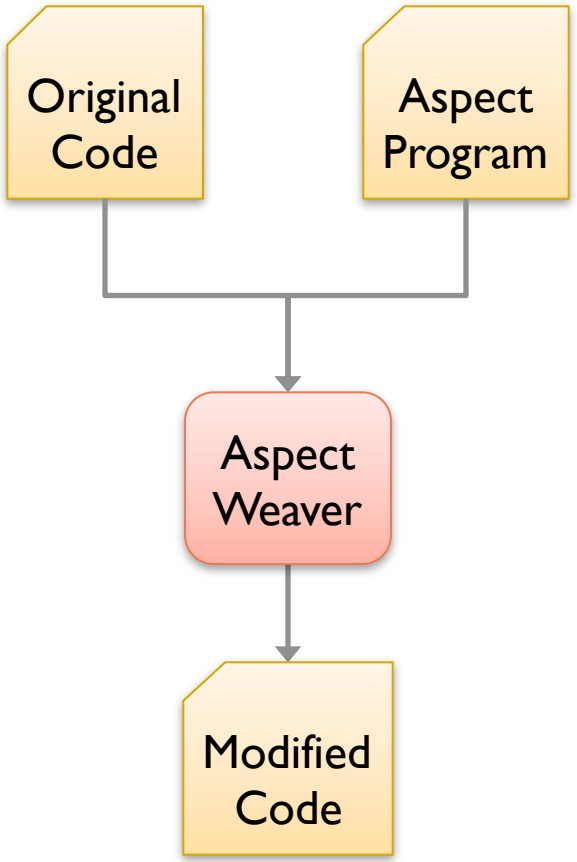
After

Crosscutting concerns extracted

Example: Code to handle EJB Entity bean passivation

How It's Done: Weaving

- The goal of AOP is to capture crosscuts in a modular way with new language constructs
- These concerns are separated out as "aspects"
- An aspect "weaver" is responsible for integrating the separated concerns into the resulting executable



AspectJ is…

"an aspect-oriented extension to Java that supports general-purpose aspect-oriented programming"

- 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
  - AJDT support for Eclipse
- Freely available implementation
  - Compiler & tools are Open Source
- Active user community
  - aspectj-users@eclipse.org

Gregor Kiczales

Basic Mechanisms

- FOUR small additions to Java
  - Pointcuts
    - Pick out join points and values at those points
    - Primitive, user-defined pointcuts
  - Advice
    - Additional action to take at join points in a pointcut
  - Inter-type declarations
  - Aspect
    - A modular unit of crosscutting behavior comprised of advice, inter-type, pointcut, field, constructor, and method declarations

Join Point Terminology

- All kinds of events could be join points but AspectJ exposes this set:
  - Method & constructor call
  - Method & constructor execution
  - Field get & set
  - Exception handler execution
  - Static & dynamic initialization

Primitive Pointcuts

- A pointcut identifies some set of join points
  - Can match or not match any given join point, and
  - Optionally, can pull out some of the values ('context') at that join point

```
call(void Line.setP1(Point))
```

Matches if the join point is a method call with this signature 'void Line.setP1(Point)'

Pointcut Composition

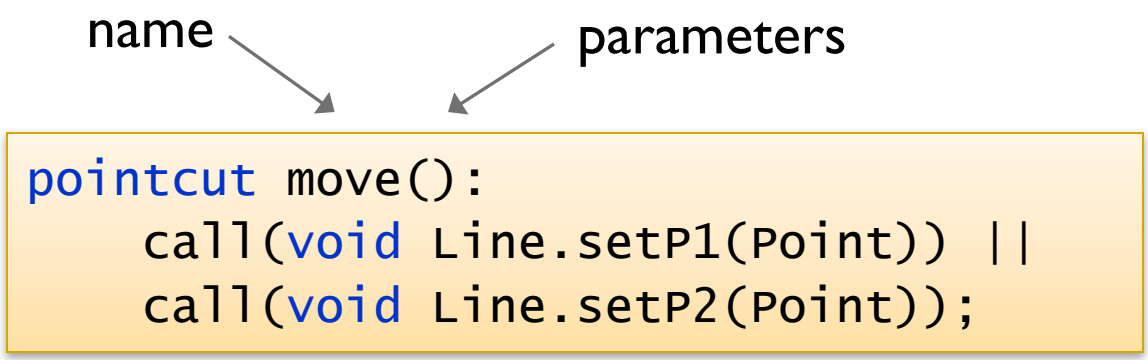
- Pointcuts can be composed
  - Using &&, ||, and !

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

matches either a call to 'void Line.setP1(Point)'
or a call to 'void Line.setP2(Point)'

User-defined Pointcut

- User-defined (*aka* named) pointcuts
  - Can be used in the same way as primitive pointcuts



Pointcuts

- Primitive pointcuts are

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

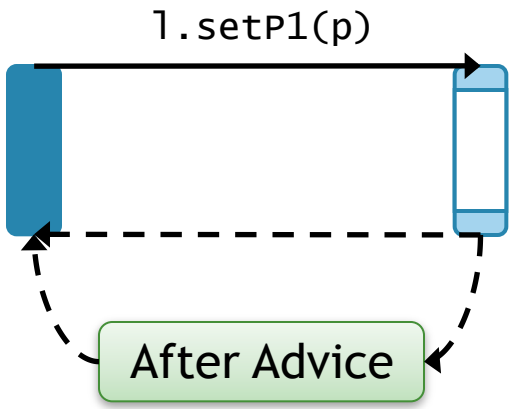
- And with AspectJ5

- @annotation, @within, @withincode, @this, @target, @args

Advice

- Additional action to take at join points in a pointcut

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

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



Advice

- `before`
  - Before proceeding at join point
- `after returning`
  - After returning a value to join point
- `after throwing`
  - 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 and if program proceeds

A Simple Aspect

- `DisplayUpdating`
  - An aspect defines a 'special module' that can crosscut other classes

```
aspect DisplayUpdating {

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

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

Without AspectJ

- What you would expect
  - Update calls are tangled through the code
  - "What is going on" is less explicit

```
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();
  }
}
```

With AspectJ

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

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

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

```
aspect DisplayUpdating {

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

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

Without AspectJ

```
class Line {
  private Point _p1, _p2;
  ...
}
```

```
class XYZZY {
  ...
  public void moveit(Point p) {
    MoveTracking.collectMove(this, p);
    p.setX(p.getX()+1);
    ...
  }
  ...
}
```

```
class Bar {
  ...
  public void mumble(Line l) {
    MoveTracking.collectMove(this, l);
    l.setP1(new Point());
    ...
  }
  ...
}
```

```
class MoveTracking {
  private static Set _movees = new HashSet();
  private static Set _movers = new HashSet();

  public static void collectMove(Object o1, Object o2) {
    _movees.add(o1);
    _movers.add(o2);
  }

  public static Set getmovees() {
    result = _movees;
    ees = new HashSet();
    rn result;
```

```
class AnotherClass {
  ...
  public void frotz(Point p) {
    MoveTracking.collectMove(this, p);
    p.setY(42);
    ...
  }
  ...
}
```

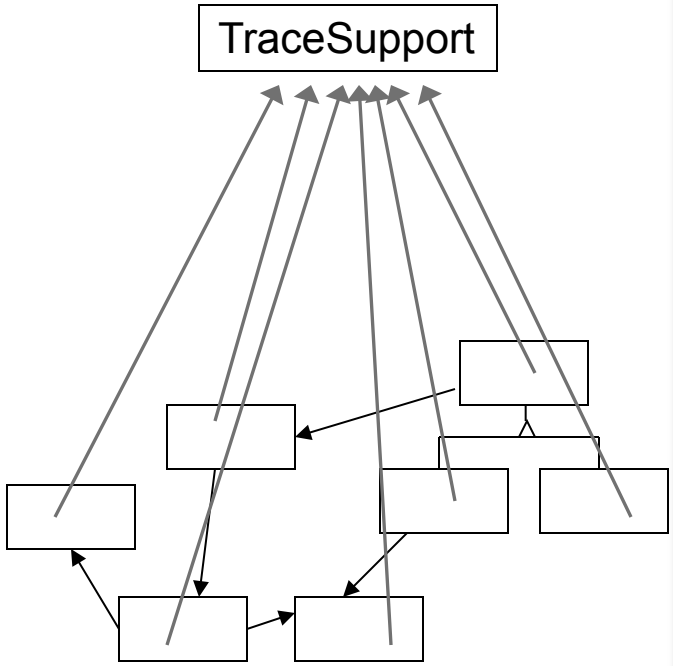
```
class Baz
  ...
  public void mumble(Line l) {
    MoveTracking.collectMove(this, l);
    l.setP1(new Point());
    ...
  }
  ...
}
```

```
) {
  public void mumble(Line l) {
    MoveTracking.collectMove(this, l);
    l.setP1(new Point());
    ...
```

```
class Foo {
  ...
  public void mbar(Line l) {
    MoveTracking.collectMove(this, l);
    l.setP1(new Point());
    ...
  }
  ...
}
```

```
  public void mumble(Line l) {
    MoveTracking.collectMove(this, l);
    l.setP1(new Point());
    ...
  }
  ...
}
```

Tracing without AspectJ

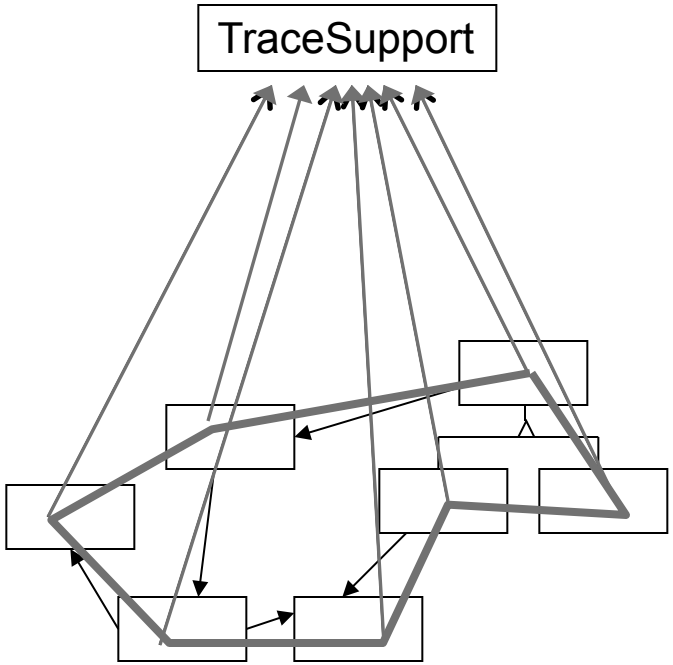



```
class TraceSupport {
  static int TRACELEVEL = 0;
  static protected PrintStream stream = null;
  static protected int callDepth = -1;

  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);
  }
}
```

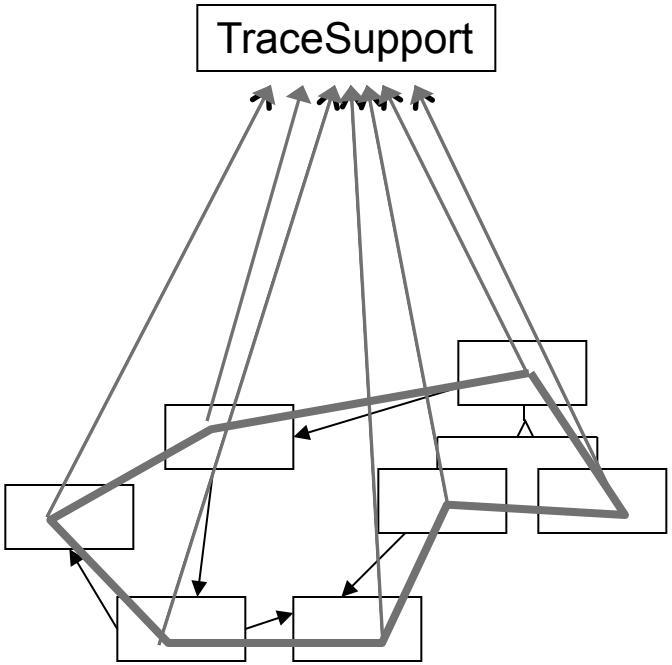
```
class Point {
  void set(int x, int y) {
    TraceSupport.traceEntry("Point.set");
    this.x = x; this.y = y;
    TraceSupport.traceExit("Point.set");
  }
}
```

A Clear Crosscutting Structure

- All modules of the system use the trace facility in a consistent way:
  - Entering the methods and exiting the methods



Tracing as an Aspect




```
aspect PointTracing {

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

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

Tracing – Using a Library Aspect

```
aspect BigBoxCoTracing {

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

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

```
abstract aspect Tracing {
  abstract pointcut trace();

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

```
aspect BigBoxCoTracing
    extends Tracing {

  pointcut trace():
      within(com.bigboxco.*)
      && execution(* *(..));
}
```

Expected Benefits of 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