Aspect-Oriented Programming

Harald Gall
University of Zurich
seal.ifi.uzh.ch/ase

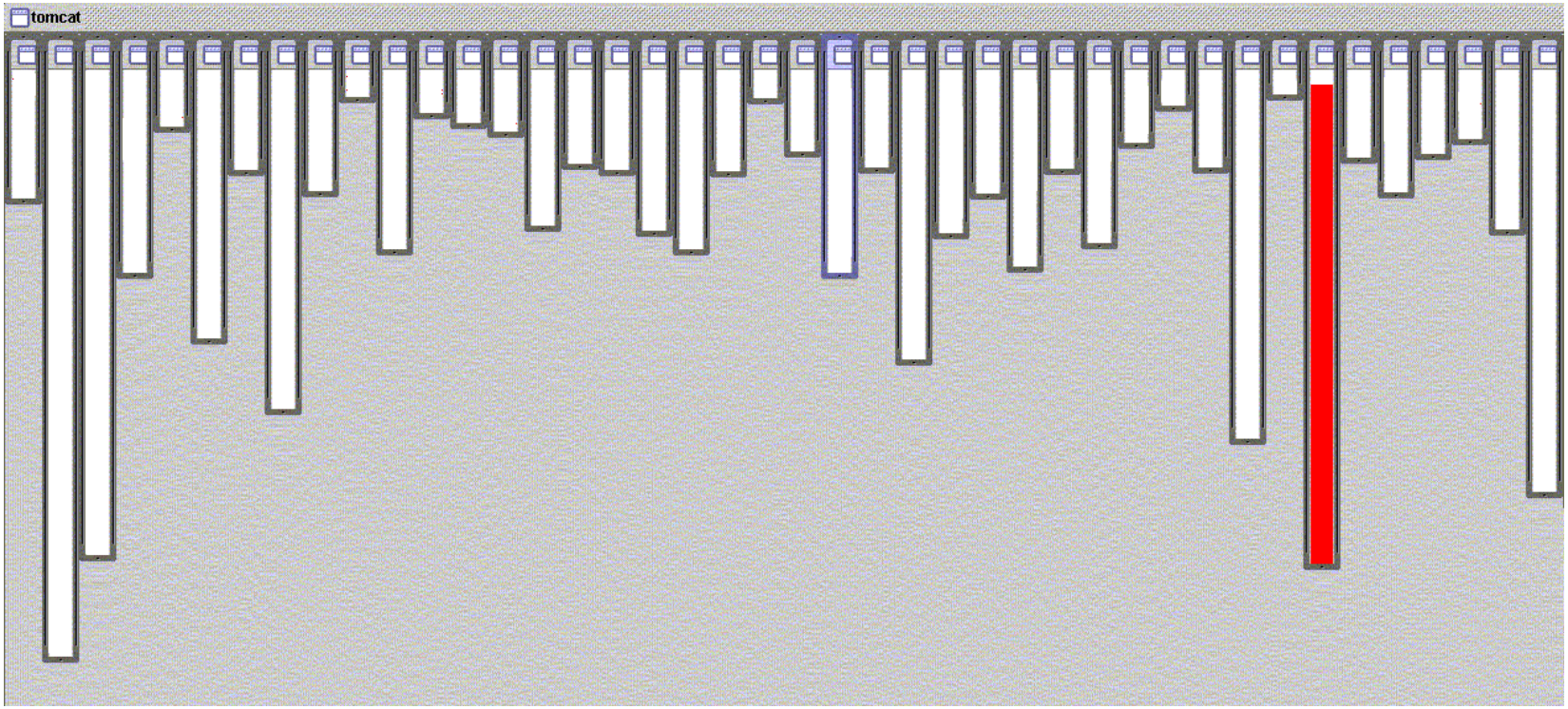
Source:
http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

Programming paradigms

- Procedural programming
  - Executing a set of commands in a given sequence
  - Fortran, C, Cobol
- Functional programming
  - Evaluating a function defined in terms of other functions
  - Lisp, ML, Scheme
- Logic programming
  - Proving a theorem by finding values for the free variables
  - Prolog
- Object-oriented programming (OOP)
  - Organizing a set of objects, each with its own responsibilities
  - Smalltalk, Java, C++ (to some extent)
- Aspect-oriented programming (AOP)
  - Executing code whenever a program shows certain behaviors
  - AspectJ (a Java extension)
  - Does not *replace* O-O programming, but rather *complements* it

good modularity

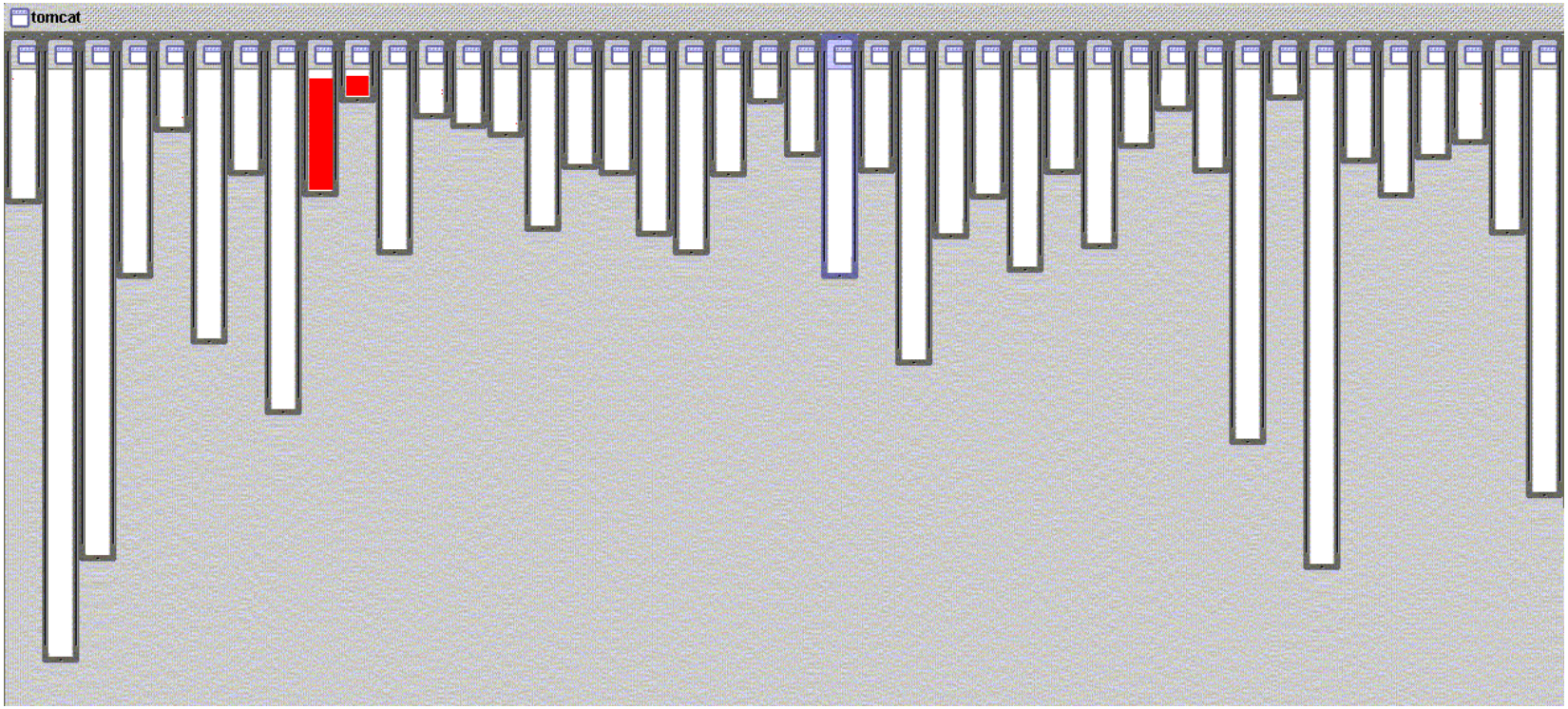
XML parsing



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

good modularity

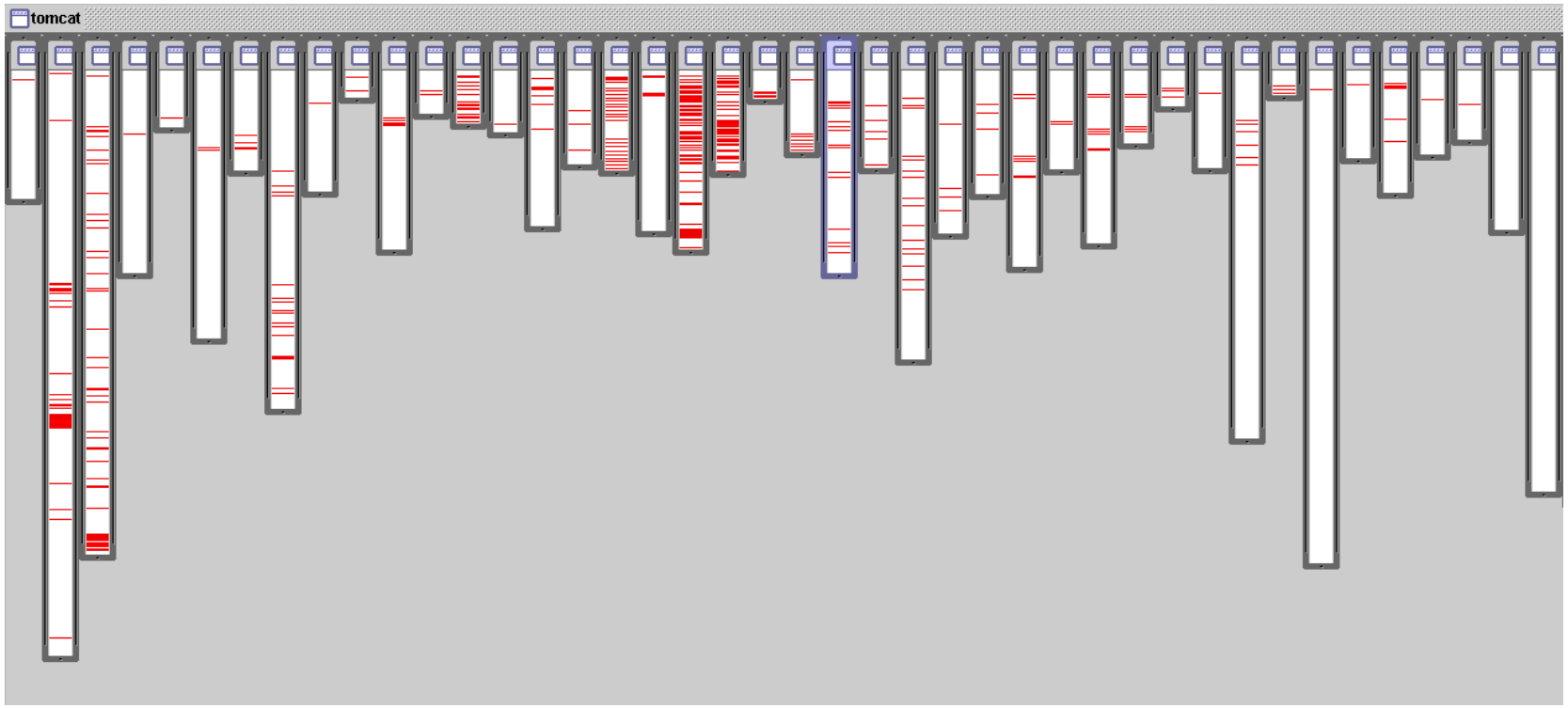
URL pattern matching



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

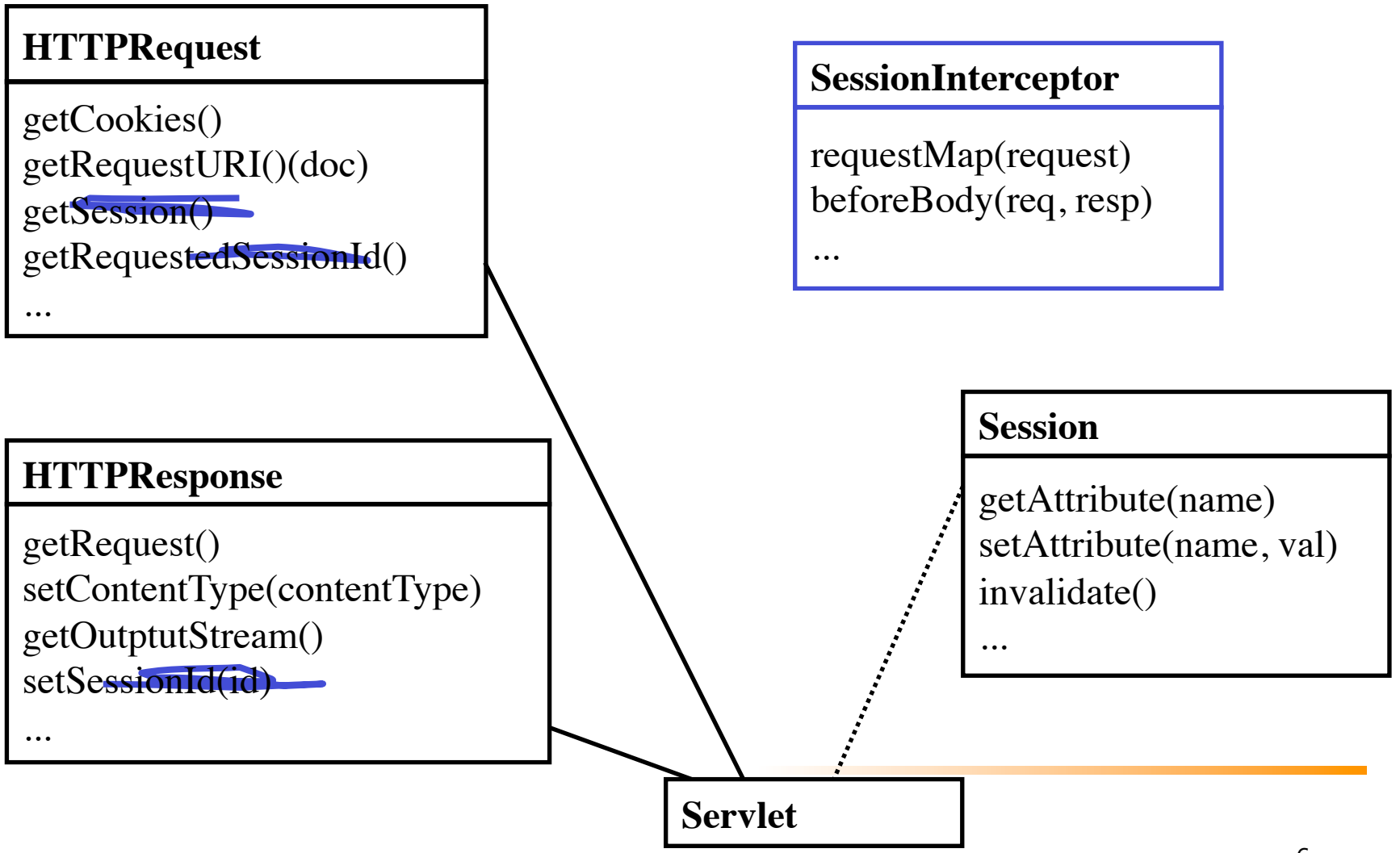
problems like…

logging is not modularized



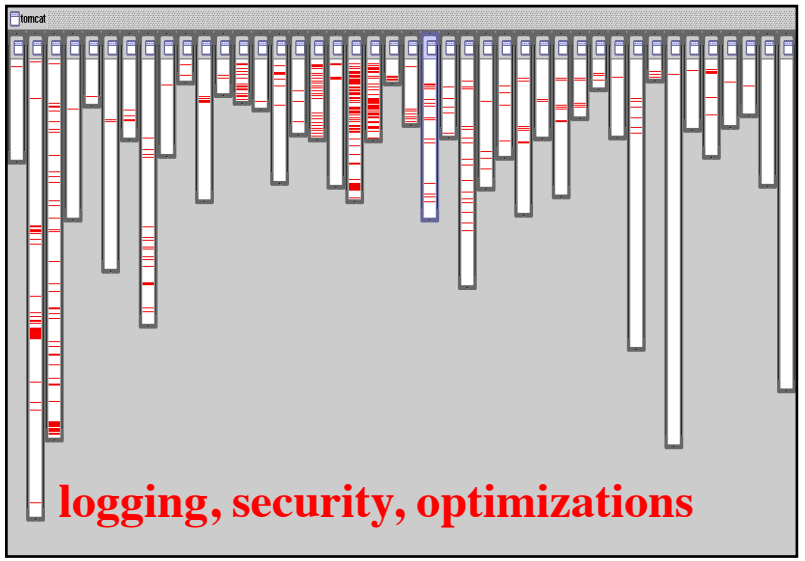
- where is logging in org.apache.tomcat
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places

problems like… session tracking is not modularized



The problem of crosscutting concerns

- critical aspects of large systems do not fit in traditional modules
  - logging, error handling
  - synchronization
  - security
  - power management
  - memory management
  - performance optimizations
- tangled code has a cost
  - difficult to understand
  - difficult to change
  - increases with size of system
  - maintenance costs are huge
- good programmers work hard to get rid of tangled code
  - the last 10% of the tangled code causes 90% of the development and maintenance headaches



The AOP idea

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
- Aspect-Oriented Software Development: AO support throughout lifecycle

Example

```
class Fraction {
    int numerator;
    int denominator;
    ...
    public Fraction multiply(Fraction that) {
        traceEnter("multiply", new Object[] {that});
        Fraction result = new Fraction(
            this.numerator * that.numerator,
            this.denominator * that.denominator);
        result = result.reduceToLowestTerms();
        traceExit("multiply", result);
        return result;
    }
    ...
}
```

- Now imagine similar code in *every method* you might want to trace

Logging Example

```
import com.foo.Bar;
// Import log4j classes.
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class MyApp {
        // Define a static logger variable so that it references the
        // Logger instance named "MyApp".
        static Logger logger = Logger.getLogger(MyApp.class);

        public static void main(String[] args) {
                // Set up a simple configuration that logs on the console.
                BasicConfigurator.configure();

                logger.setLevel(Level.DEBUG); // optional if log4j.properties r
                // Possible levels: TRACE, DEBUG, INFO, WARN, ERROR, and FATAL

                logger.info("Entering application.");
                Bar bar = new Bar();
                bar.doIt();
                logger.info("Exiting application.");
        }
}
```

Consequences of crosscutting code

- Redundant code
  - Same fragment of code in many places
- Difficult to reason about
  - No 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
- Inefficient when crosscuting code is not needed

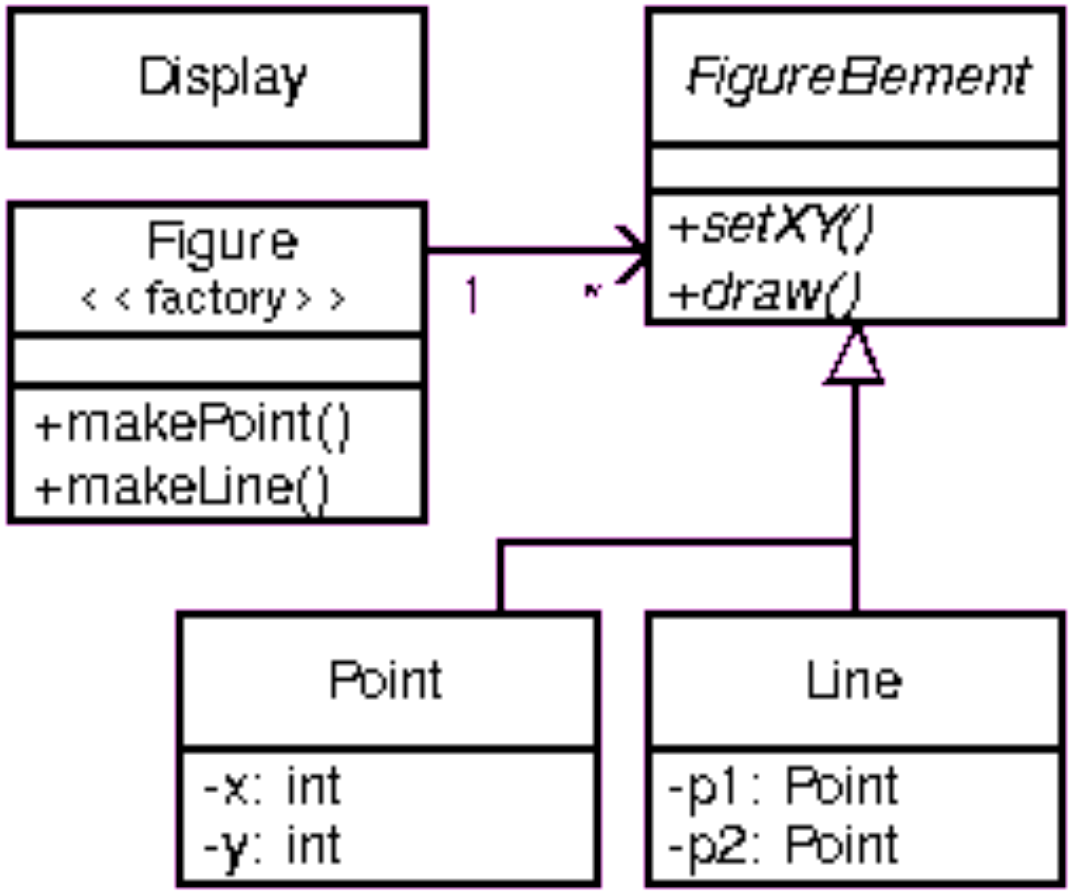
AspectJ™

- AspectJ is a small, well-integrated extension to Java
  - Based on the 1997 PhD thesis by Christina Lopes, *A Language Framework for Distributed Programming*
- AspectJ modularizes crosscutting concerns
  - That is, code for one *aspect* of the program (such as tracing) is collected together in one place
  - The AspectJ compiler is free and open source
  - AspectJ works with JBuilder, Forté, Eclipse, etc.
- Best online writeup: http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html
  - Parts of this lecture were taken from the above paper

Terminology

- A join point is a well-defined point in the program flow
- A pointcut is a group of join points
- Advice is code that is executed at a pointcut
- Introduction modifies the members of a class and the relationships between classes
- An aspect is a module for handling crosscutting concerns
  - Aspects are defined in terms of pointcuts, advice, and introduction
  - Aspects are reusable and inheritable
- Each of these terms will be discussed in greater detail

The Figure Element example



Example I

- A pointcut named move that chooses various method calls:
  - pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));
- Advice (code) that runs before the move pointcut:
  - before(): move() {
    System.out.println("About to move");}
- Advice that runs after the move pointcut:
  - after(): move() {
    System.out.println("Just successfully moved");}

Join points

- A join point is a well-defined point in the program flow
  - We want to execute some code ("advice") each time a join point is reached
  - We do *not* want to clutter up the code with explicit indicators saying *"This is a join point"*
  - AspectJ provides a syntax for indicating these join points "from outside" the actual code
- A join point is a point in the program flow "where something happens"
  - Examples:
    - When a method is called
    - When an exception is thrown
    - When a variable is accessed

Pointcuts

- Pointcut definitions consist of a left-hand side and a right-hand side, separated by a colon
  - The left-hand side consists of the pointcut name and the pointcut parameters (i.e. the data available when the events happen)
  - The right-hand side consists of the pointcut itself
- Example pointcut:
  `pointcut setter(): call(void setX(int));`
  - The name of this pointcut is `setter`
  - The pointcut has no parameters
  - The pointcut itself is `call(void setX(int))`
  - The pointcut refers to any time the `void setX(int)` method is called

Example pointcut designators I

- When a particular method body executes:
  - execution(void Point.setX(int))
- When a method is called:
  - call(void Point.setX(int))
- When an exception handler executes:
  - handler(ArrayOutOfBoundsException)
- When the object currently executing (i.e. this) is of type SomeType:
  - this(SomeType)

Example pointcut designators II

- When the target object is of type SomeType
  - target(SomeType)
- When the executing code belongs to class MyClass
  - within(MyClass)
- When the join point is in the control flow of a call to a Test's no-argument main method
  - cflow(call(void Test.main()))

Pointcut designator wildcards

- It is possible to use wildcards to declare pointcuts:
  - execution(* *(..))
    - Chooses the execution of any method regardless of return or parameter types
  - call(* set(..))
    - Chooses the call to any method named set regardless of return or parameter type
    - In case of overloading there may be more than one such set method; this pointcut picks out calls to all of them

Pointcut designators based on types

- You can select elements based on types, e.g.
  - execution(int *())
    - Chooses the execution of any method with no parameters that returns an int
  - call(* setY(long))
    - Chooses the call to any setY method that takes a long as an argument, regardless of return type or declaring type
  - call(* Point.setY(int))
    - Chooses the call to any of Point's setY methods that take an int as an argument, regardless of return type
  - call(*.new(int, int))
    - Chooses the call to any classes' constructor, so long as it takes exactly two ints as arguments

Pointcut designator composition

- Pointcuts compose through the operations **or** (“||”), **and** (“&&”) and **not** (“!”)
- Examples:
  - target(Point) && call(int *())
    - Chooses any call to an int method with no arguments on an instance of Point, regardless of its name
  - call(* *(..)) && (within(Line) || within(Point))
    - Chooses any call to any method where the call is made from the code in Point’s or Line’s type declaration
  - within(*) && execution(*.new(int))
    - Chooses the execution of any constructor taking exactly one int argument, regardless of where the call is made from
  - !this(Point) && call(int *(..))
    - Chooses any method call to an int method when the executing object is any type except Point

Pointcut designators based on modifiers

- call(public * *(..))
  - Chooses any call to a public method
- execution(!static * *(..))
  - Chooses any execution of a non-static method
- execution(public !static * *(..))
  - Chooses any execution of a public, non-static method

- Pointcut designators can be based on interfaces as well as on classes

Example I, repeated

- A pointcut named move that chooses various method calls:
  - pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int)) ||
    call(void Point.setY(int)) ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));
- Advice (code) that runs before the move pointcut:
  - before(): move() {
    System.out.println("About to move"); }
- Advice that runs after the move pointcut:
  - after(): move() {
    System.out.println("Just successfully moved"); }

Kinds of advice

- AspectJ has several kinds of advice:
  - Before advice runs as a join point is reached, before the program proceeds with the join point
  - After advice on a particular join point runs after the program proceeds with that join point
    - after returning advice is executed after a method returns normally
    - after throwing advice is executed after a method returns by throwing an exception
    - after advice is executed after a method returns, regardless of whether it returns normally or by throwing an exception
  - Around advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point

Example II, with parameters

- You can access the context of the join point:

```
pointcut setXY(FigureElement fe, int x, int y):
    call(void FigureElement.setXY(int, int))
    && target(fe)
    && args(x, y);
```

```
after(FigureElement fe, int x, int y) returning: setXY(fe, x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ").");
}
```

Introduction

- An introduction is a member of an aspect, but it defines or modifies a member of another type (class). With introduction we can
  - add methods to an existing class
  - add fields to an existing class
  - extend an existing class with another
  - implement an interface in an existing class
  - convert checked exceptions into unchecked exceptions

Example introduction

- aspect CloneablePoint {

  declare parents: Point implements Cloneable;

  declare soft: CloneNotSupportedException:
  execution(Object clone());

  Object Point.clone() { return super.clone(); }
  }

Approximate syntax

- An aspect is: aspect ***nameOfAspect*** { ***body*** }
  - An aspect contains introductions, pointcuts, and advice
- A pointcut designator is: ***when*(*signature*)**
  - The signature includes the return type
  - The "***when***" is call, handler, execution, etc.
- A named pointcut designator is:
  name*(*parameters*): *pointcutDesignator
- Advice is:
  adviceType*(*parameters*): *pointcutDesignator
  { ***body*** }
- Introductions are basically like normal Java code

Example aspect I

```
aspect PointWatching {
   private Vector Point.Watchers = new Vector();

   public static void addWatcher(Point p, Screen s) {
      p.Watchers.add(s);
   }

   public static void removeWatcher(Point p, Screen s) {
      p.Watchers.remove(s);
   }

   static void updateWatcher(Point p, Screen s) {
      s.display(p);
   }
// continued on next slide
```

Example aspect II

- // continued from previous slide

```
    pointcut changes(Point p): target(p) && call(void
Point.set*(int));

    after(Point p): changes(p) {
        Iterator iter = p.Watchers.iterator();
        while ( iter.hasNext() ) {
            updateWatcher(p, (Screen)iter.next());
        }
    }
}
```

Simple tracing

```
aspect SimpleTracing {
    pointcut tracedCall():
        call(void FigureElement.draw(GraphicsContext));

    before(): tracedCall() {
        System.out.println("Entering: " + thisJoinPoint);
    }
}
```


32

Checking pre- and post-conditions

```
aspect PointBoundsChecking {

    pointcut setX(int x):
        (call(void FigureElement.setXY(int, int)) && args(x, *))
        || (call(void Point.setX(int)) && args(x));

    pointcut setY(int y):|
        (call(void FigureElement.setXY(int, int)) && args(*, y))
       || (call(void Point.setY(int)) && args(y));

    before(int x): setX(x) {
        if ( x < MIN_X || x > MAX_X )
            throw new IllegalArgumentException("x is out of bounds.");
    }

    before(int y): setY(y) {
        if ( y < MIN_Y || y > MAX_Y )
            throw new IllegalArgumentException("y is out of bounds.");
    }
}
```


33

Updates

- The preceding slides, while accurate enough, do not reflect the most recent changes in AspectJ
- Good reference: **The AspectJ™ 5 Development Kit Developer's Notebook**
  - http://www.eclipse.org/aspectj/doc/released/adk15notebook/

34

Concluding remarks

- Aspect-oriented programming (AOP) is a new paradigm--a new way to think about programming
- AOP is somewhat similar to event handling, where the "events" are defined outside the code itself
- AspectJ is not itself a complete programming language, but an adjunct to Java
- AspectJ does not add new capabilities to what Java can do, but adds new ways of modularizing the code
- AspectJ is free, open source software
- Like all new technologies, AOP may--or may not--catch on in a big way