EECE 310 – Software Engineering

Aspect-Oriented Programming

Slides based on:

Gregor Kiczales, Jorrit N. Herder, Mik Kersten

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, Ocaml, JavaScript
- 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 set of 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 *complements* it

OOP benefits

- OOP builds on existing paradigms
- Elegant programming solutions
  - Inheritance (specialized classes)
  - Dynamic binding (polymorphic behavior)
- Maintenance becomes easier
  - Object model is rather stable
- Reuse of components is easy

OOP shortcomings

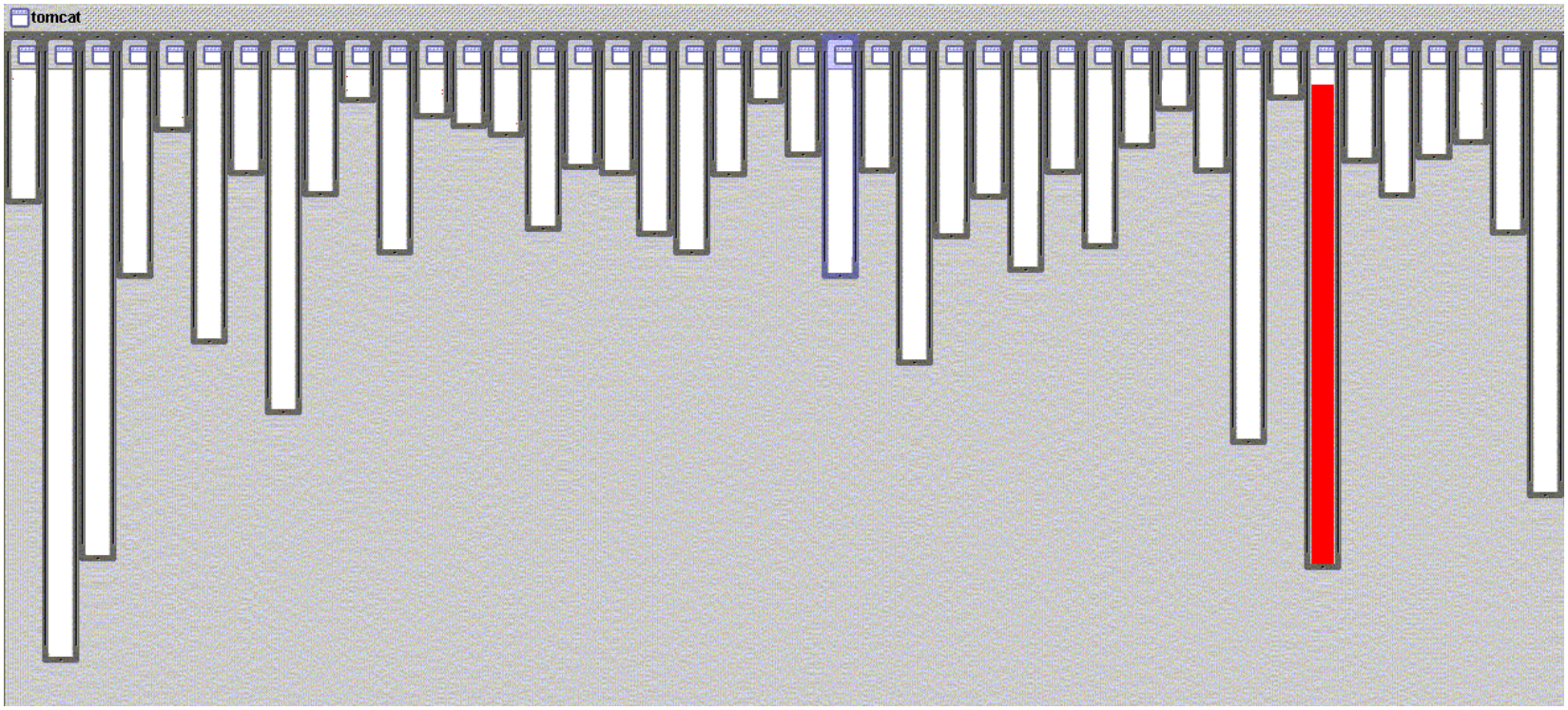
- While OOP allows to decompose a system into units of function or behaviour
- certain software properties **cannot** be isolated in a single functional unit, instead they *crosscut* multiple components
- Such **crosscutting concerns** result in *tangled code* that is **hard to maintain**

Aspect-Oriented Programming

- Definition: "... to support the programmer in cleanly **separating** components and aspects from each other, by providing mechanisms that allow to **abstract** and **compose** them to produce the overall system."

good modularity

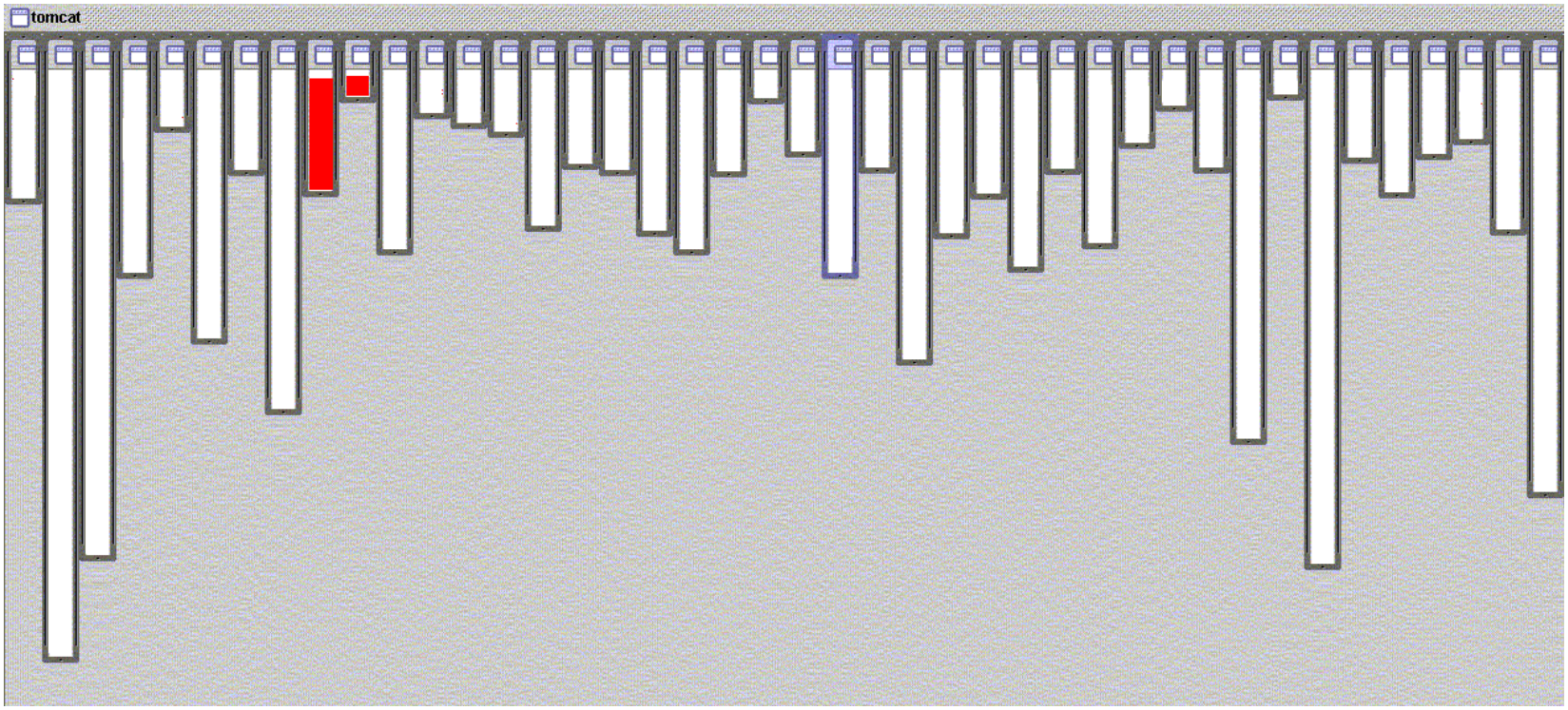
XML parsing



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

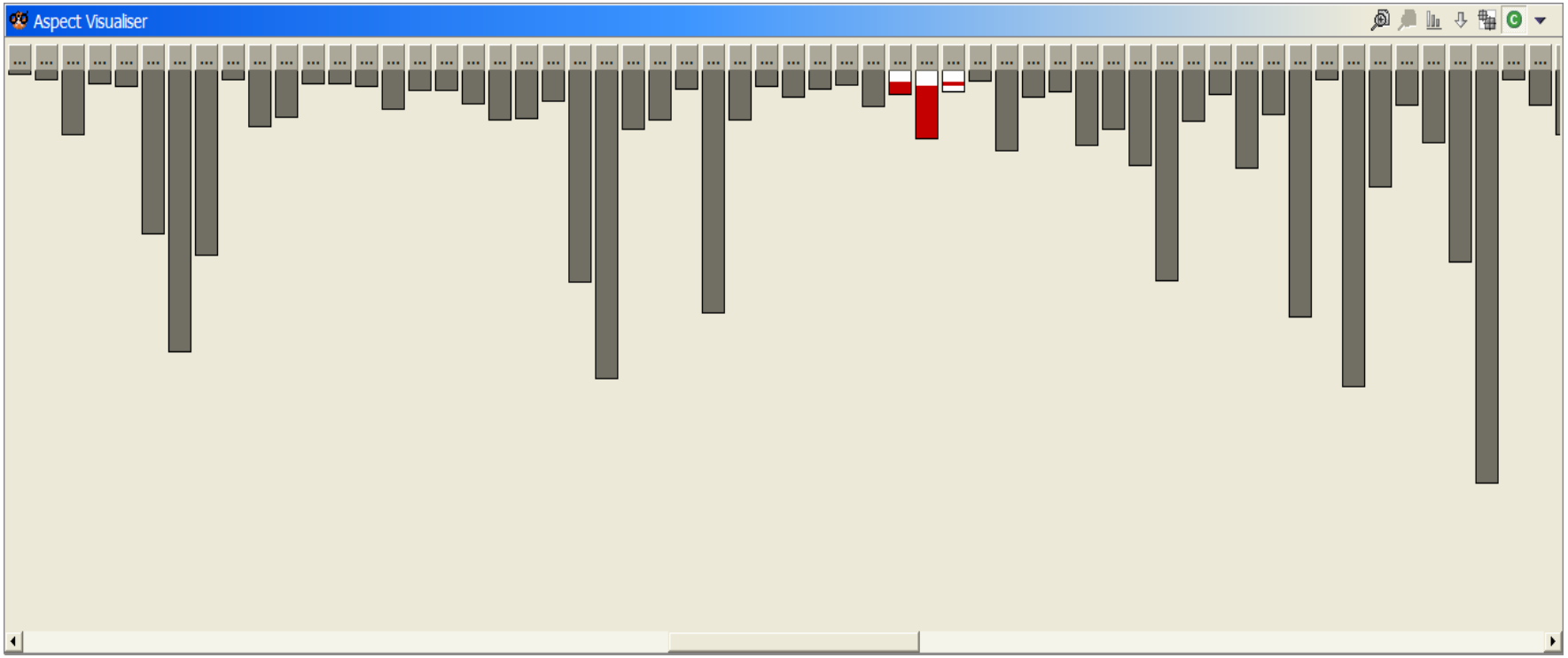
good modularity

URL pattern matching



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

good modularity

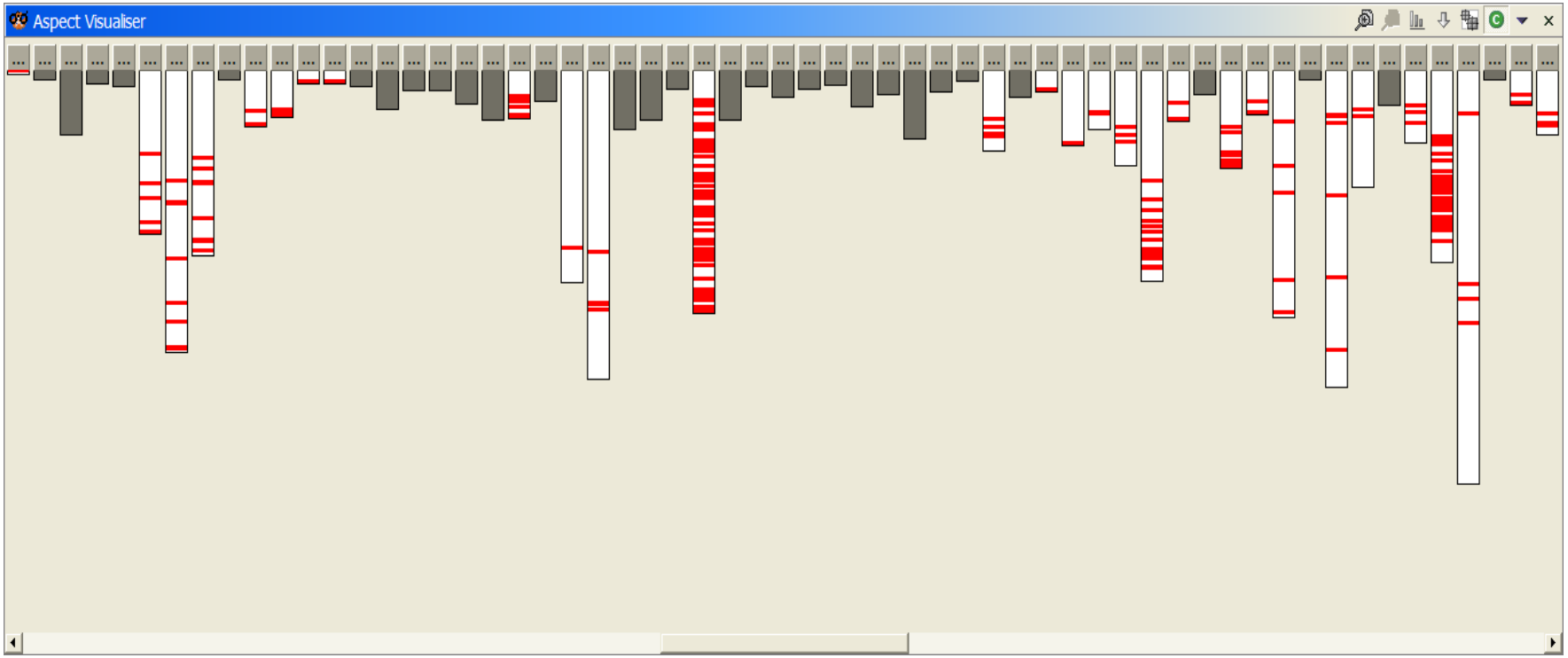


- socket creation in Tomcat
  - colored lines show relevant lines of code
  - fits nicely into one package (3 classes)

pretty good modularity

- class loading in Tomcat
  - colored lines show relevant lines of code
  - mostly in one package (9 classes)

not so good modularity



- logging in Tomcat
  - scattered across the packages and classes
  - error handling, security, business rules, …

logging, zoomed in

//From ContextManager

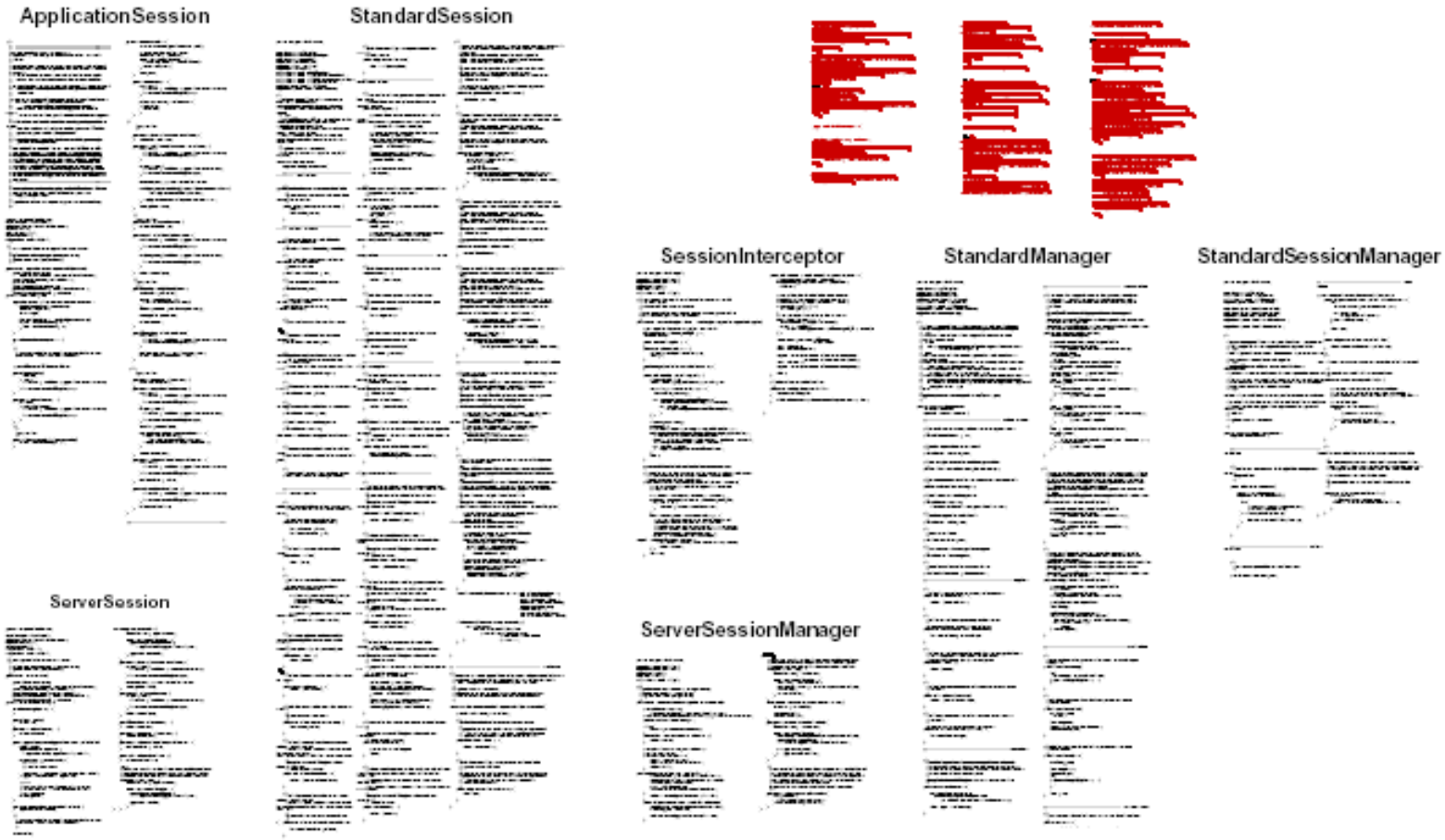
```
public void service( Request rrequest, Response rresponse ) {
   log( "New  request " + rrequest );
 try {
     //     System.out.print("A");
     rrequest.setContextManager( this );
     rrequest.setResponse(rresponse);
     rresponse.setRequest(rrequest);

     // wront request - parsing error
     int status=rresponse.getStatus();

     if( status < 400 )
  status= processRequest( rrequest );

     if(status==0)
  status=authenticate( rrequest, rresponse );
     if(status == 0)
  status=authorize( rrequest, rresponse );
     if( status == 0 ) {
  rrequest.getWrapper().handleRequest(rrequest,
          rresponse);
     } else {
  // something went wrong
  handleError( rrequest, rresponse, null, status );
     }
 } catch (Throwable t) {
     handleError( rrequest, rresponse, t, 0 );
 }
 // System.out.print("B");
 try {
     rresponse.finish();
     rrequest.recycle();
     rresponse.recycle();
 } catch( Throwable ex ) {
     if(debug>0) log( "Error closing request " + ex);
 }
  log( "Done with request " + rrequest );
 // System.out.print("C");
 return;
}
```

```
log( "New  request " + rrequest );

// System.out.print("A");

// System.out.print("B");

if(debug>0)
    log("Error closing request " + ex);

log("Done with request " + rrequest);

// System.out.print("C");
```

Without AOP


ApplicationSession
StandardSession
SessionInterceptor
StandardManager
StandardSessionManager
ServerSession
ServerSessionManager

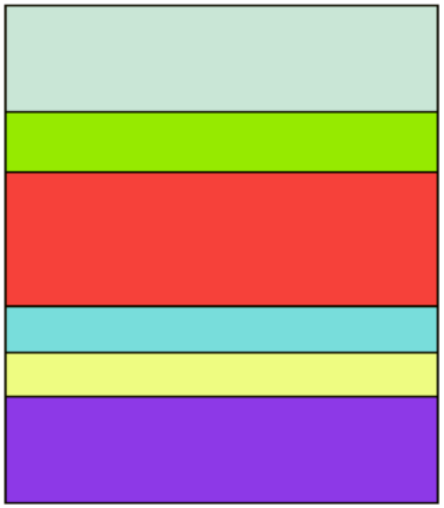
With AOP


ApplicationSession
StandardSession
SessionInterceptor
StandardManager
StandardSessionManager
ServerSession
ServerSessionManager

Two problems AOP tries to resolve

Code Tangling

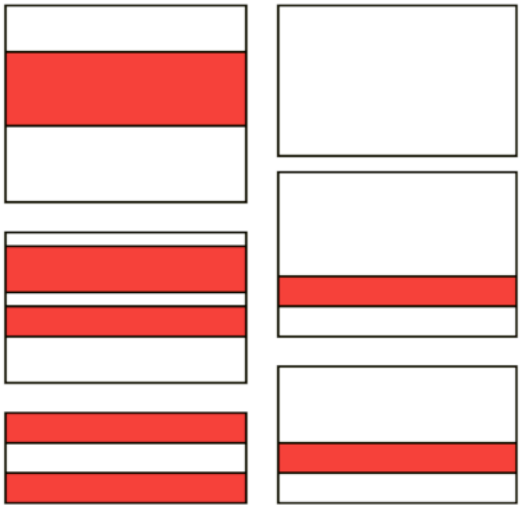
One module, many concerns, mixed with core business logic



logging

Code Scattering

One concern, many modules



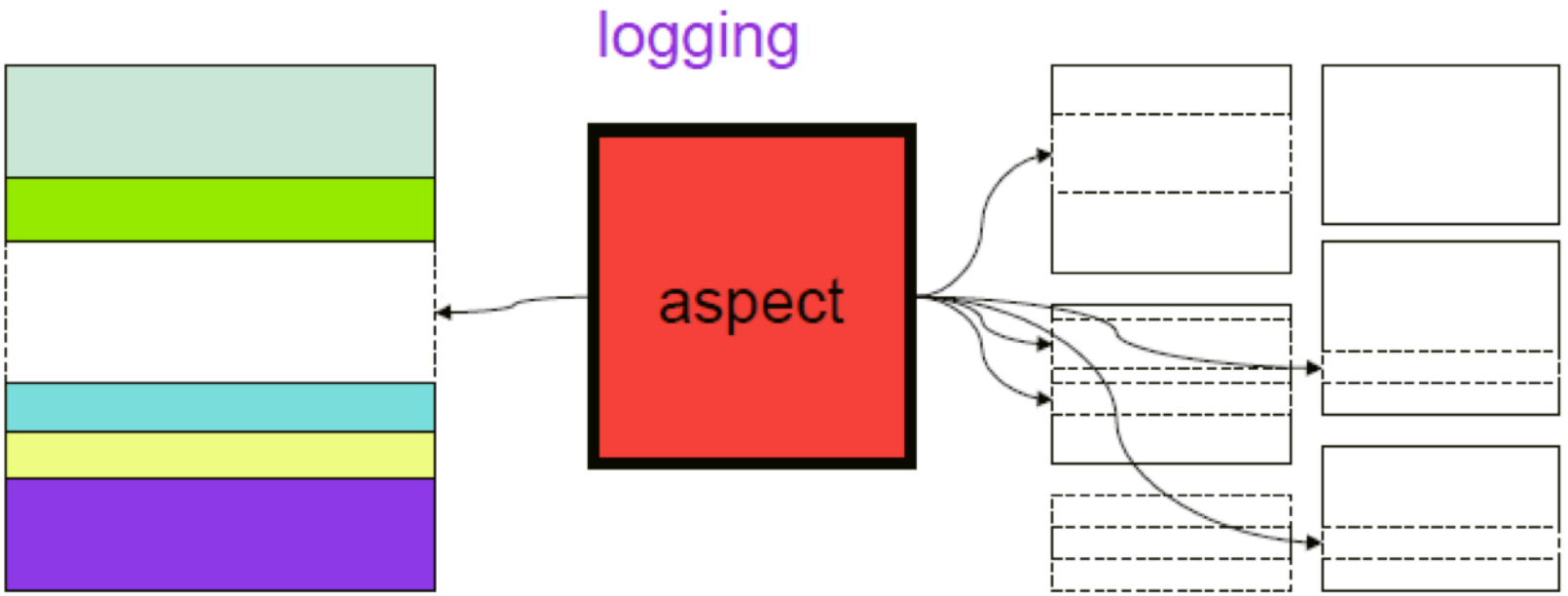
Two problems AOP tries to resolve

Code Tangling

One module, many concerns, mixed with core business logic

Code Scattering

One concern, many modules



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

The problem

- Some programming tasks **cannot** be neatly encapsulated in objects, but must be scattered throughout the code
- Examples:
  - Logging (tracking program behavior to a file)
  - Profiling (determining where a program spends its time)
  - Tracing (determining what methods are called when)
  - Session tracking, session expiration
  - Special security management
- The result is **crosscuting code**--the necessary code "cuts across" many different classes and methods

Crosscutting Concern

- Behavior that cuts across the typical divisions of responsibility, such as logging or debugging
- A problem which a program tries to solve.
- Aspects of a program that **do not relate to the core concerns** directly, but which proper program execution nevertheless requires.

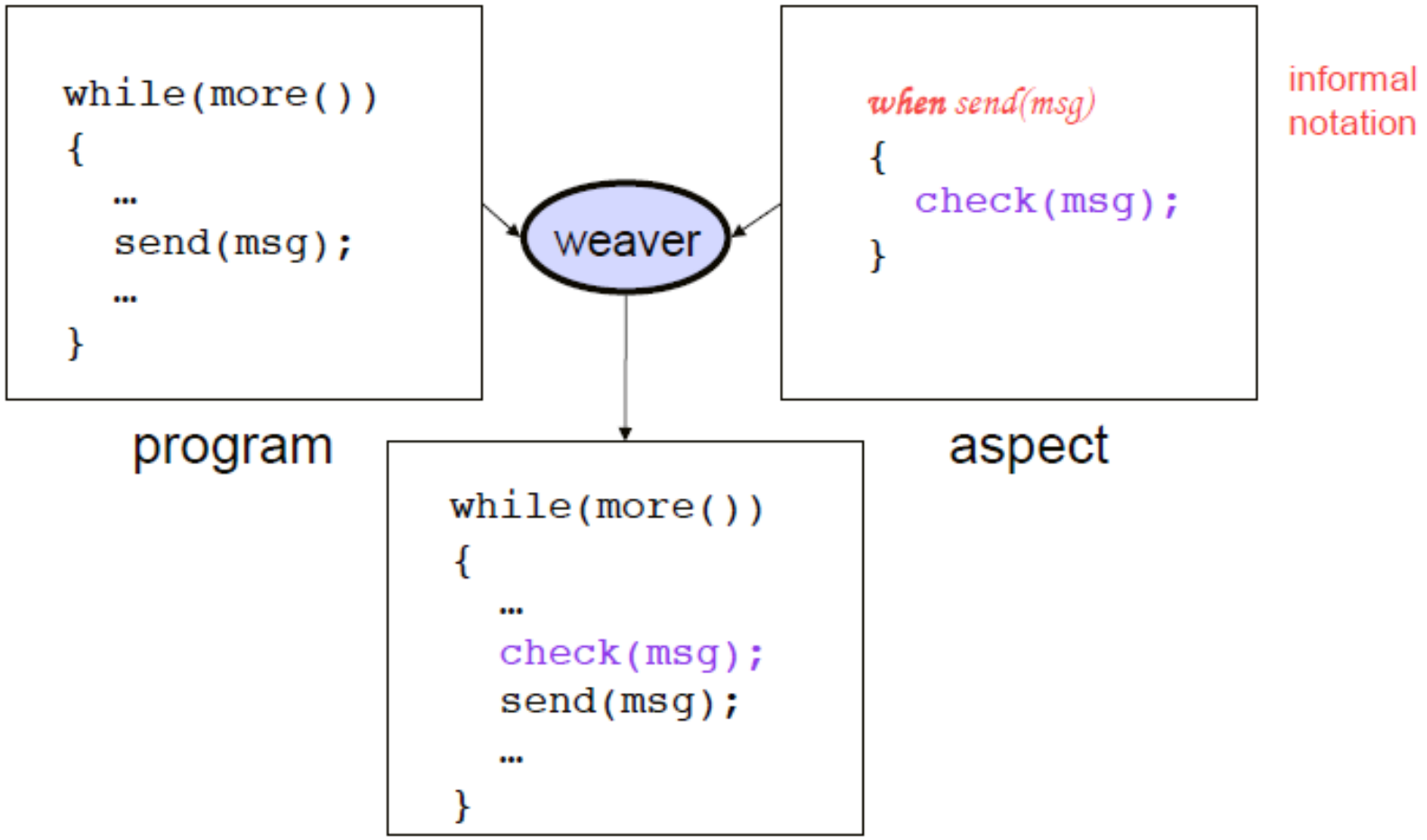
What's the solution then?

- How can me modularize crosscutting concerns?

The AOP Idea

- crosscutting is inherent in complex systems
- crosscutting concerns
  - have a **clear purpose**
  - have a **natural structure**
- 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

very simplified view of AOP



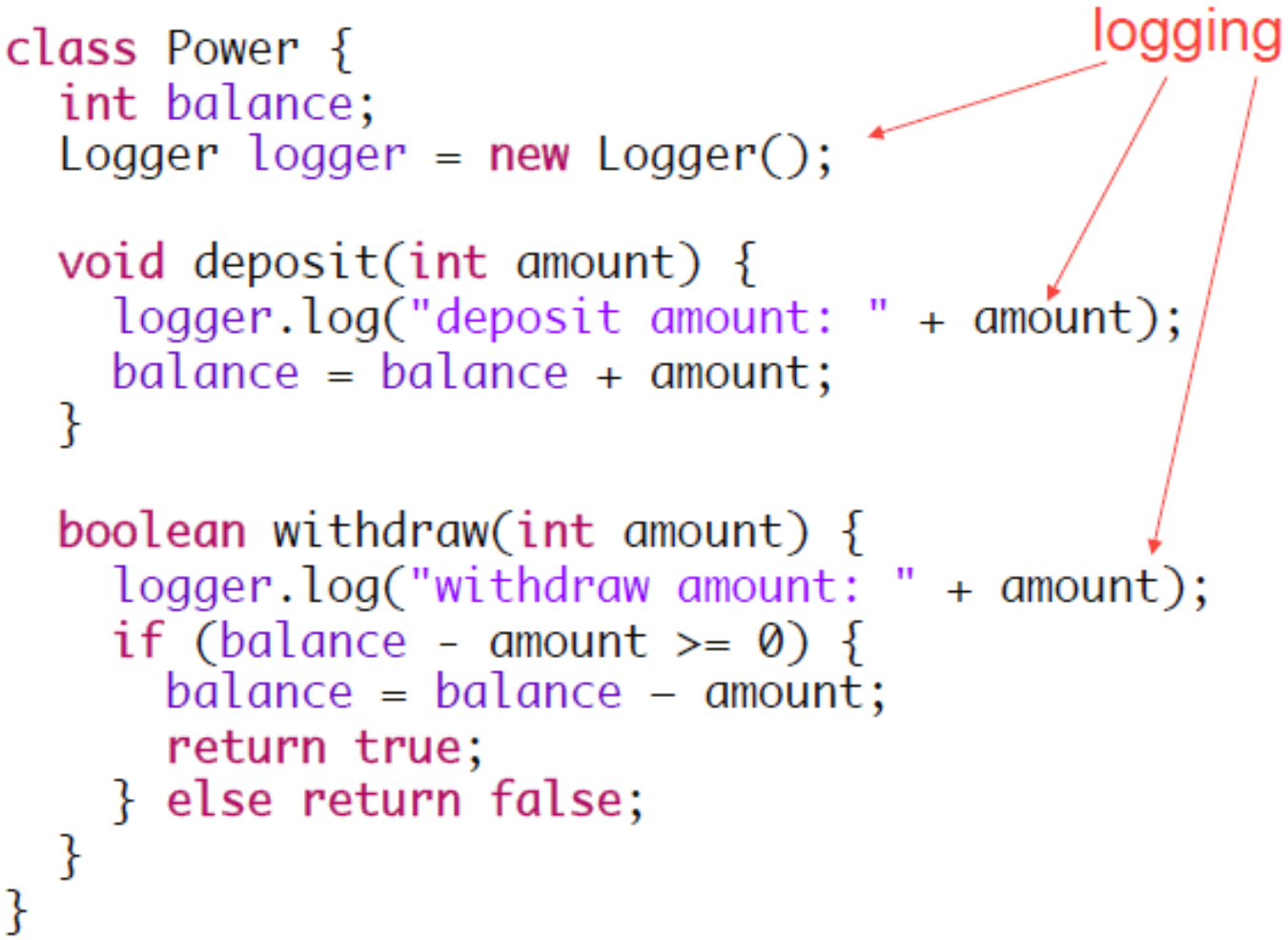
example

```
class Power {
  int balance;

  void deposit(int amount) {
    balance = balance + amount;
  }

  boolean withdraw(int amount) {
    if (balance - amount > 0) {
      balance = balance - amount;
      return true;
    } else return false;
  }
}
```

logging class

```
class Logger {
  private PrintStream stream;

  Logger() {
    … create stream
  }

  void log(String message) {
    stream.println(message);
  }
}
```

logging the traditional way

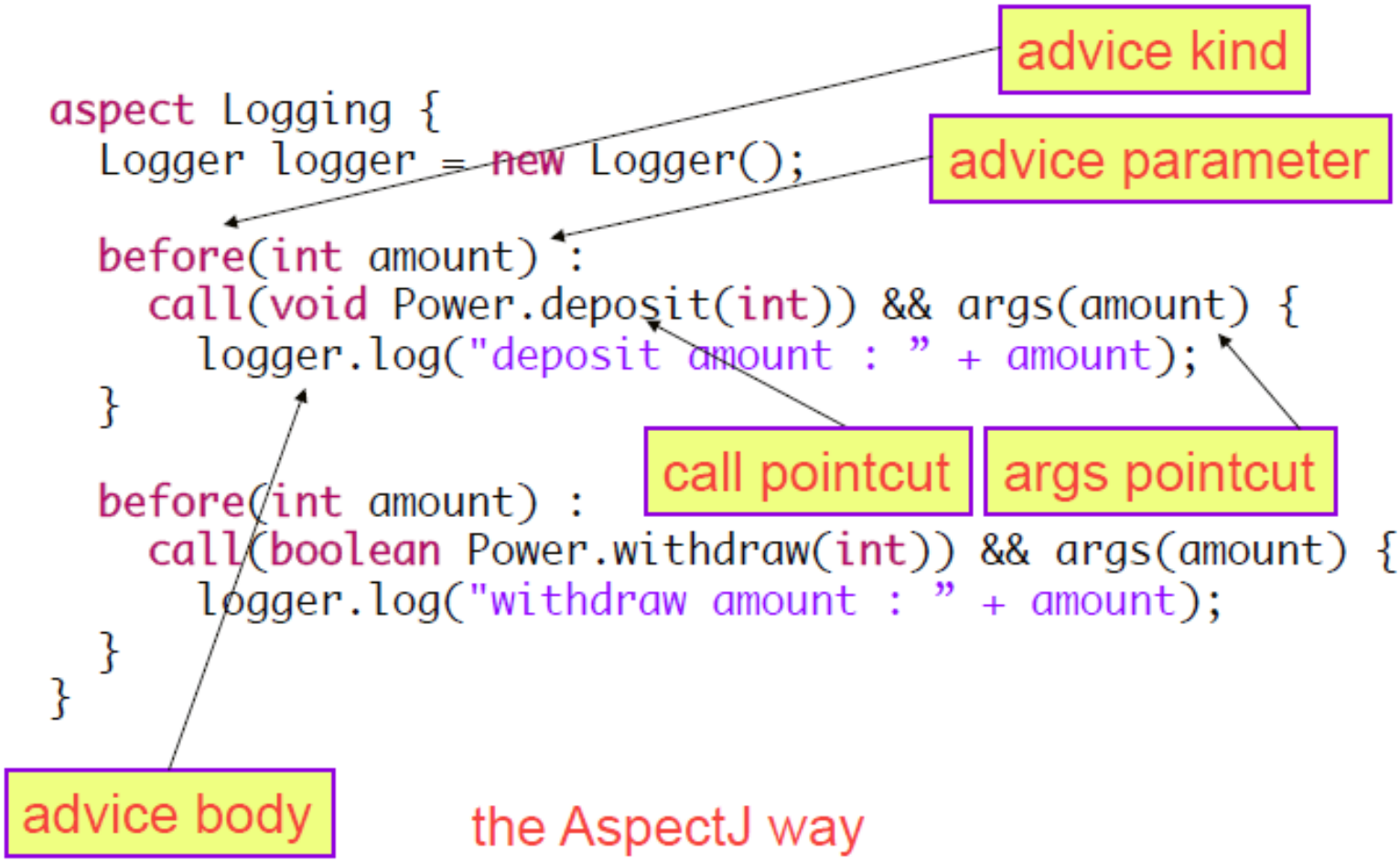


logging the AOP way

```
aspect Logging {
  Logger logger = new Logger();

  when deposit(amount) {
    logger.log("deposit amount : " + amount);
  }

  when withdraw(amount) {
    logger.log("withdraw amount : " + amount);
  }
}
```

that's not quite how it is written though

logging the AOP way



AspectJ History

- Java with Aspects
- Developed at Xerox PARC (Palo Alto RC)
- Launched in 1998
- PARC transferred AspectJ to an openly-developed eclipse.org project in December of 2002.

For more info: www.eclipse.org/aspectj

Terminology

- **Cross-cutting** – Identify areas of code where common functionality exists
- **Advice** – The code to be injected
- **Joinpoint** – Where one or more aspects can be applied
- **Pointcut** – A collection of joinpoints
- **Aspect** – General term for where advice and point-cuts are combined

Terminology

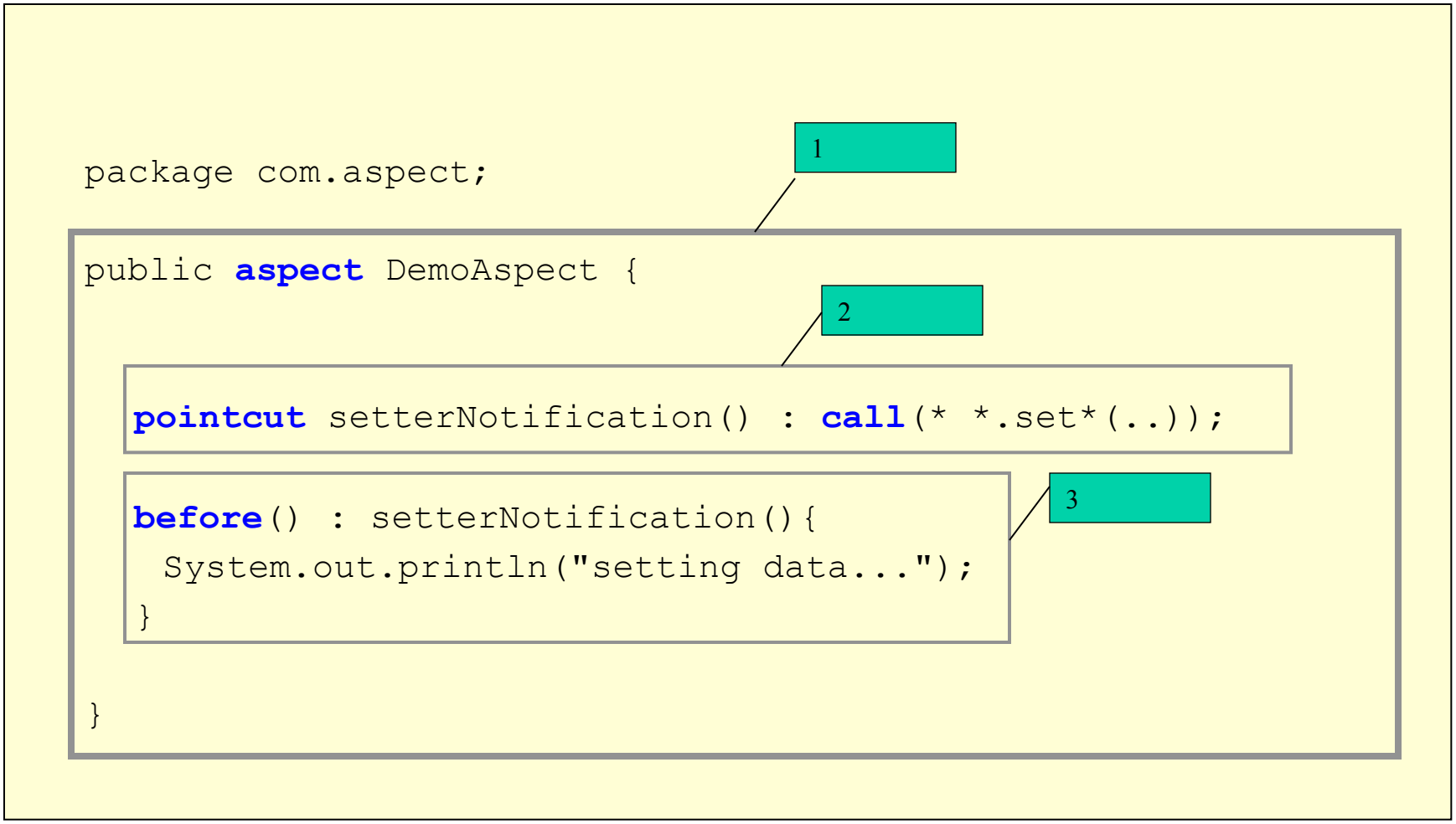
- **Weaving** – Integrating applications and aspects (e.g. AspectJ is an “aspect weaver”)
  - **Compile-time** – Can produce integrated source-code, but typically only produces woven byte-code.
  - **Run-time** – Aspects are applied “on the fly” (typically when classes are loaded)

basic mechanisms

- join points
  - points in a Java program
- three main additions to Java
  - pointcut
    - picks out join points and values at those points
      - primitive and user-defined pointcuts
  - advice
    - additional action to take at join points matching a pointcut
  - aspect
    - a modular unit of crosscutting behavior
      - normal Java declarations
      - pointcut definitions
      - advice

- inter-type declarations
  add fields, methods to classes

What does this aspect do?



Quick Look

```
package com.aspect;

public aspect DemoAspect {

   pointcut setterNotification() : call(* *.set*(..));

   before() : setterNotification(){
     System.out.println("setting data...");
   }

}
```

Aspect

Point-cut

Advice

Language: Join Points

- Well-defined points in the execution of a program:
  - Method call, Method execution
  - Constructor call, Constructor execution
  - Static initializer execution
  - Object pre-initialization, Object initialization
  - Field reference, Field set
  - Handler
  - etc

Joinpoint Types

- **call(***method expr***)**
  - when a method is called
- **execution(***method expr***)**
  - when a method is executed
- **handler(***exception expr***)**
  - when a catch block is executed

Defining Joinpoints

```
//call is the most common joinpoint type
call([access modifier] returnType package.ClassName.method(args));

//Examples
call(public * *.set*(..));
call(void *.set*(..));
call(* *.set*(int));
call(String com.foo.Customer.set*(..));
call(public void com..*.set*(int));
call(* *.set*(int, ..));
call(* *.set*(int, .., String));
```

Joinpoint Types

- **get(***field expr***)**
  - when a field is read
- **set(***field expr***)**
  - when a field is set
- **staticinitialization(***class expr***)**
  - when a static block of a class is exectued

Joinpoint Types

- **initialization(***constructor expr***)**
  - when a constructor is executed
- **preinitialization(***constructor expr***)**
  - when inherited constructors are executed
- **adviceexecution()**
  - when an advice block is executed

Language: Pointcuts

- A set of join point, plus, optionally, some of the values in the execution context of those join points.
- Can be composed using boolean operators || , &&
- Matched at runtime

Defining Pointcuts

- Defining a pointcut is similar to defining a method
- Define the pointcut name and the joinpoint type/ expression

```
public aspect SomeAspect{

    pointcut uberPointCut() : call(* *.*(..));
    pointcut setterTrace() : call(* *.set*(int, ..));
    pointcut exceptionTrace() : handler( java.io.Exception+ );

}
```

Defining Advice

- Advice, is even more similar to defining a method

```
pointcut somePointCut() :
     call(* *.*(..));

before() : somePointCut(){
     System.out.println("Injecting advice...");
}

after() : somePointCut(){
     System.out.println("Advice injected");
}
```

Defining Advice

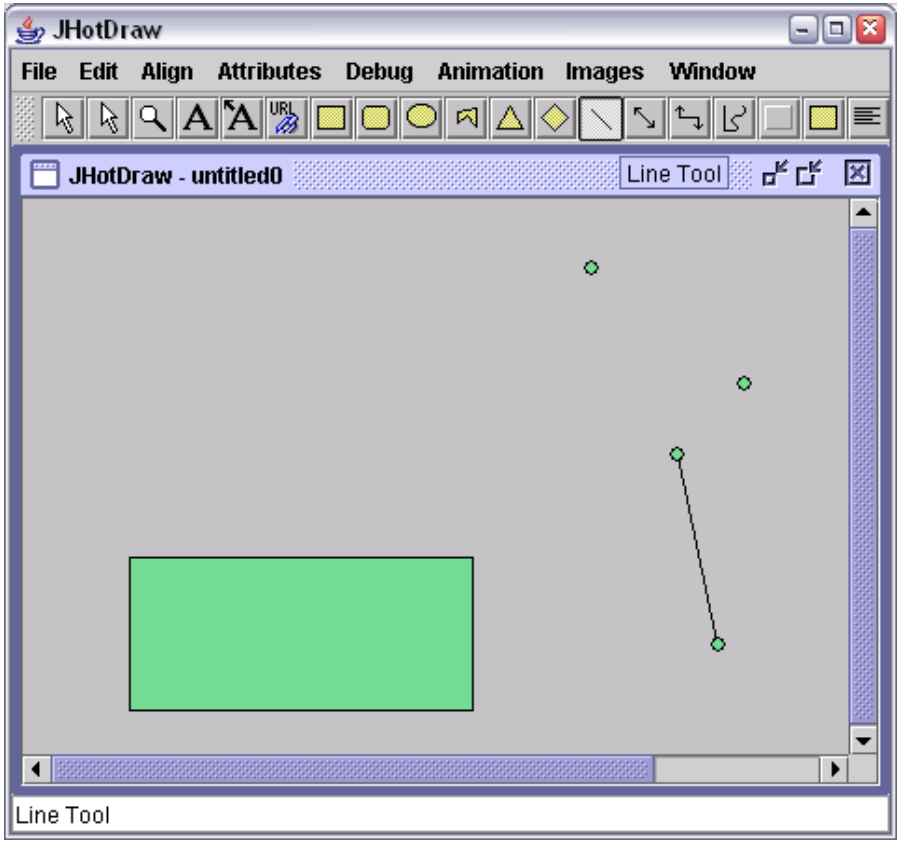
```
//NOTE:  You can streamline your definition of a
//       joinpoint and advice.

before() : call(* *.*(..)) {
     System.out.println("Injecting advice...");
}

//vs.

before() : somePointCut(){
     System.out.println("Injecting advice...");
}
```

Advice Types

- before()
- after() returning
- after() throwing
- after()
- around() \\ in place of (replaces the body)

AOP Exercise

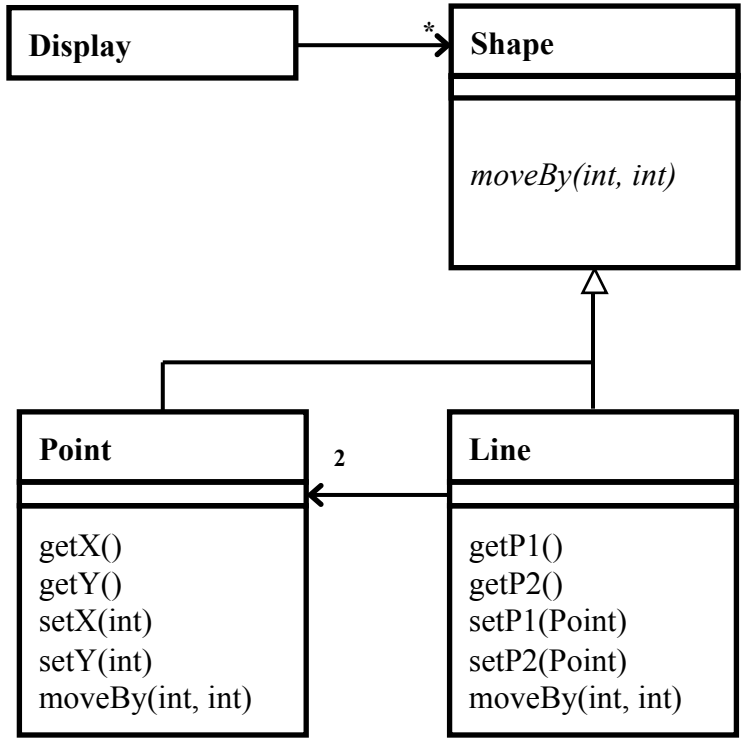
Consider developing…



a simple drawing application (JHotDraw)

Intuitively thinking of objects?

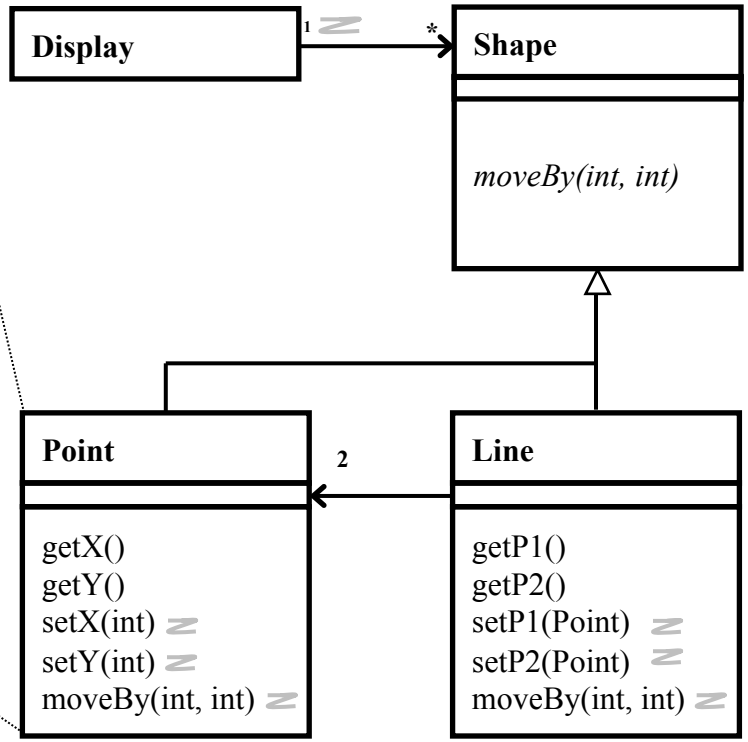
- Points, Lines…
- Drawing surfaces
- GUI Widgets
- …



"Update" is crosscutting: AOP Solution?

i.e. a simple Observer pattern

```
class Point extends Shape {
  private int x = 0, y = 0;

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

  void setX(int x) {
    this.x = x;
    display.update(this);
  }
  void setY(int y) {
    this.y = y;
    display.update(this);
  }
}
```

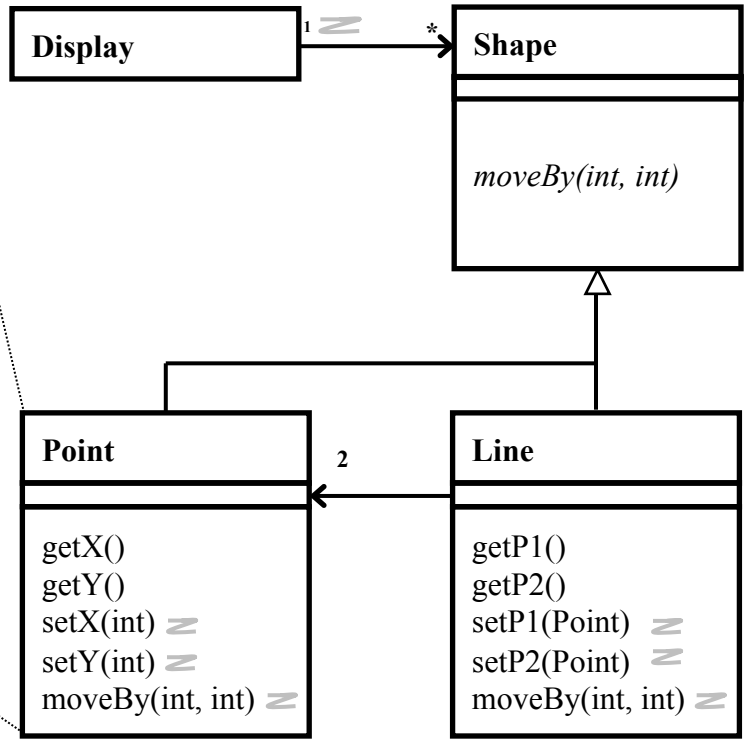


But some concerns "don't fit"

i.e. a simple Observer pattern

```
class Point extends Shape {
  private int x = 0, y = 0;

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

  void setX(int x) {
    this.x = x;
    display.update(this);
  }
  void setY(int y) {
    this.y = y;
    display.update(this);
  }
}
```

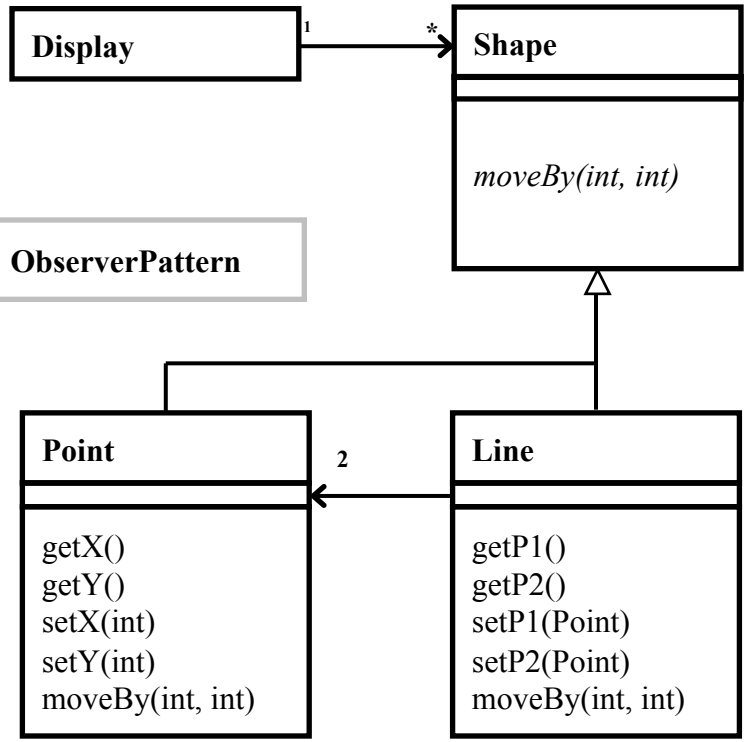
fair design modularity
but poor code modularity



With AOP they do fit

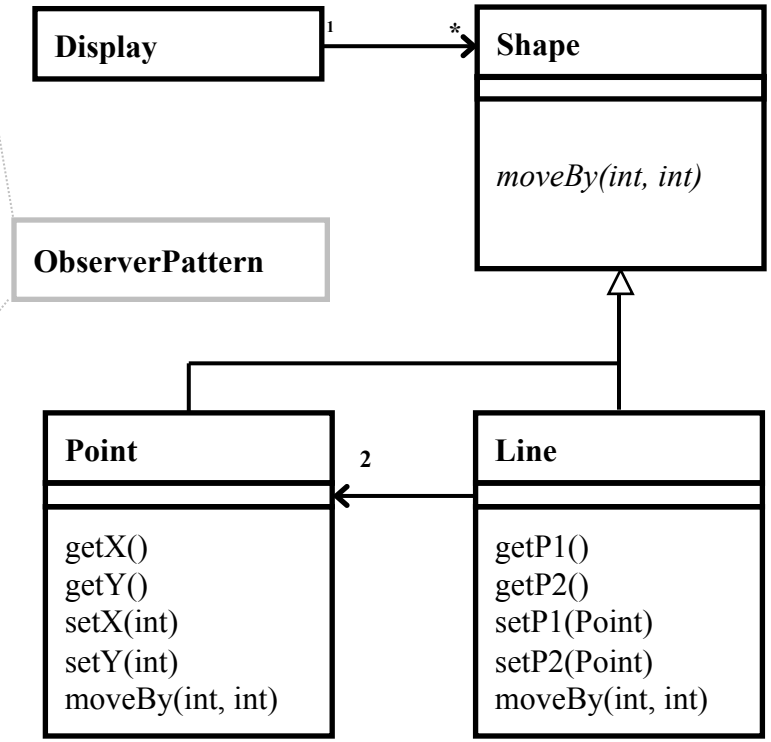
good design modularity
good code modularity

```
aspect ObserverPattern {

  private Display Shape.display;

  pointcut change():
    call(void figures.Point.setX(int))
    || call(void Point.setY(int))
    || call(void Line.setP1(Point))
    || call(void Line.setP2(Point))
    || call(void Shape.moveBy(int, int));

 after(Shape s) returning: change()
                           && target(s) {
   s.display.update();
  }
}
```



Code looks like the design

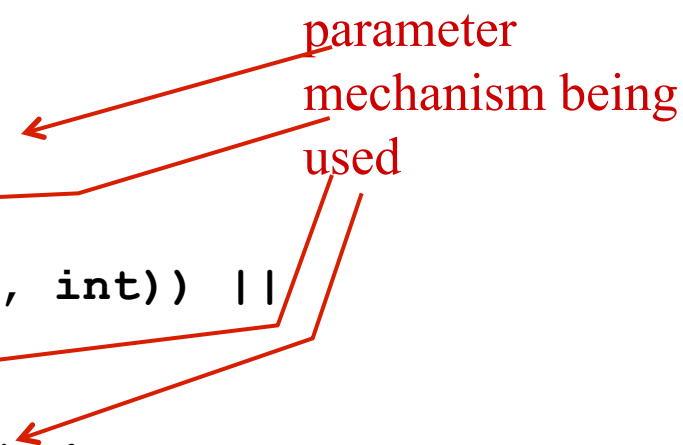
```
aspect ObserverPattern {

  private Display Shape.display;

  pointcut change():
    call(void Shape.moveBy(int, int))
    || call(void Shape+.set*(..));

  after(Shape s) returning: change()
                          && target(s) {
   s.display.update();
  }
}
```



Values at Join Points

- pointcut can explicitly expose certain values
- advice can use explicitly exposed values

```
private Display Shape.display;

pointcut change(Shape shape):
   this(shape) &&
   (execution(void Shape.moveBy(int, int)) ||
    execution(void Shape+.set*(*));

after(Shape s) returning: change(s) {
    s.display.update();
}
```



Aspect Can Own Object State

ObserverPattern v4

```
aspect ObserverPattern {

  private Display Shape.display;

  static void setDisplay(Shape s, Display d) {
    s.display = d;
  }

 pointcut cha
    this(shap
    (executio
     executio

  after(Shape shape): change(shape) {
    shape.display.update(s);
  }
}
```

private with respect to aspect

- display field
  - is in objects of type Shape
  - but belongs to ObserverPattern aspect
  - so ObserverPattern provide accessors

Without AspectJ

```
class Line extends Shape {
  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 extends Shape {
  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

ObserverPattern v1

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

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

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

class Point extends Shape {
  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

ObserverPattern v2

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

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

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

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

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

  void setX(int x) {
    this.x = x;
    Display.update();
  }
  void setY(int y) {
    this.y = y;
    Display.update();
  }
}
```

Without AspectJ

ObserverPattern v3

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

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

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

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

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

  void setX(int x) {
    this.x = x;
    Display.update(this);
  }
  void setY(int y) {
    this.y = y;
    Display.update(this);
  }
}
```

Without AspectJ

ObserverPattern v4

```
class Shape {
  private Display display;

  abstract void moveBy(int, int);
}

class Line extends Shape {
  private Point p1, p2;

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

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

class Point extends Shape {
  ...
}
```

"display updating" is not modular

- evolution is cumbersome
- changes are scattered
- have to track & change all callers
- it is harder to think about

With AspectJ

```
class Line extends Shape {
  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 extends Shape {
  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 ObserverPattern v1

```
class Line extends Shape {
  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 extends Shape {
  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 ObserverPattern {

  pointcut change():
    execution(void Line.setP1(Point)) ||
    execution(void Line.setP2(Point));

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

With AspectJ ObserverPattern v2

```
class Line extends Shape {
  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 extends Shape {
  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 ObserverPattern {

  pointcut change():
    execution(void Shape.moveBy(int, int) ||
    execution(void Line.setP1(Point))     ||
    execution(void Line.setP2(Point))     ||
    execution(void Point.setX(int))       ||
    execution(void Point.setY(int));

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

With AspectJ ObserverPattern v2.5

```
class Line extends Shape {
  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 extends Shape {
  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 ObserverPattern {

  pointcut change():
    execution(void Shape.moveBy(int, int) ||
    execution(void Shape+.set*(*));

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

With AspectJ

ObserverPattern v3

```
class Line extends Shape {
  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 extends Shape {
  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 ObserverPattern {

  pointcut change(Shape shape):
    this(shape) &&
    (execution(void Shape.moveBy(int, int) ||
     execution(void Shape+.set*(*)));

  after(Shape s) returning: change(s) {
    Display.update(s);
  }
}
```

With AspectJ

ObserverPattern v4

```
class Line extends Shape {
  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 extends Shape {
  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 ObserverPattern {

 private Display Shape.display;

  static void setDisplay(Shape s, Display d) {
    s.display = d;
  }

 pointcut change(Shape shape):
    this(shape) &&
    (execution(void Shape.moveBy(int, int)) ||
     execution(void Shape+.set*(*)));

  after(Shape shape): change(shape) {
    shape.display.update(s);
  }
}
```

ObserverPattern is modular

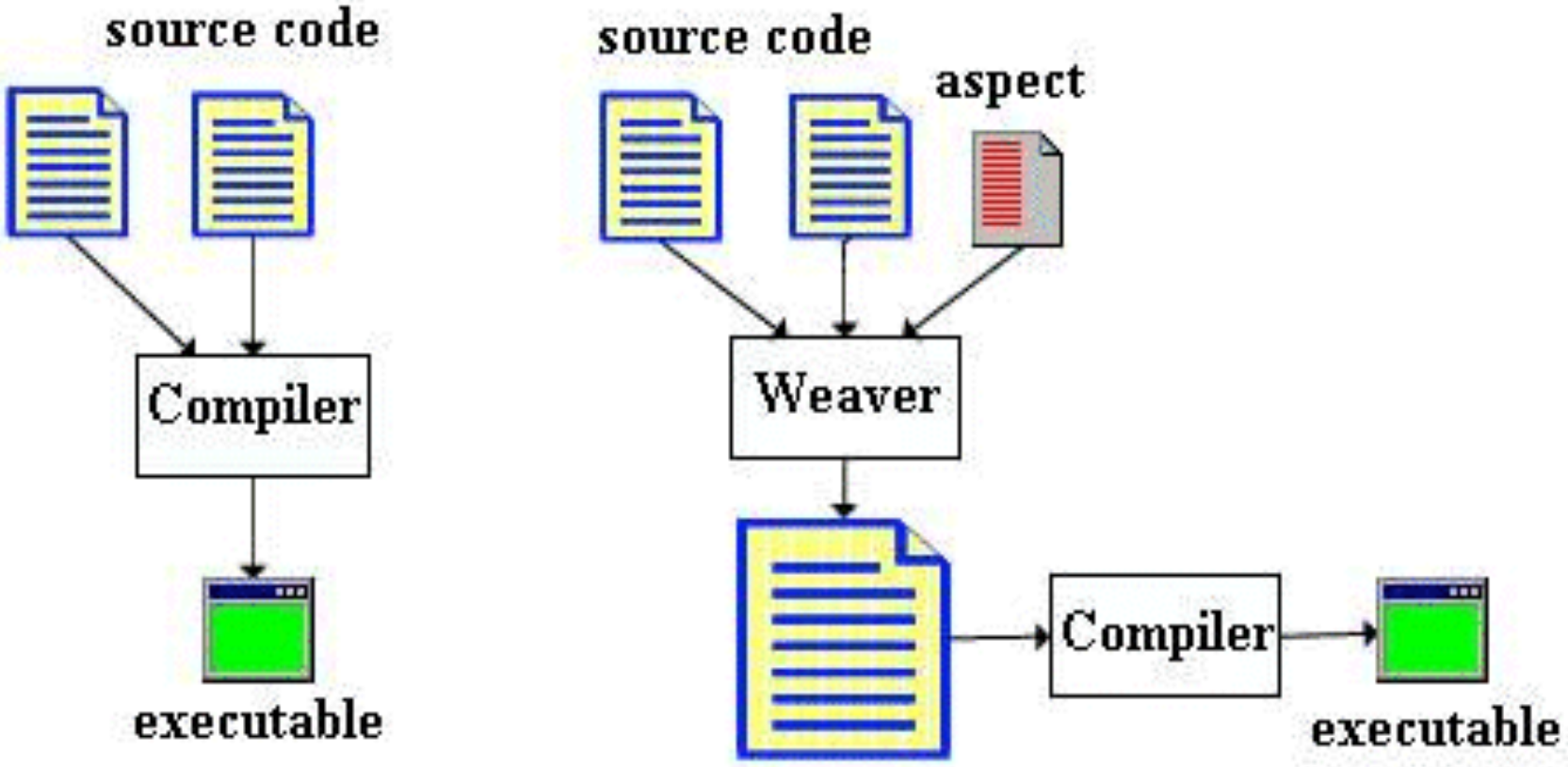
- all changes in single aspect
- evolution is modular
- it is easier to think about

IDE support

- AJDT (AspectJ Development Tool)
  - An Eclipse Project (Java)
  - highlighting, completion, wizards…
  - structure browser
    - immediate
    - outline
    - overview

- Many other languages have Aspect-supporting versions of them

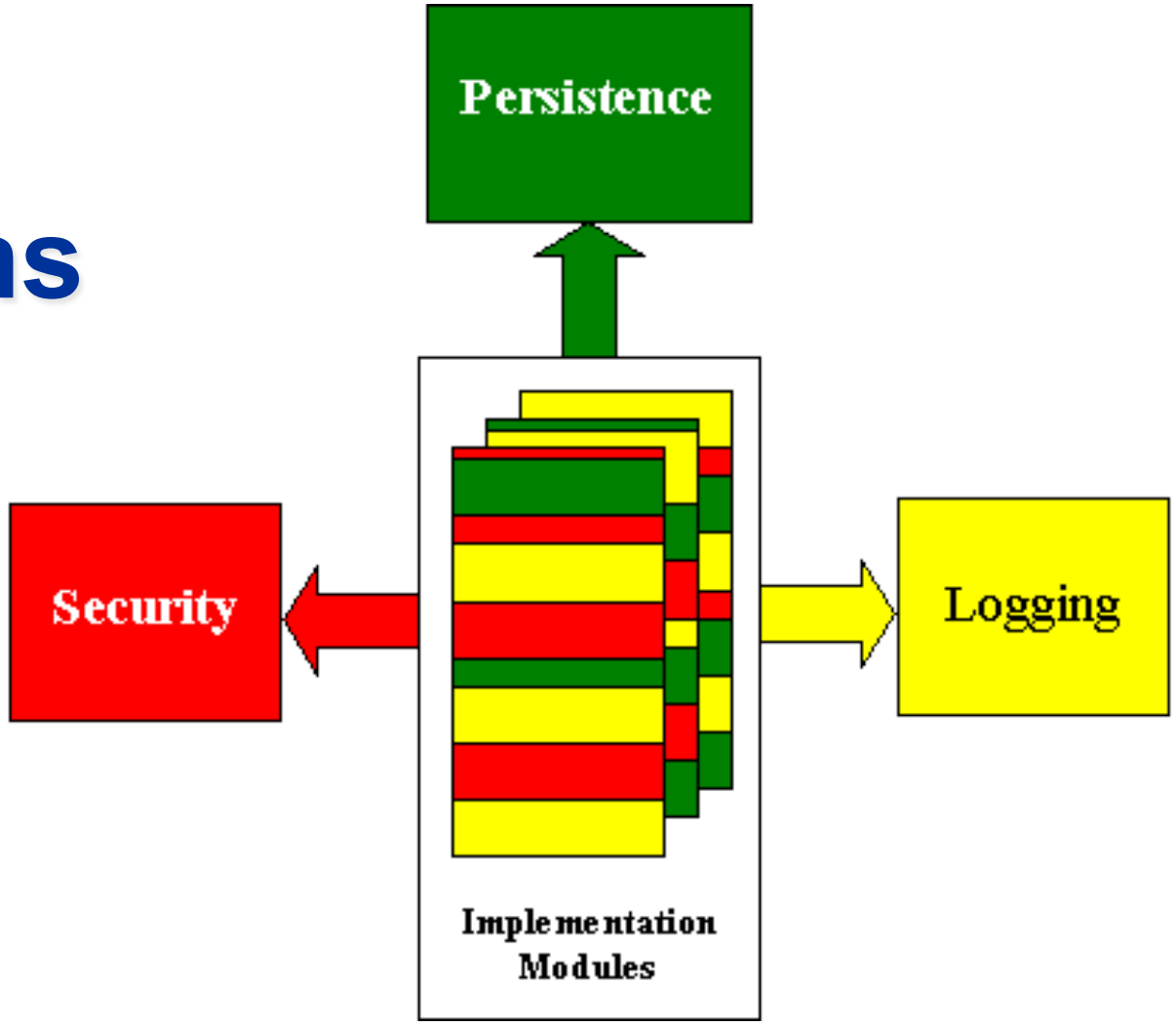
AOP Infrastructure



Weaving

- Aspect description languages cannot be processed by tradional compilers
- Therefore, aspects and components are woven into intermediate source code
- Weaving is no longer needed if there is an aspect-oriented compiler available
- This can be compared to preprocessing of C++ code to generate a C representation

Cross Cutting Concerns



Other Aspects

- Security
- Optimization
- Distribution
- Synchronization
- Persistence
- and of course, Logging
- and very many application-specific aspects
  - i.e. EnsureLiveness
  - J2EE transactions

AOP Benefits

- All benefits of OOP discussed before
- Separation of components and aspects
  - Better understanding of software because of high level of abstraction
  - Easier development and maintenance because tangling of code is prevented
  - Increased potential for reuse for both components as well as aspects

AOP Disadvantages

- Different mental model
- Harder to comprehend
- Harder to test

- Does this imply -> AOP is harder to maintain?

Summary

- AOP is a learned *intuitive way of thinking*
  - aspects, crosscutting structure
- Supporting mechanisms
  - join points, pointcuts, advice, inter-type declarations
- Allows us to
  - make code look like the design
  - improve design and code modularity
- A practical way to improve your
  - software designs, code, product, productivity