Aspect-Oriented Programming

Software Engineering
Andreas Zeller • Saarland University

Sources: http://www.eclipse.org/aspectj/

Principles

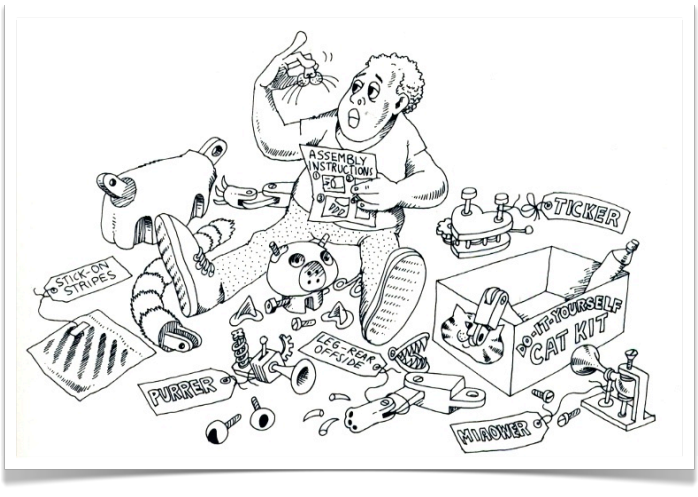
of object-oriented design

- Abstraction – Hide details
- Encapsulation – Keep changes local
- Modularity – Control information flow
  High cohesion • weak coupling • talk only to friends
- Hierarchy – Order abstractions
  Classes open for extensions, closed for changes • Subclasses that do not require more or deliver less • depend only on abstractions

Goal: Maintainability and Reusability

Let's recall the OO principles.

Modularity



We'll specifically focus on the issue of ~~modularity~~.

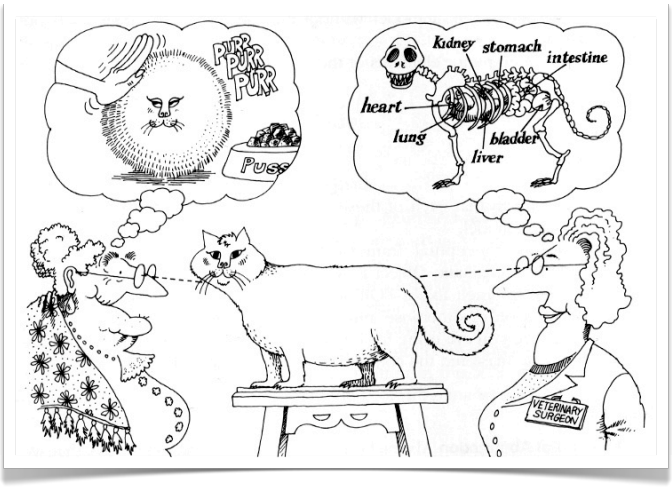
Principles of Modularity

- High cohesion – Modules should contain functions that logically belong together
- Weak coupling – Changes to modules should not affect other modules
- Law of Demeter – talk only to friends

High cohesion

- Modules should contain functions that logically belong together — *what does this mean?*
- Achieved by grouping functions that work on the same data
- "natural" grouping in object oriented design

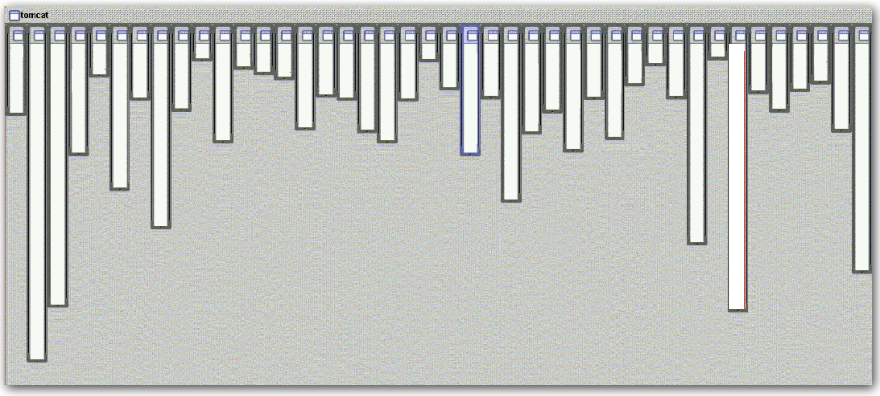
Perspectives



Sometimes, there are multiple ways to ~~abstract – and to decompose.~~

Modularity in Apache

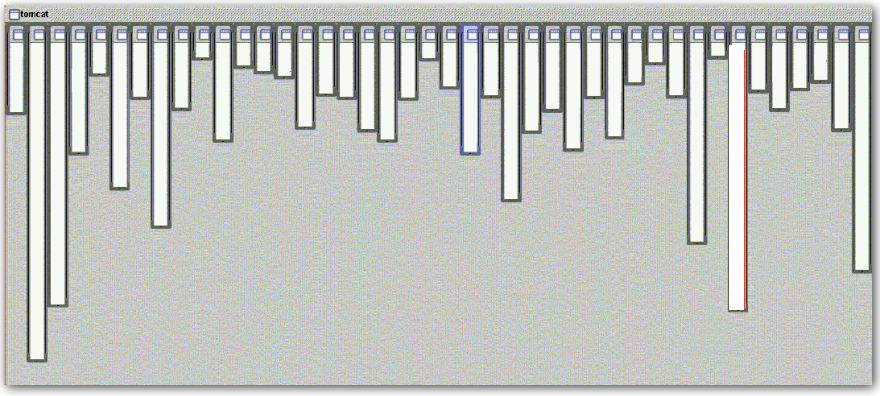
classes



Every class becomes a bar; the larger the source, the longer the bar.

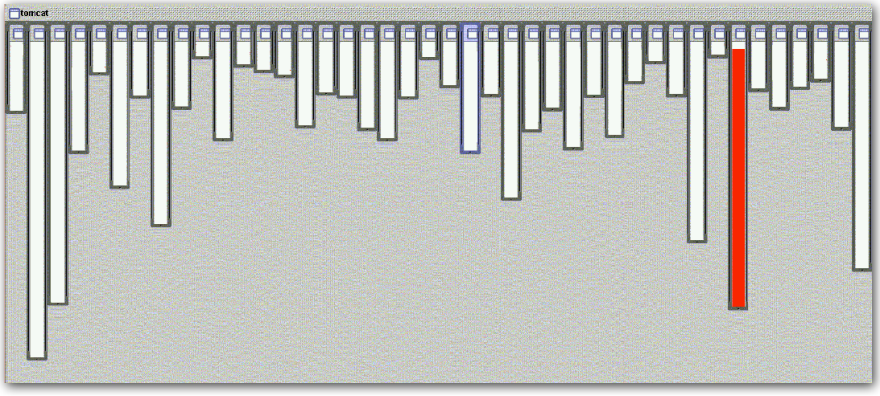
XML Parsing

classes



XML Parsing

classes



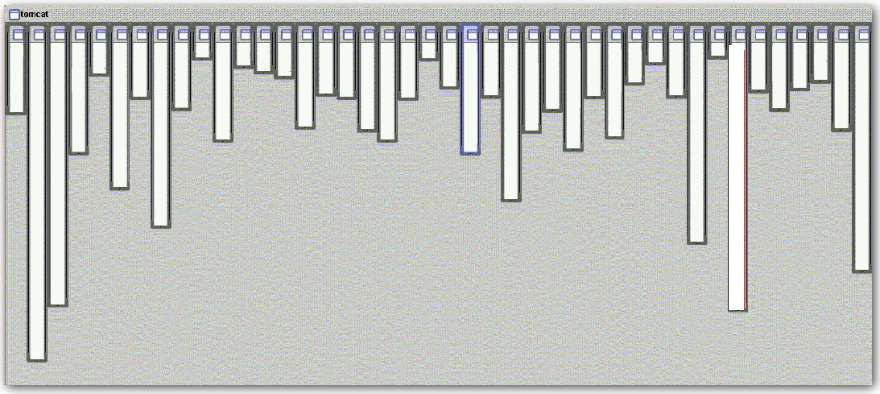
This is good modularity.

High cohesion

- Modules should contain functions that logically belong together —such as XML parsing
- Achieved by grouping functions that work on the same data
- "natural" grouping in object oriented design

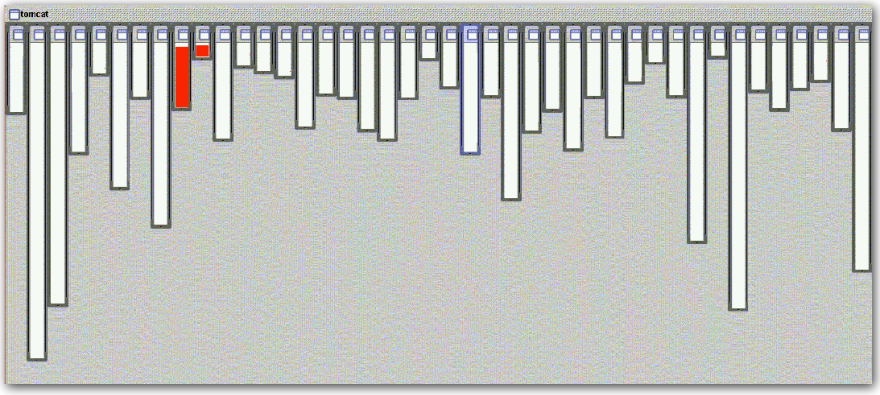
URL Matching

classes



URL Matching

classes



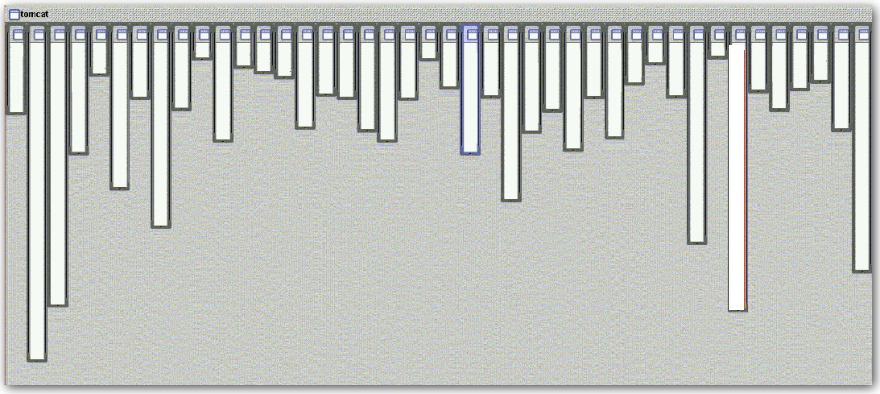
Spread across superclass and subclass

High cohesion

- Modules should contain functions that logically belong together — or UML parsing...
- Achieved by grouping functions that work on the same data
- "natural" grouping in object oriented design

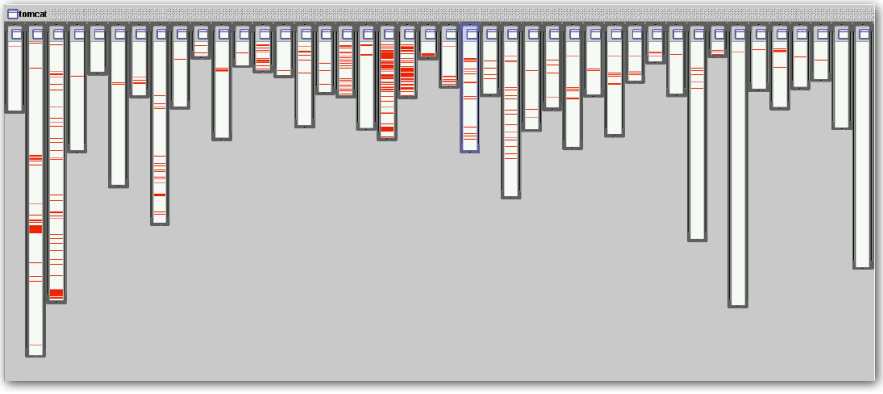
Logging

classes



Logging

classes



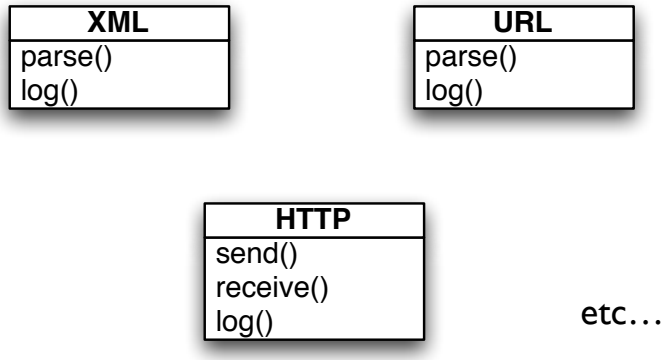
- Logging is a *cross-cutting concern*

The logging functionality is distributed all across the code – a *cross-cutting concern*

High cohesion

- Modules should contain functions that logically belong together — e.g., logging?
- Achieved by grouping functions that work on the same data
- "natural" grouping in object oriented design

Cohesion Alternative #1



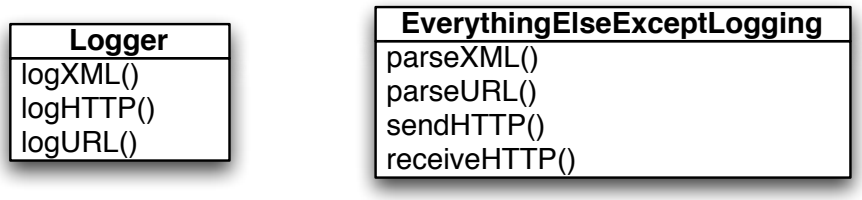
The state of things in Apache – and the standard way of doing things.

Cohesion Alternative #1

- log() is scattered across several classes
- Changes to logging format, etc. will all be non-local

The state of things in Apache – and the standard way of doing things.

Cohesion Alternative #2



Here, a Logger class groups everything related to logging – but how do we deal with the remaining classes?

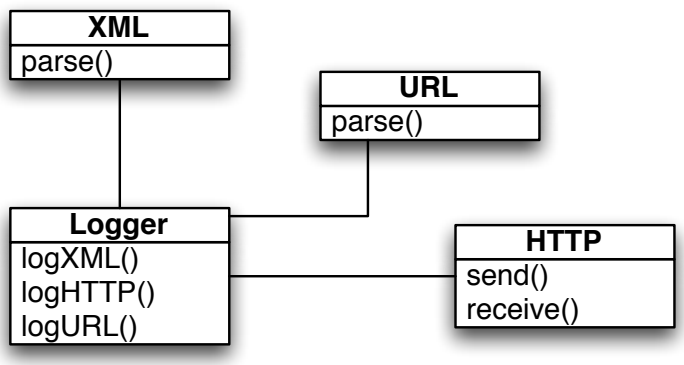
Cohesion Alternative #2

- Localizes everything related to logging
- Hard to decompose remaining functions into something meaningful

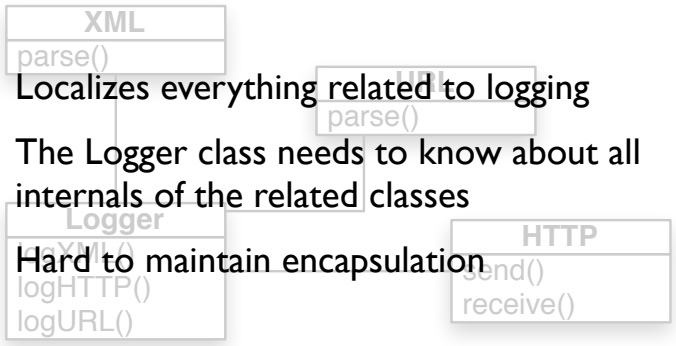
etc…

Here, a Logger class groups everything related to logging – but how do we deal with the remaining classes?

An Inconvenient Mixture

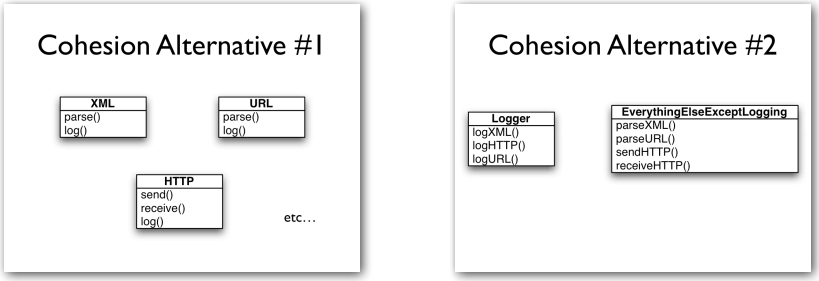


An Inconvenient Mixture



- Localizes everything related to logging
- The Logger class needs to know about all internals of the related classes
- Hard to maintain encapsulation

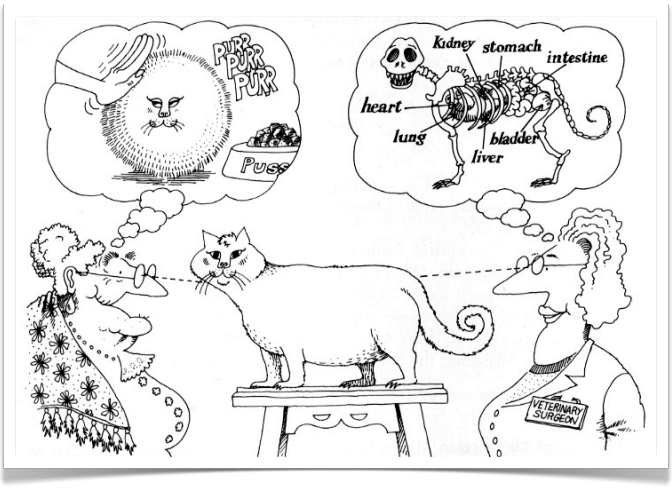
Dominant Decomposition



- We can only have *one* decomposition at a time
  (also called *tyranny of the dominant decomposition*)

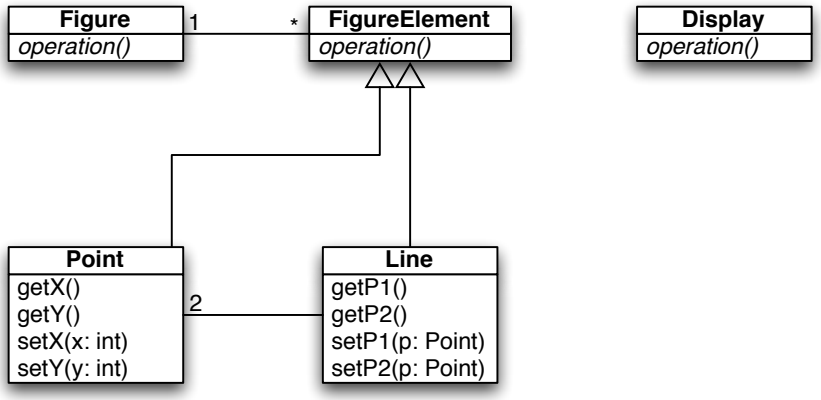
Tyranny of the Dominant Decomposition: the program can be modularized in only one way at a time, and the many kinds of concerns that do not align with that modularization end up scattered across many modules and tangled with one another.

~~Perspectives~~ Aspects



These perspectives are called “aspects” – and they are heavily interwoven.

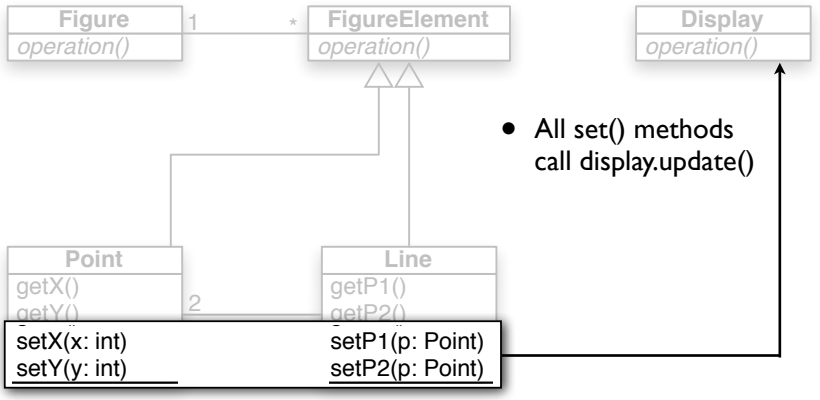
Figures



Here's another example – a simple figure object

from "An Introduction to AspectJ", http://www.eclipse.org/aspectj/

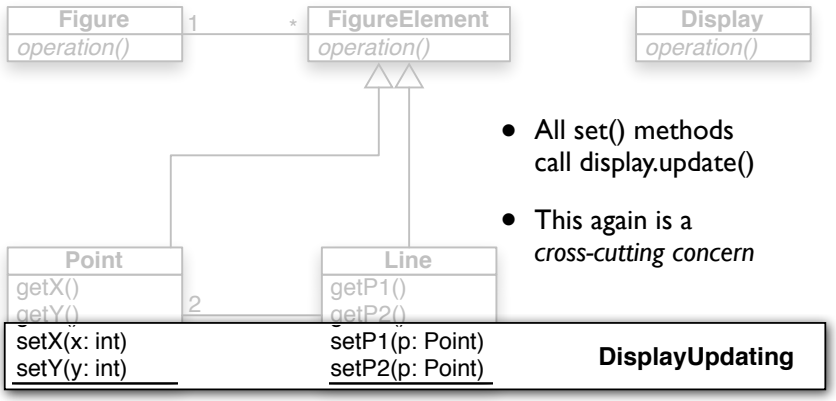
Figures



- All set() methods call display.update()

Here's another example – a simple figure object

Figures



- All set() methods call display.update()
- This again is a *cross-cutting concern*

Here's another example – a simple figure object

Aspects

- General idea: Create *syntactic structures* for cross-cutting concerns (*aspects*)
- Aspect-Oriented Programming introduces *aspects* into programs and programming languages

AspectJ

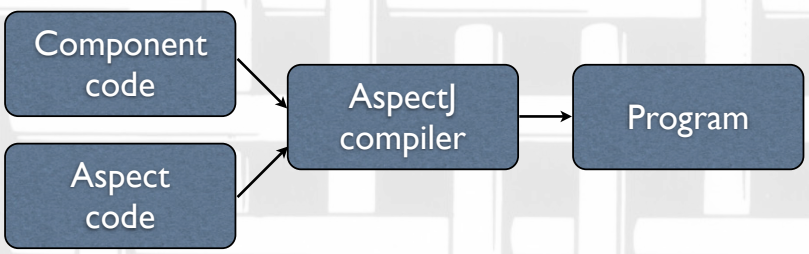
- Aspect-oriented extension to Java
- Developed by Gregor Kiczales et al. at XEROX PARC (~2001)

aspectj *crosscutting objects for better modularity*

See Website: http://www.eclipse.org/aspectj/

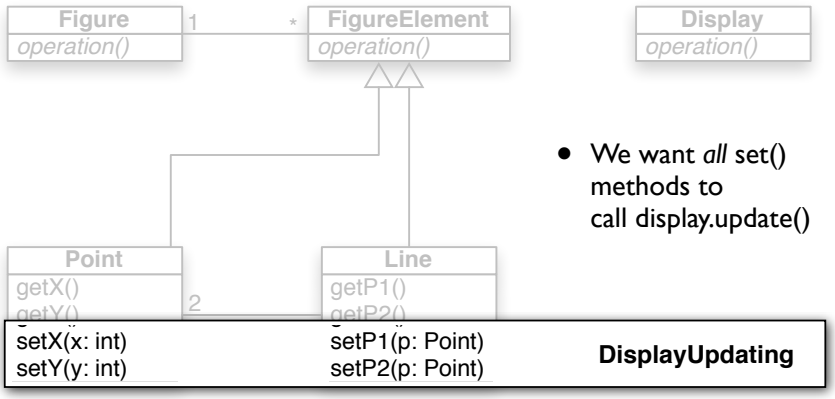
AspectJ

- General idea: Have *aspects* contain code that is executed at specific locations
- Aspects are *interwoven* with the remainder of the program



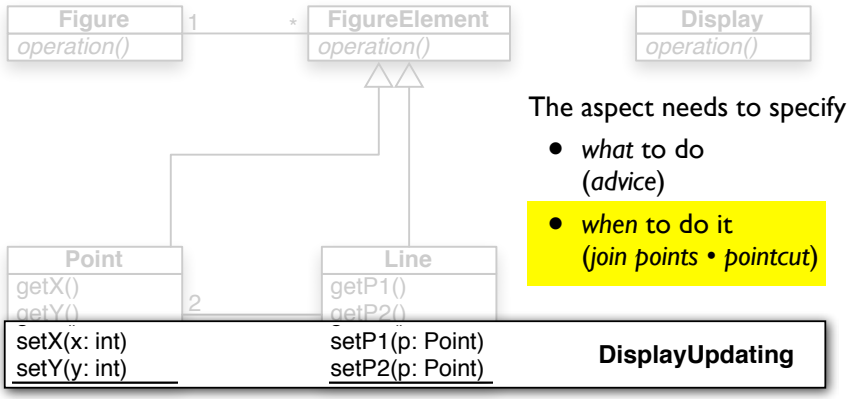
See Website: http://www.eclipse.org/aspectj/

An Aspect



- We want *all* set() methods to call display.update()

An Aspect



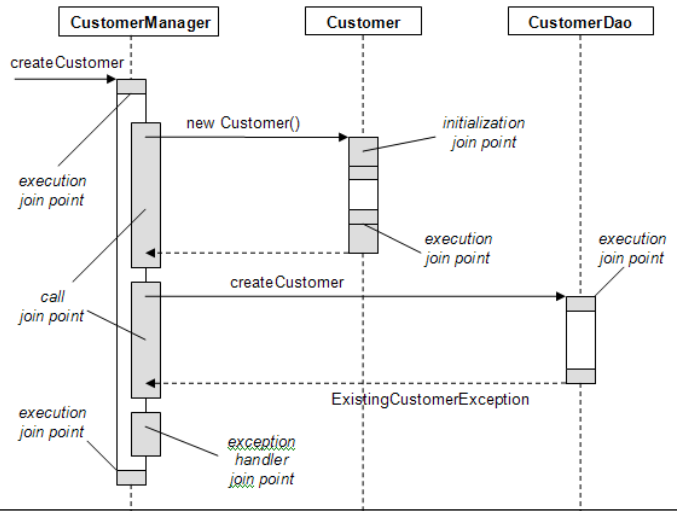
The aspect needs to specify

- *what* to do (*advice*)
- *when* to do it (*join points* • *pointcut*)

Join Points

- *Aspects* define types that cut across well-defined points in a program's execution
- These points are called *join points*
- The main joint point type is *method call*
  other types: method execution, object initialization, exceptions…

Join Point Types



Source: Alex Ruiz, "An Introduction to AspectJ", ObjectiveView vol. 9, p. 29

Pointcuts

- Pointcuts pick out certain join points in the program flow
- The pointcut
  `call(void Point.setX(int))`
  picks out each join point that is a call to
  `Point.setX(int)`

From "An Introduction to AspectJ", http://www.eclipse.org/aspectj/

Building Pointcuts

- A pointcut can be built out of other pointcuts with and, or, and not:
- `call(void Point.setX(int)) ||`
  `call(void Point.setY(int))`
  picks out each join point that is a call to
  `setX()` or `setY()`

Moving Elements

```
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));
```

In our example system, this pointcut captures all the join points when a FigureElement moves.

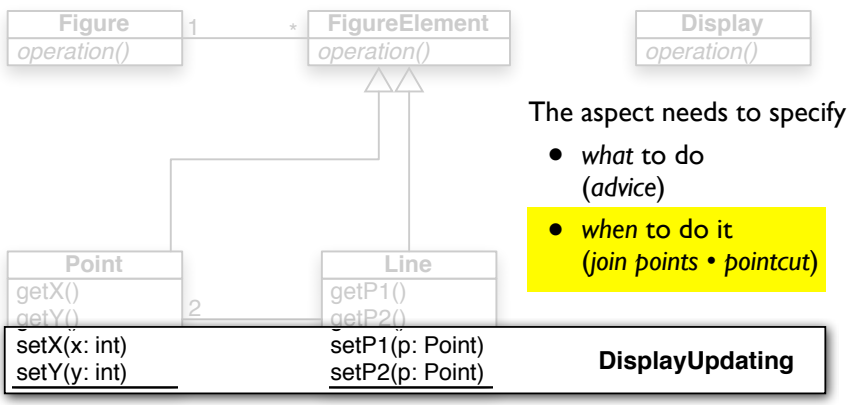
Naming Pointcuts

```
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));
```

- This defines all moments an object is moved
  which means that the display must be updated!

While this is a useful way to specify this crosscutting concern, it is a bit of a mouthful. So AspectJ allows programmers to define their own named pointcuts with the pointcut form.

An Aspect



The aspect needs to specify

- *what* to do
  (*advice*)
- *when* to do it
  (*join points* • *pointcut*)

Advice

- *Advices* bring together
  - a *point cut*
    (to pick out join points)
  - a *piece of code*
    (to be executed at each of these join points)

Before Advice

- runs as a join point is reached, *before* the program proceeds with the join point:

```
before(): move() {
    System.out.println("about to move");
}
```


After Advice

- runs *after* the program proceeds with that join point:

```
after() returning: move() {
    System.out.println("just moved");
}
```

- Three variants:
  `after returning` • `after throwing` • plain `after`

plain after runs after returning or throwing, like Java's finally

Aspects

- *Aspects* wrap up *pointcuts*, *advice*, and *inter-type* declarations in a modular unit of crosscutting implementation
- Defined very much like a class
- Can have methods, fields, and initializers in addition to the crosscutting members

A Logging Aspect

```
aspect Logging {

  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));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```

pointcut: when to call the advice

advice: code called at pointcuts

Logging Output

```
$ ajc Logging.aj MyProgram.java
$ java MyProgram
just moved
just moved
just moved
just moved
$
```

whenever an object moves

A Logging Aspect

```
aspect Logging {
  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));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```


An Updating Aspect

```
aspect Logging {
  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));

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


Fine Points

Patterns

```
aspect Logging {
  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));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```

Flow

```
aspect SetsInRotateCounting {   use for profiling
  int rotateCount = 0;
  int setCount = 0;             aspects are singletons

  before(): call(void Line.rotate(double)) {
    rotateCount++;
  }                                  function is active

  before(): call(void Point.set*(int))
   && cflow(call(void Line.rotate(double))) {
    setCount++;
  }
}
```

- counts all calls to set*() while rotate() is active

Context

```
aspect Logging {
  pointcut move(Line a_line, Point p):
    call(void Line.setP*(Point))
    && target(a_line)        object being called
    && args(p);              call arguments

  after(Line a_line, Point p)
    returning: move(a_line, p) {
    System.out.println(
      "Line " + a_line + " moved to " + p + ". ");
  }
}
```

Inter-type Declarations

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    …
}
```

adds new field to Point class

Patterns

```
aspect Logging {
  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));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```


Patterns

```
aspect Logging {
  pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.set*(int))              ||
    call(void Line.set*(Point));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```

matches setX(), setY()...

We can use wildcards like * to match a set of methods, types, or classes.

Error Logger

```
aspect PublicErrorLogging {
  Log log = new Log();

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

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

all calls to xerox functions (all types, all args)

- logs all exceptions being thrown from Xerox

".." matches args.

Universal Logger

```
aspect SimpleTracing {
  pointcut tracedCall():
    call (* Point.*(..));

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

all calls to Point functions (all types, all args)

- logs all calls into Point methods

thisjoinpoint allows to dynamically access information about the current location

Caching

```
aspect MoveTracking {
  // set to true as objects are moved
  private static boolean dirty = false;

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

  pointcut move(): // as seen

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

use as:
if (testAndClear())
  display.update()

After every move, dirty is set, indicating that the display needs updating.

Fine Points

Patterns

```
aspect Logging {
  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));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```

Flow

```
aspect SetsInRotateCounting {   use for profiling
  int rotateCount = 0;
  int setCount = 0;             aspects are singletons

  before(): call(void Line.rotate(double)) {
    rotateCount++;
  }                                  function is active

  before(): call(void Point.set*(int))
   && cflow(call(void Line.rotate(double))) {
    setCount++;
  }
```

- counts all calls to set*() while rotate() is active

Context

```
aspect Logging {
  pointcut move(Line a_line, Point p):
    call(void Line.setP*(Point))
    && target(a_line)      object being called
    && args(p);            call arguments

  after(Line a_line, Point p)
    returning: move(a_line, p) {
    System.out.println(
      "Line " + a_line + " moved to " + p + ". ");
  }
}
```

Inter-type Declarations

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    …
}
```

adds new field to Point class

Flow

```
aspect SetsInRotateCounting {
  int rotateCount = 0;
  int setCount = 0;

  before(): call(void Line.rotate(double)) {
    rotateCount++;
  }

  before(): call(void Point.set*(int))
   && cflow(call(void Line.rotate(double))) {
    setCount++;
  }
```

use for profiling

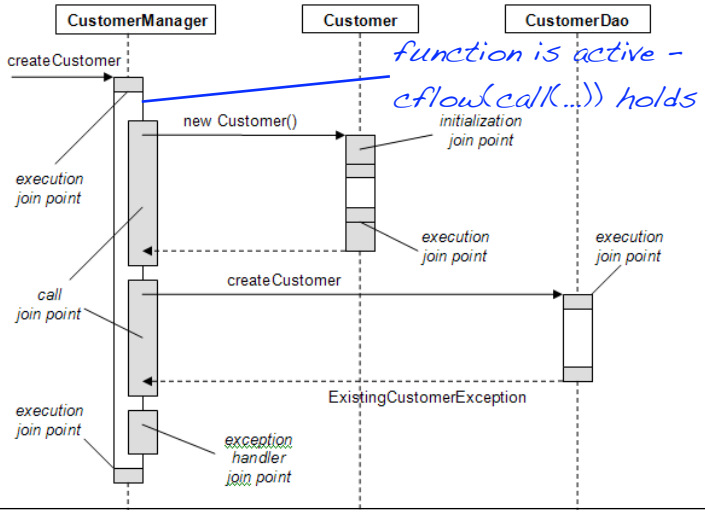
aspects are singletons

function is active

- counts all calls to set*() while rotate() is active

Here is a profiling example, which can be used to optimize run time.

Join Point Types



Source: Alex Ruiz, "An Introduction to AspectJ", ObjectiveView vol. 9, p. 29

Flow

```
aspect SetsInRotateCounting {
  int rotateCount = 0;
  int setCount = 0;

  before(): call(void Line.rotate(double)) {
    rotateCount++;
  }

  before(): call(void Point.set*(int))
   && cflow(call(void Line.rotate(double))) {
    setCount++;
  }
```

use for profiling

aspects are singletons

function is active

- counts all calls to set*() while rotate() is active

Fine Points

Patterns

```
aspect Logging {
  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));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```

Flow

```
aspect SetsInRotateCounting {   use for profiling
  int rotateCount = 0;
  int setCount = 0;             aspects are singletons

  before(): call(void Line.rotate(double)) {
    rotateCount++;
  }                                  function is active

  before(): call(void Point.set*(int))
   && cflow(call(void Line.rotate(double))) {
    setCount++;
  }
```

- counts all calls to set*() while rotate() is active

Context

```
aspect Logging {
  pointcut move(Line a_line, Point p):
    call(void Line.setP*(Point))
    && target(a_line)        object being called
    && args(p);              call arguments

  after(Line a_line, Point p)
    returning: move(a_line, p) {
    System.out.println(
      "Line " + a_line + " moved to " + p + ". ");
  }
}
```

Inter-type Declarations

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    …
}
```

adds new field to Point class

Context

```
aspect Logging {
  pointcut move(Line a_line, Point p):
    call(void Line.setP*(Point))
    && target(a_line)        object being called
    && args(p);              call arguments

  after(Line a_line, Point p)
    returning: move(a_line, p) {
    System.out.println(
      "Line " + a_line + " moved to " + p + ". ");
  }
}
```

target() is used to access the object whose method is being called; args() allows to access the arguments.

Checking Context

```
aspect CheckPointsRange {
  pointcut set_x(Point p, int x):
    call(void Point.setX(int))
    && args(x)
    && target(p);

  before(): set_x(p, new_x) {
    if (x < MIN_X || x > MAX_X)
      throw new IllegalArgumentException();
  }

  // same for set_y
}
```

use like assertions - turn on & off

A simple assert() replacement…

Contract Enforcement

```
aspect RegistrationProtection {
  pointcut register():
    call(void Registry.register(FigureElement));

  pointcut canRegister():
    withincode(static * FigureElement.make*(..));

  before(): register() && !canRegister() {
    throw new IllegalAccessException(
      "Illegal call " + thisJoinPoint);
  }
}
```

only make* can call register()

but more powerful, as control flow can also be part of the game

Contract Enforcement

```
aspect RegistrationProtection {
  pointcut register():
    call(void Registry.register(FigureElement));

  pointcut canRegister():
    withincode(static * FigureElement.make*(..));

  declare error: register() && !canRegister();

  }
}
```

checked statically

Some control flow properties can even be checked statically (i.e. at compile time).

Wrappers

```
aspect WrappedTracing {
  pointcut tracedCall(int x):
    call (void Point.set*(int)) && args(x);

  around(int x): tracedCall(x) {
    System.out.println("Entering");
    proceed(x);
    System.out.println("Leaving");
  }
}
```

call wrapped method

- wraps around all set* methods

Finally, context is also used to implement wrappers that are called instead of a method. With proceed(), we can access the original “wrapped” method.

Fine Points

Patterns

```
aspect Logging {
  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));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```

Flow

```
aspect SetsInRotateCounting {   use for profiling
  int rotateCount = 0;
  int setCount = 0;             aspects are singletons

  before(): call(void Line.rotate(double)) {
    rotateCount++;
  }                                  function is active

  before(): call(void Point.set*(int))
   && cflow(call(void Line.rotate(double))) {
    setCount++;
  }
```

- counts all calls to set*() while rotate() is active

Context

```
aspect Logging {
  pointcut move(Line a_line, Point p):
    call(void Line.setP*(Point))
    && target(a_line)        object being called
    && args(p);              call arguments

  after(Line a_line, Point p)
    returning: move(a_line, p) {
    System.out.println(
      "Line " + a_line + " moved to " + p + ". ");
  }
}
```

Inter-type Declarations

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    …
}
```

adds new field to Point class

Inter-type Declarations

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    …
}
```

adds new field to Point class

Inter-type declarations in AspectJ are declarations that cut across classes and their hierarchies. Unlike advice, which operates primarily dynamically, introduction operates statically, at compile-time.

Inter-type Declarations

```
aspect PointObserving {
  private Vector Point.observers = new Vector();

  public static void addObserver(Point p, Screen s)
  {
    p.observers.add(s);
  }

  public static void removeObserver(Point p, Screen s)
  {
    p.observers.remove(s);
  }
}
```

methods to access the new field

Cloneable Figures

```
aspect CloneableFigures {

    declare parents: (Point || Line || Square)
        implements Cloneable;

    public Object (Point || Line || Square).clone()
        throws CloneNotSupportedException {
        return super.clone();
    }
}
```

- introduces declaration and method for three classes

Inter-type declarations may declare members that cut across multiple classes, or change the inheritance relationship between classes.

Fine Points

Patterns

```
aspect Logging {
  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));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```

Flow

```
aspect SetsInRotateCounting {   use for profiling
  int rotateCount = 0;
  int setCount = 0;             aspects are singletons

  before(): call(void Line.rotate(double)) {
    rotateCount++;
  }                                  function is active

  before(): call(void Point.set*(int))
   && cflow(call(void Line.rotate(double))) {
    setCount++;
  }
```

- counts all calls to set*() while rotate() is active

Context

```
aspect Logging {
  pointcut move(Line a_line, Point p):
    call(void Line.setP*(Point))
    && target(a_line)        object being called
    && args(p);              call arguments

  after(Line a_line, Point p)
    returning: move(a_line, p) {
    System.out.println(
      "Line " + a_line + " moved to " + p + ". ");
  }
}
```

Inter-type Declarations

```
aspect PointObserving {
    private Vector Point.observers = new Vector()
    …
}
```

adds new field to Point class

AspectJ Critized

- Non-local control flow hard to understand
- Aspects may interfere with each other
- Few guarantees, as aspects can change anything

A Coordinate System

```
10 input x
20 print “result is: “
30 x = x * x
40 print x
50 end
```

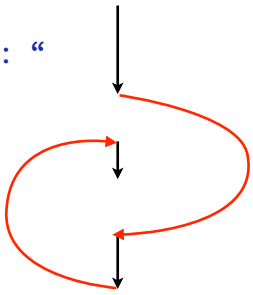
control flow

- Control through the program roughly follows the program text (top-bottom, left-right).

Goto Statement

```
10 input x
20 print “result is: “
30 goto 60
40 print x
50 end

60 x = x * x
70 goto 40
```



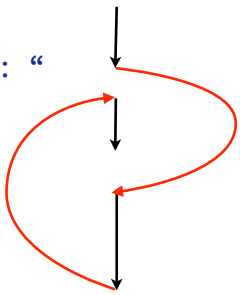
- The *goto* statement destroys the programmer’s coordinate system as induced by text flow

“goto statement considered harmful” – Dijkstra

Come from Statement

```
10 input x
20 print “result is: “
30 print x
40 end

50 come from 20
60 x = x * x
70 return
```



- Originally meant as a joke to illustrate the importance of coordinate systems

The INTERCAL language implements a “come from” statement

Come from with Aspects

```
main() {
  input x
  print (result(x))
}

int result(int x) { return x }

around(int x): call(result(int)) && args(x)
{
    int temp = proceed(x)
    return temp * temp
}
```

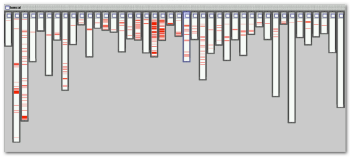

AspectJ

AspectJ Critized

- Undisputed: Helpful tool for development (e.g., profiling, logging, contracts…)

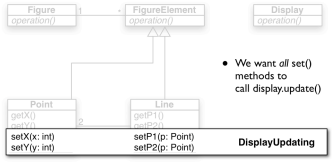
Logging

classes



- Logging is a *cross-cutting concern*

An Aspect



- We want *all* set() methods to call display.update()

Summary

Fine Points

Patterns

```
aspect Logging {
  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));

  after() returning: move() {
    System.out.println("just moved");
  }
}
```

Flow

```
aspect SetsInRotateCounting {  // acts like javaflow
  int rotateCount = 0;         // supports one singlethread

  before(): call(void Line.rotate(double)) {
    ++rotateCount;             // dataflow in action
  }
  before(): call(void Point.set*(int)) &&
    if(rotateCount)(void Line.rotate(double)) {
    setCount++;
  }
```

- counts all calls to set*() while rotate() is active

Context

```
aspect Logging {
  pointcut moveAllLine(Line a_line, Point p):
    call(void Line.setP1(Point)) &&
    target(a_line) &&            // context during called
    args(p);                     // call arguments

  after(Line a_line, Point p)
    returning: moveAllLine(a_line, p) {
    System.out.println(
      "Line " + a_line + " moved to " + p + ".");
  }
}
```

Inter-type Declarations

```
aspect PointObserving {
  private Vector Point.observers = new Vector();
  //                 adds new field to
  //                 Point class
}
```

AspectJ