Vrije Universiteit Brussel

AspectLab

Training Event on AspectJ

February, 2007

Vrije Universiteit Brussel

Many of the slides in this presentation are adapted from material currently and previously available on http://www.aosd.net/

Vrije Universiteit Brussel

What is AOSD?

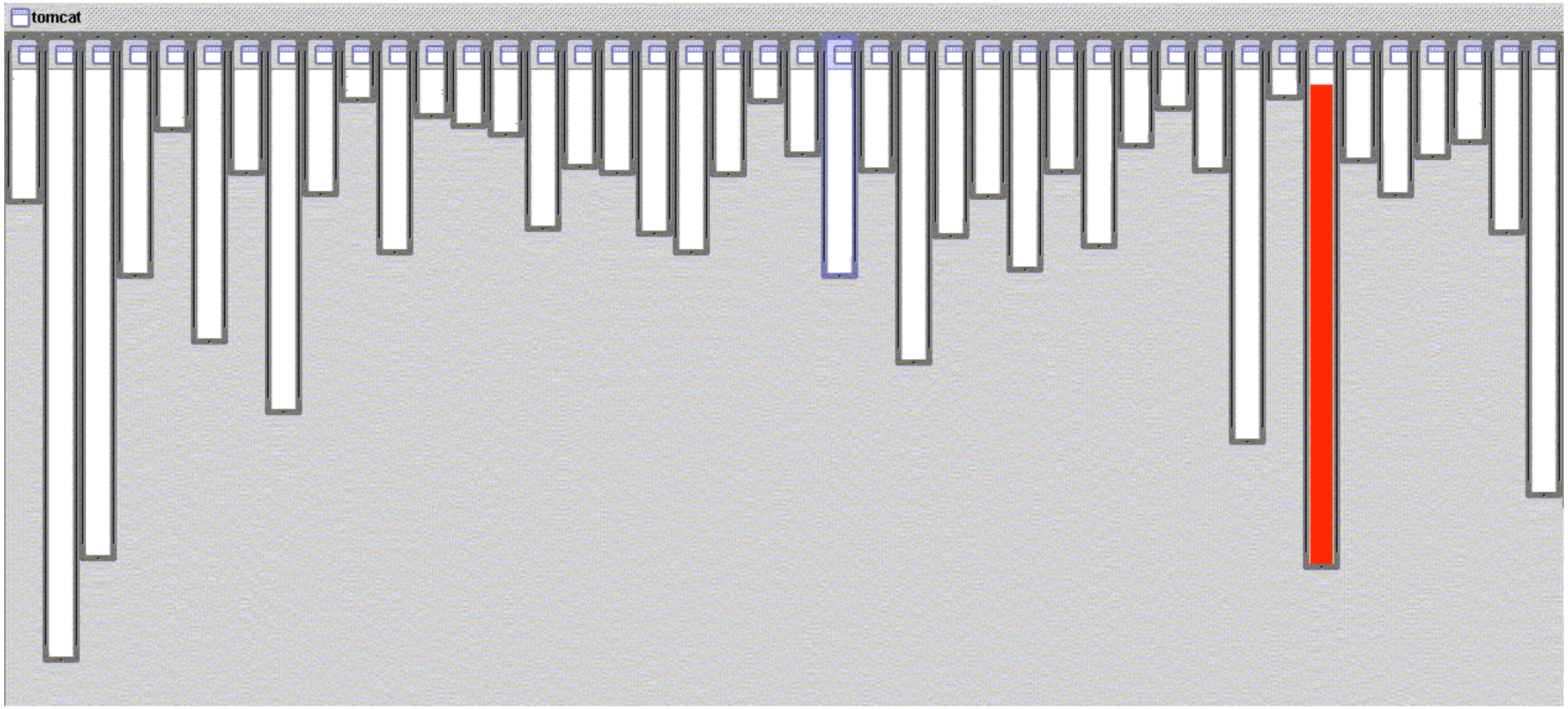
- A software development paradigm that advocates better separation of concerns

Concern: "*Something the developer needs to care about*" (e.g. functionality, QoS requirement, software process requirement..)

Separation of concerns: handle each concern separately

- An AOP language allows the modularisation of crosscutting concerns

XML parsing in org.apache.tomcat

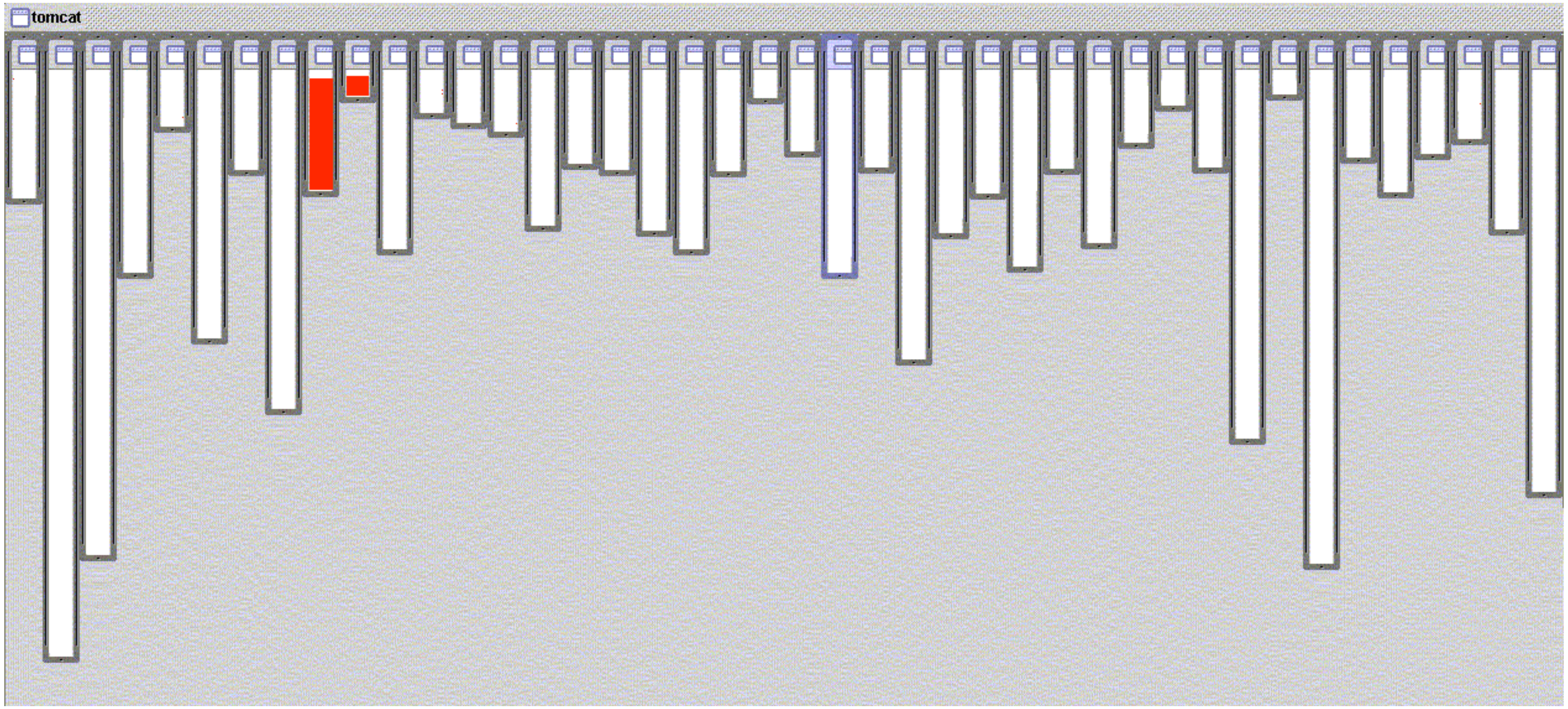


[Picture taken from the aspectj.org website]

Good modularity:

handled by code in one class

URL pattern matching in org.apache.tomcat

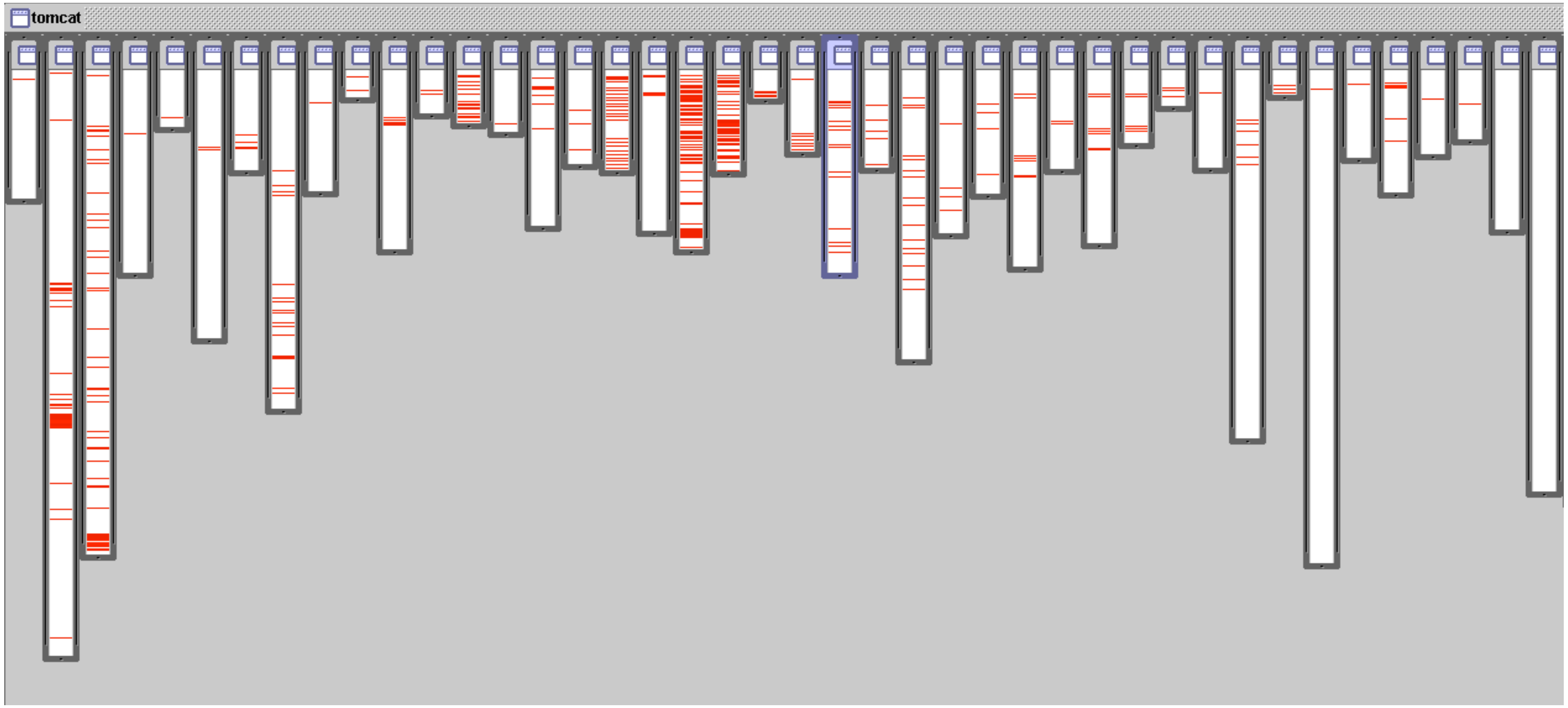


[Picture taken from the aspectj.org website]

Good modularity:

- **handled by code in two classes related by inheritance**

Logging in org.apache.tomcat



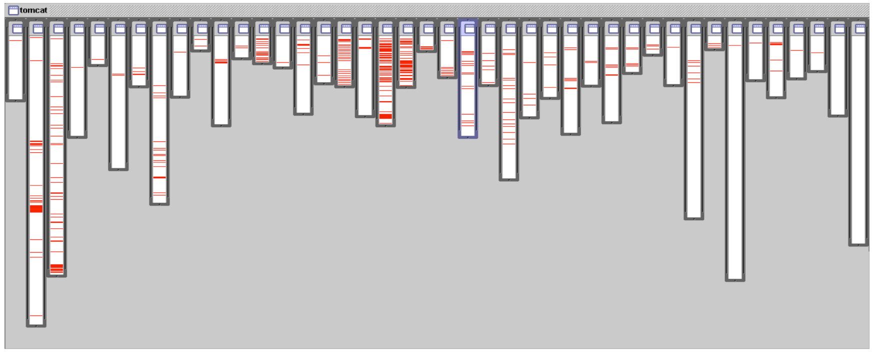
[Picture taken from the aspectj.org website]

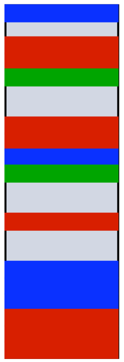
BAD modularity:

handled by code that is scattered over almost all classes

Scattering & Tangling

- code scattering – code for one concern is spread over many modules
- code tangling – code in one module addresses multiple concerns
- scattering and tangling tend to appear together; they describe different facets of the same problem





- redundant code, same or similar fragment of code in many places
- difficult to reason about
- difficult to change
  - have to find all the code involved
  - and be sure to change it consistently

The AOSD idea

- crosscutting is inherent in complex systems

  "tyranny of the dominant decomposition"

- crosscutting concerns
  - have a clear purpose ***What***
  - have some regular interaction points ***Where/When***
- AOSD proposes to capture crosscutting concerns explicitly...
  - in a modular way
  - not only in programming languages but in all stages of software development
  - and with appropriate tool support

AspectJ

- First production-quality AOP-technology
- Allows specifying crosscutting concerns as separate entities: **Aspects**
- Introduces:
  - **Join point**: some point in the execution of an application
  - **Pointcut**: a set of logically related join points
  - **Advice**: some behavior that should become active whenever a join point is encountered
  - **Weaving**: a technology for bringing aspects and base code together

Aspect: a special kind of unit

Aspect

Aspect applicability code

Pointcut

Aspect functionality code

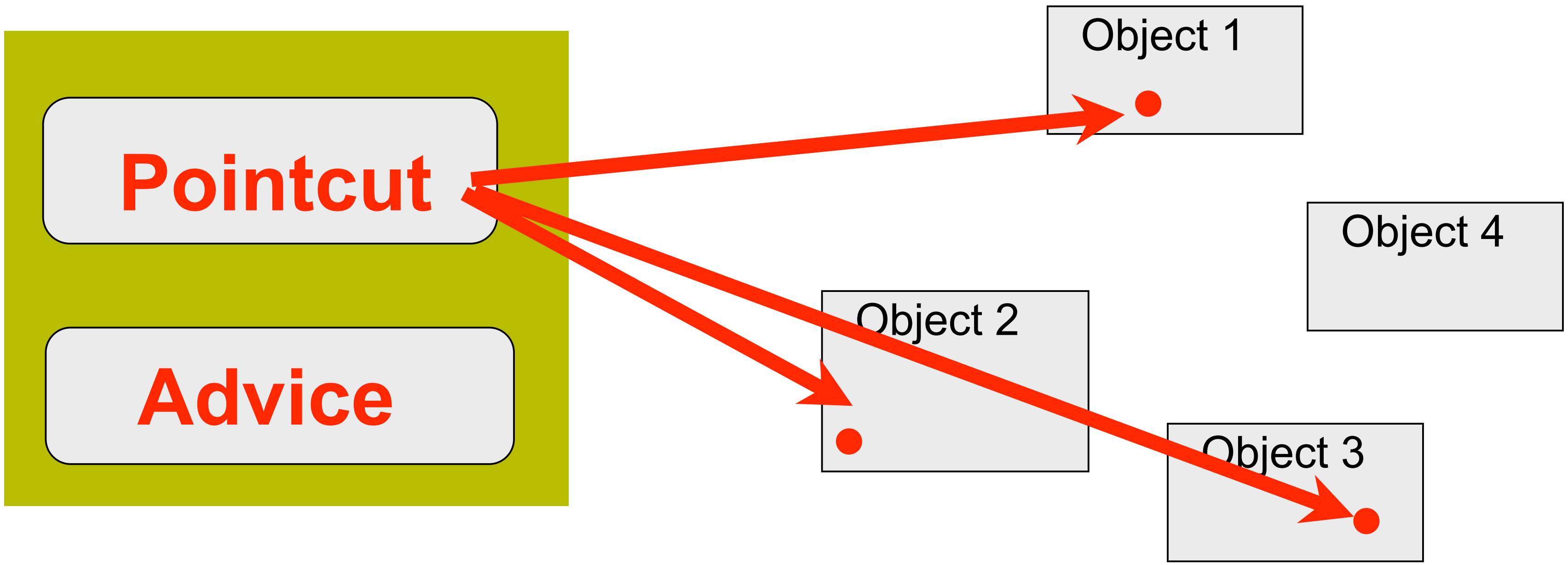
Advice

Where / when

What

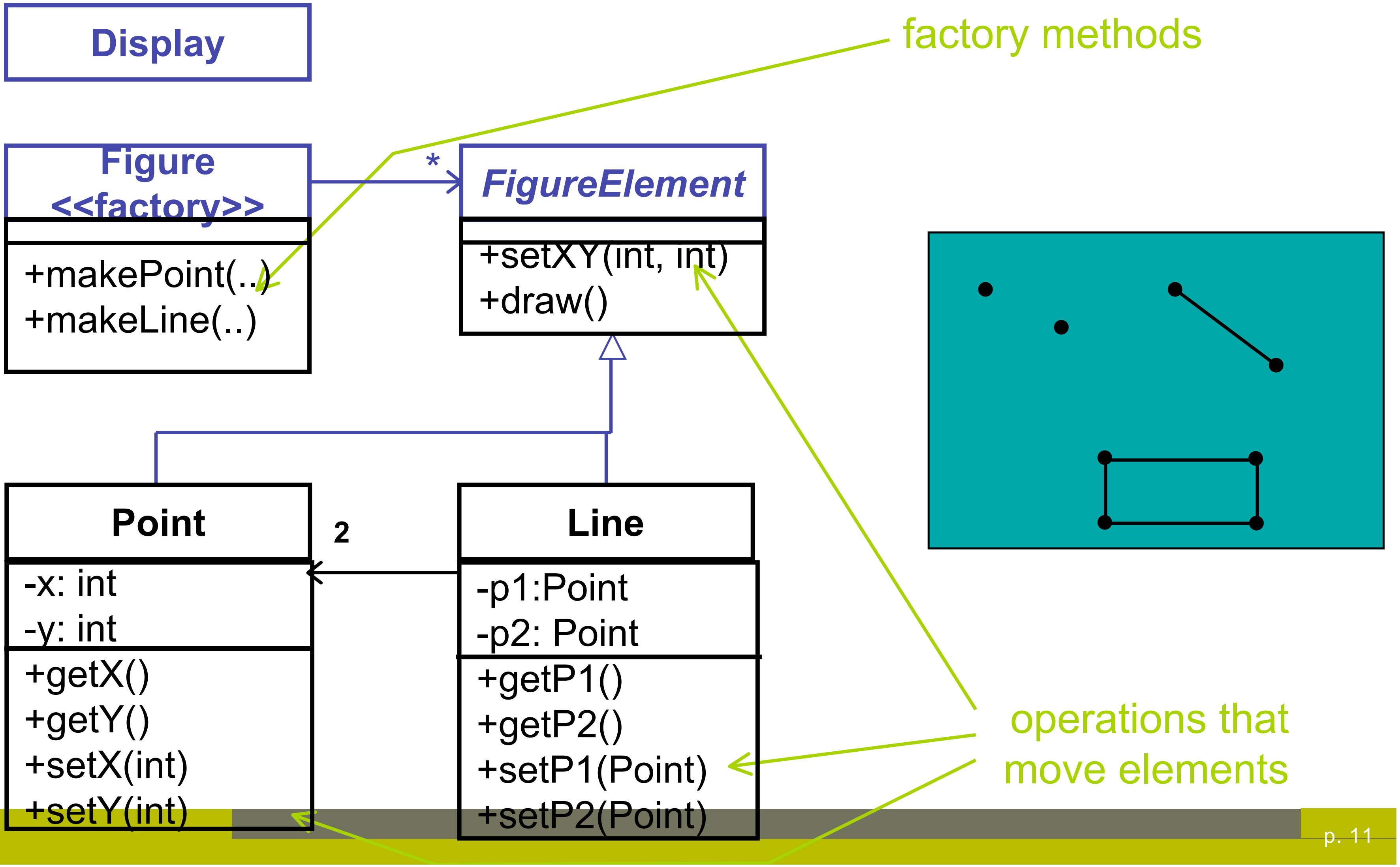
Vrije Universiteit Brussel

Implicit Invocation



joinpoint: ●

a simple figure editor

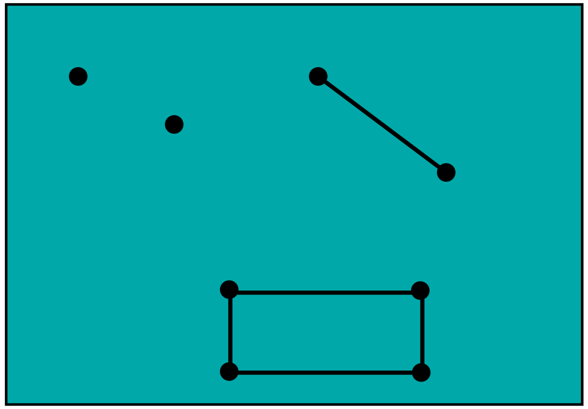


Vrije Universiteit Brussel

a simple figure editor

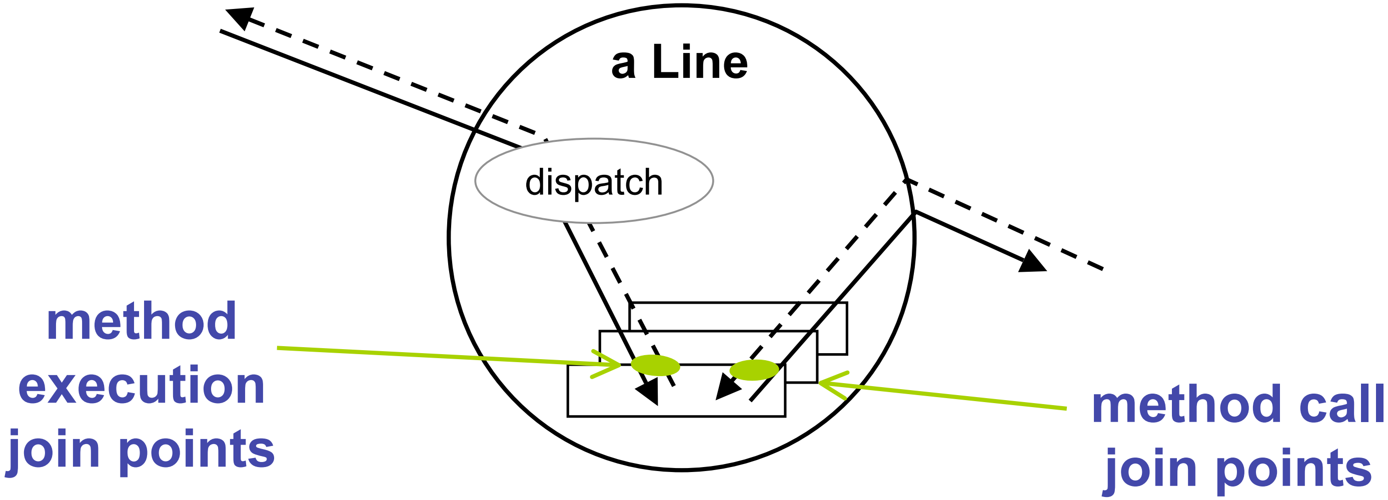
```
class Line implements FigureElement{
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) { this.p1 = p1; }
  void setP2(Point p2) { this.p2 = p2; }
  void setXY(int x, int y) {…}
}

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; }
  void setXY(int x, int y){…}
}
```



join point

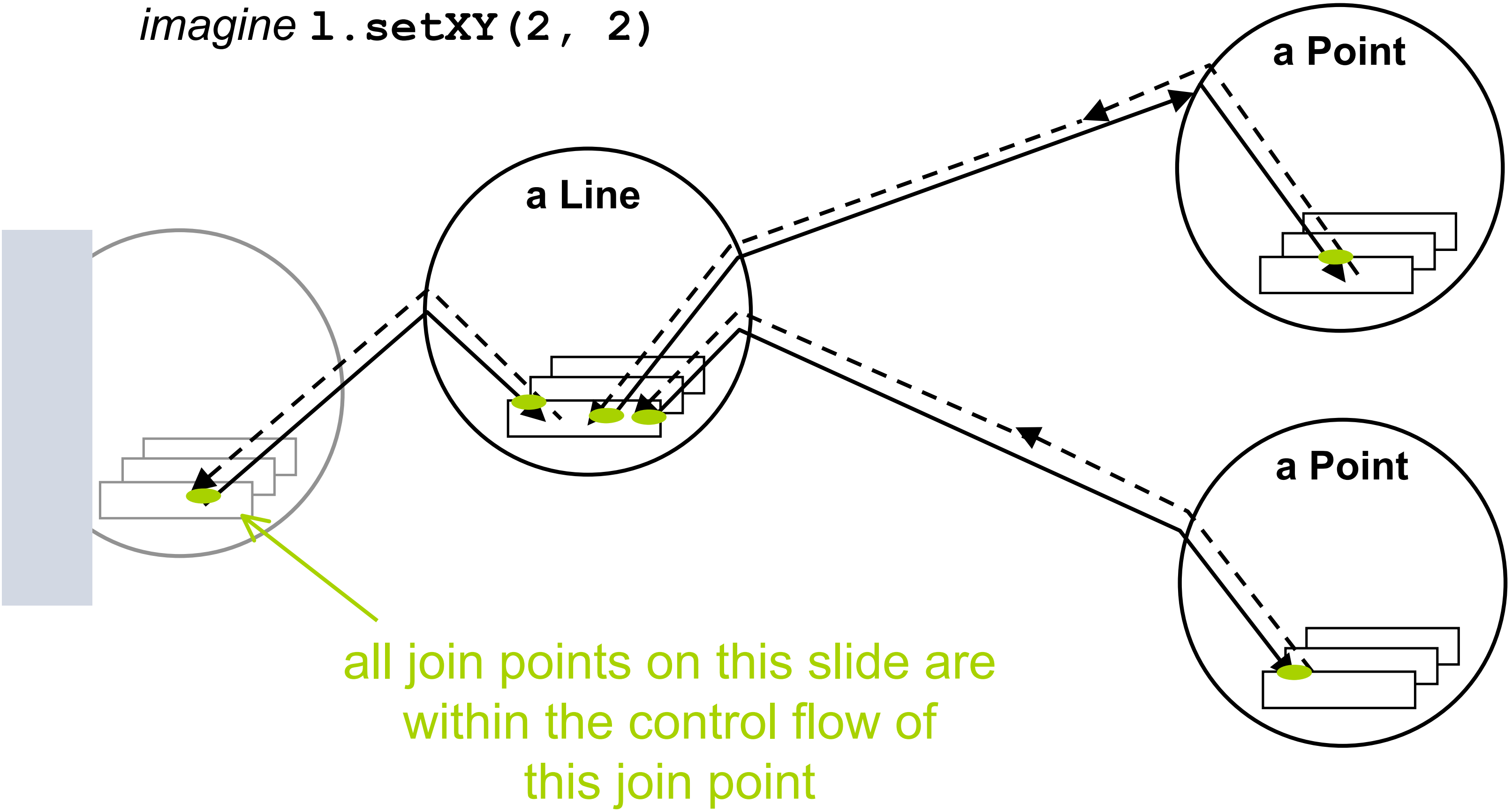
“a point of interest in a dynamic call graph”



- method & constructor call
- method & constructor execution
- field get & set
- returning normally from a method call
- returning from a method call by throwing an error
- exception handler execution
- static & dynamic initialization

join point

imagine `l.setXY(2, 2)`



primitive pointcuts

“a means of identifying join points”

a pointcut is a kind of predicate on join points that:

- can match or not match any given join point and
- optionally, can pull out some of the values at that join point

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

matches if the join point is a method call with this signature

Vrije Universiteit Brussel

pointcuts

Compose like predicates, using &&, || and !

Can crosscut types

Can use interface signatures

or

```
call(voidFigureElement.setXY(int,int))   ||
call(void Point.setX(int))               ||
call(void Point.setY(int))               ||
call(void Line.setP1(Point))             ||
call(void Line.setP2(Point));
```

This pointcut captures all the join points where a FigureElement moves

Vrije Universiteit Brussel

named pointcuts

Defined using the pointcut construct

Can be used in the same way as primitive pointcuts

name

parameters

more on parameters and how pointcut can expose values at join points in a few slides

```
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));
```

name-based and property-based pointcuts

All examples above are name-based pointcuts

Property-based pointcuts specify a pointcut in term of properties of methods other than their name

wildcard

```
call(void Figure.make*(..))

call(public * Figure.* (..))

cflow(move())
```

special primitive pointcut

Vrije Universiteit Brussel

example primitive pointcuts

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)`

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()))`

example pointcuts

Using wildcards

```
execution(* *(..))
call(* set(..))
```

Select elements based on types

```
execution(int *())
call(* setY(long))
call(* Point.setY(int))
call(*.new(int, int)
```

Composed pointcuts

```
target(Point) && call(int *())
call(* *(..)) && (within(Line) || within(Point))
within(*) && execution(*.new(int))
!this(Point) && call(int *(..))
```

Bases on modifiers and negation of modifiers

```
call(public * *(..))
execution(public !static * *(..))
```

advice

before — before proceeding at join point

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

after returning — a value

after throwing — an exception

after — returning either way

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

around — on arrival at join point gets explicit control over when&if program proceeds

Vrije Universiteit Brussel

exposing context in pointcuts

Pointcut can explicitly expose certain values

Advice can use these values

Advice declares and use parameter list

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

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

Pointcuts publish values

Vrije Universiteit Brussel

exposing context in named pointcuts

Named pointcuts may have parameters

When the pointcut is used it publishes its parameters by name

pointcut parameter list

```
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 +
").");}
```

pointcuts publish context

value is 'pulled'

- right to left across ':'   left side : right side
- from pointcuts to user-defined pointcuts
- from pointcuts to advice, and then advice body

aspects

- Wrap up pointcuts and advice in a modular unit
- Are very much like a class, can have methods, fields and initialisers
- Instantation is under the control of AspectJ
- By default an aspect is a singleton, only one aspect instance is created

```
aspect Logging {
    OutputStream logStream = System.err;

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

Vrije Universiteit Brussel

inter-type declarations

Aspects may declare members and fields that cut across multiple existing classes

```
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);    }
    pointcut changes(Point p):
        target(p) && call(void Point.set*(int));
    after(Point p): changes(p) {
       Iterator iter = p.observers.iterator();
       while ( iter.hasNext() ) {
           updateObserver(p, (Screen)iter.next());    }}
    static void updateObserver(Point p, Screen s) {
        s.display(p);    }}
```

Examples

- Development aspects
  - Instrumental during development of a Java application
  - Easily removed from production builds
  - Tracing, profiling&logging, checking pre- and post-conditions, contract enforcement
- Production aspects
  - To be used in both development and production
  - Change monitoring, context passing, providing consistent behavior

Vrije Universiteit Brussel

Tracing

Prints a message at specified method calls

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

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

`thisJoinPoint` is a special variable that is bound to an object that describes the current joinpoint

Profiling and Logging

Counts the number of calls to the rotate method on a line

Counts the number of calls to the set methods of a point that happen within the control flow of those calls to rotate

```
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++;
    }
}
```

Vrije Universiteit Brussel

Pre- and post- conditions

Checks whether the x and y coordinates of a point stay within given bounderies

```
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.");     }}
```

Contract enforcement

Identifies a method call that in a correct program should not exist

Enforces the constraint that only the well-known factory methods can add an element to the registry of figure elements

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

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

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

Contract enforcement

In this example the compiler can signal the error

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

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

    declare error: register() && !canRegister(): "Illegal
call"}
}
```

Vrije Universiteit Brussel

Change monitoring

Supports the code that refreshes the display when a figure element moved

Whithout AOP every method that updates the position of a figure element should manipulate the dirty bit (or call refresh)

```
aspect MoveTracking {
    private static boolean dirty = false;
    public static boolean testAndClear() {
        boolean result = dirty;
        dirty = false;
        return result;    }
    pointcut move():
        call(void FigureElement.setXY(int, int)) ||
        call(void Line.setP1(Point))             ||
        call(void Line.setP2(Point))             ||
        call(void Point.setX(int))               ||
        call(void Point.setY(int));
    after() returning: move() {
        dirty = true;
    }
}
```

Vrije Universiteit Brussel

Context passing

Captures calls to the factory methods of figure elements within the control flow of all calls to methods of a particular client and runs after advice that allows this client to pass context to the new object

Comes in the place of an extra parameter in all methods from the client method down to the factory methods to pass that context

```
aspect ColorControl {
    pointcut CCClientCflow(ColorControllingClient client):
        cflow(call(* * (..)) && target(client));

    pointcut make(): call(FigureElement Figure.make*(..));

    after (ColorControllingClient c) returning
                                        (FigureElement fe):
              make() && CCClientCflow(c) {
        fe.setColor(c.colorFor(fe));
    }
}
```

Vrije Universiteit Brussel

Providing consistent behavior

Enshures that all public methods of a given package log any Error they throw to their caller

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

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

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

The cflow primitive can be used to avoid logging an exception twice when a method of the package calls another public method of the package

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