44. Aspect-Oriented Programming with Aspect/J

Lecturer: Dr. Sebastian Götz

Prof. Dr. Uwe Aßmann

Technische Universität Dresden

Institut für Software- und Multimediatechnik

http://st.inf.tu-dresden.de

27. Juni 2018

1. The Problem of Crosscutting
2. Aspect-Oriented Programming
3. Composition Operators and Point-Cuts
4. Evaluation as Composition System

Literature

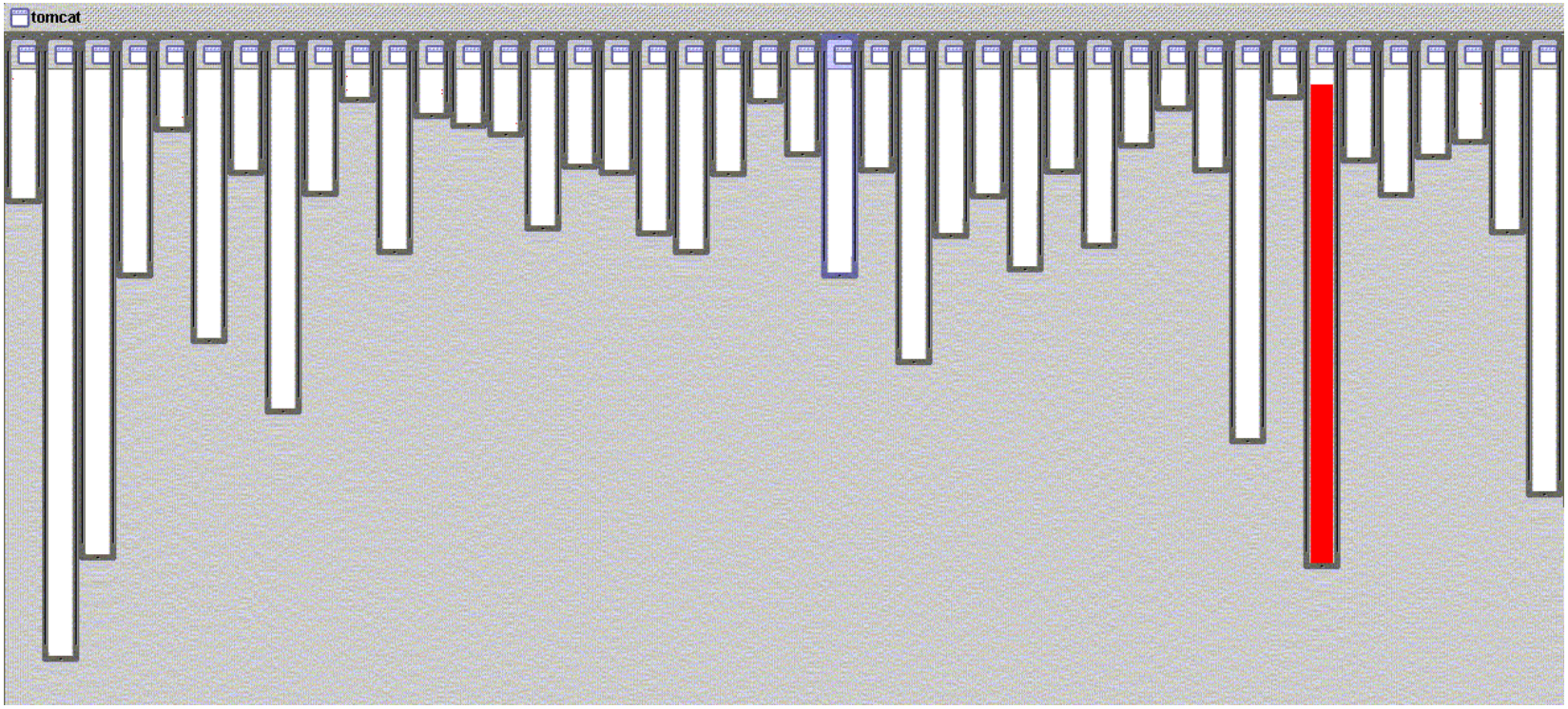
- http://www.eclipse.org/aspectj/
- http://aosd.net/
- [KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin. *Aspect-Oriented Programming*. 1997
- R. Laddad. *Aspect/J in Action*. Manning Publishers. 2003. Book with many details and applications of Aspect/J.

Other literature

- C. V. Lopes. *Aspect-Oriented Programming: An Historical Perspective (What's in a Name?)*. 2002 http://www.isr.uci.edu/tech_reports/UCI-ISR-02-5.pdf
- G. Kiczales. *Aspect Oriented Programming - Radical Research in Modularity.* Google Tech Talk, 57 min http://video.google.com/videosearch?q=Kiczales
- Jendrik Johannes. Component-Based Model-Driven Software Development. PhD thesis, Dresden University of Technology, December 2010.
- Jendrik Johannes and Uwe Aßmann. Concern-based (de-)composition of model-driven software development processes. In D. C. Petriu, N. Rouquette, and O.Haugen, editors, MoDELS (2), volume 6395 of Lecture Notes in Computer Science, pages 47-62. Springer, 2010.

44.1 The Problem of Crosscutting

XML parsing in org.apache.tomcat

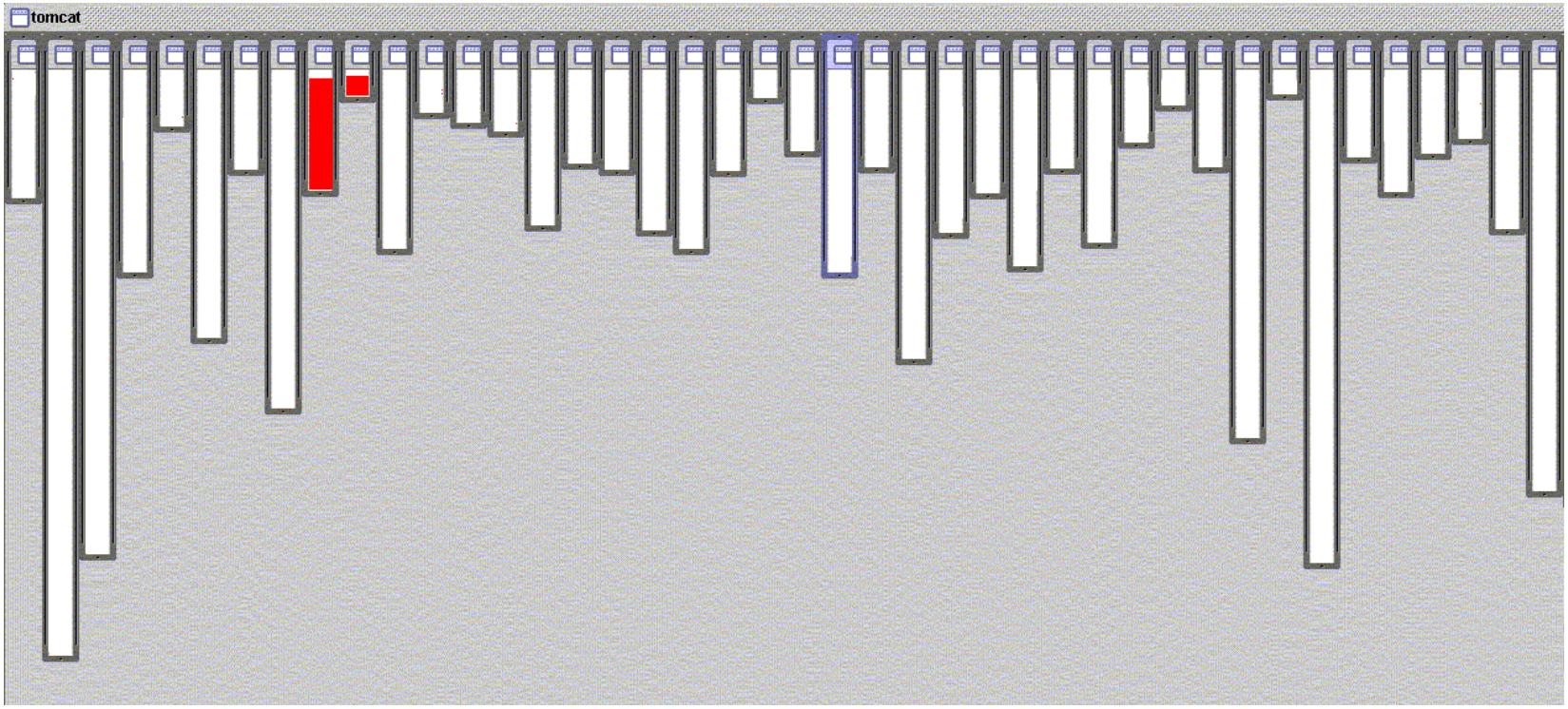


[Picture taken from the aspectj.org website]

Good modularity:

The „red“ concern is handled by code in one class

URL pattern matching in org.apache.tomcat

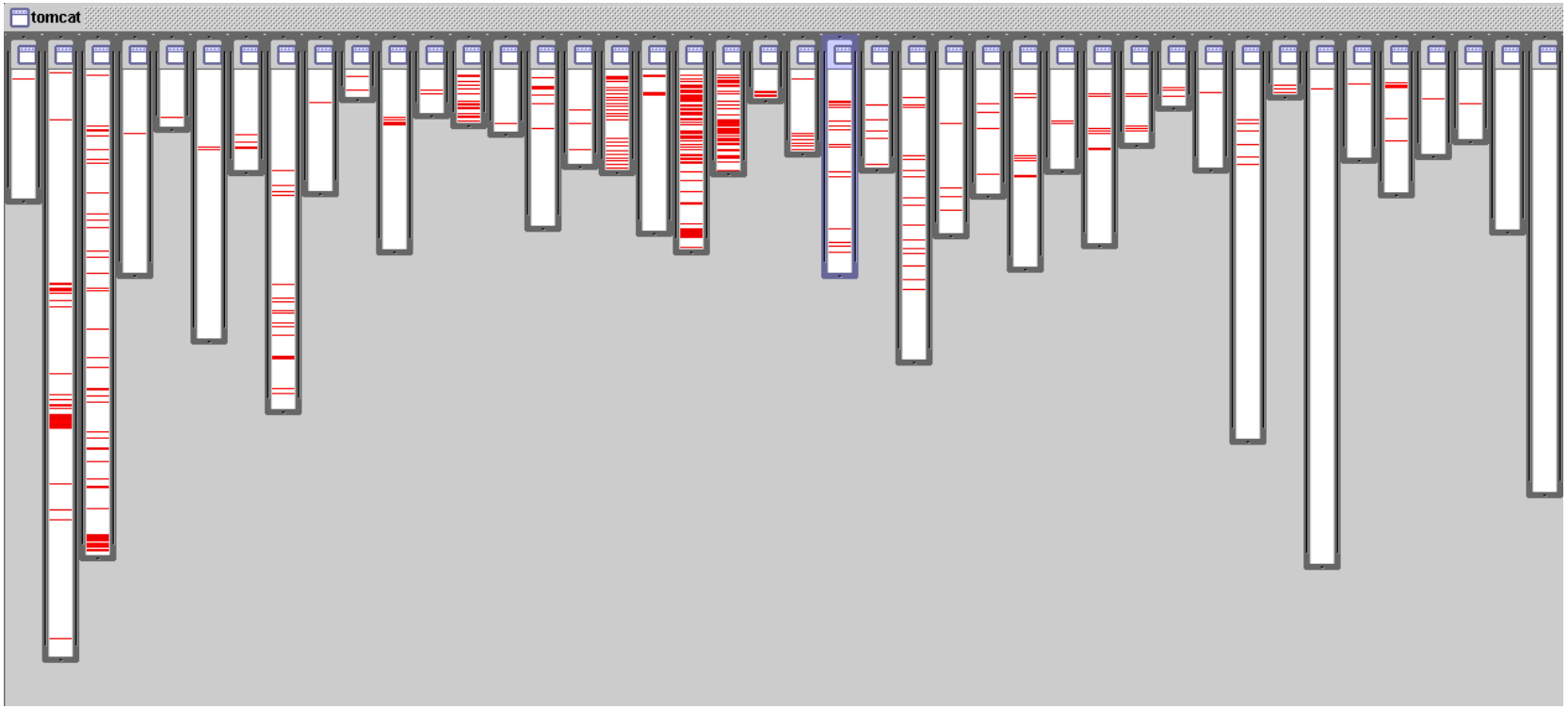


[Picture taken from the aspectj.org website]

Good modularity:

The “red” concern is handled by code in two classes related by inheritance

Logging in org.apache.tomcat



[Picture taken from the aspectj.org website]

BAD modularity:

The concern is handled by code that is scattered over almost all classes

Crosscutting: Scattering and Tangling

- Bad modularity
  - **scattering** – code addressing one concern is spread around in the code
    - "many places in the code are colored with the color of the concern"
  - **tangling** – code in one region addresses multiple concerns
    - "one places in the code is colored with the colors of *many* concerns"
  - Scattering and tangling appear together; they describe different facets of the same problem
    - redundant code
    - difficult to reason about
    - difficult to change
- Good Modularity
  - **separated** – implementation of a concern can be treated as relatively separate entity
  - **localized** – implementation of a concern appears in one part of program
  - **modular** – above + has a clear, well defined interface to rest of system

A first example for scattering

- Every call to foo is preceded by a log call (scattering)
- Observe the green color of the concern “logging”

```
    :
System.out.println("foo called");
Helper.foo(n/3);
```

```
    :
System.out.println("foo called");
Helper.foo(i+j+k);
    :
```

```
    :
System.out.println("foo called")
Helper.foo(x);
    :
```

```
class Helper {
    :
  public static void foo(int n) {

    …
  }
    :
}
```

Classic Solution: Refactoring of Scattered Calls

- Procedures can modularize this case (unless logs use calling context)
- Scattered calls can be refactored *into* called procedures

```
    :

  Helper.foo(n/3);
```

```
    :

  Helper.foo(i+j+k);

    :
```

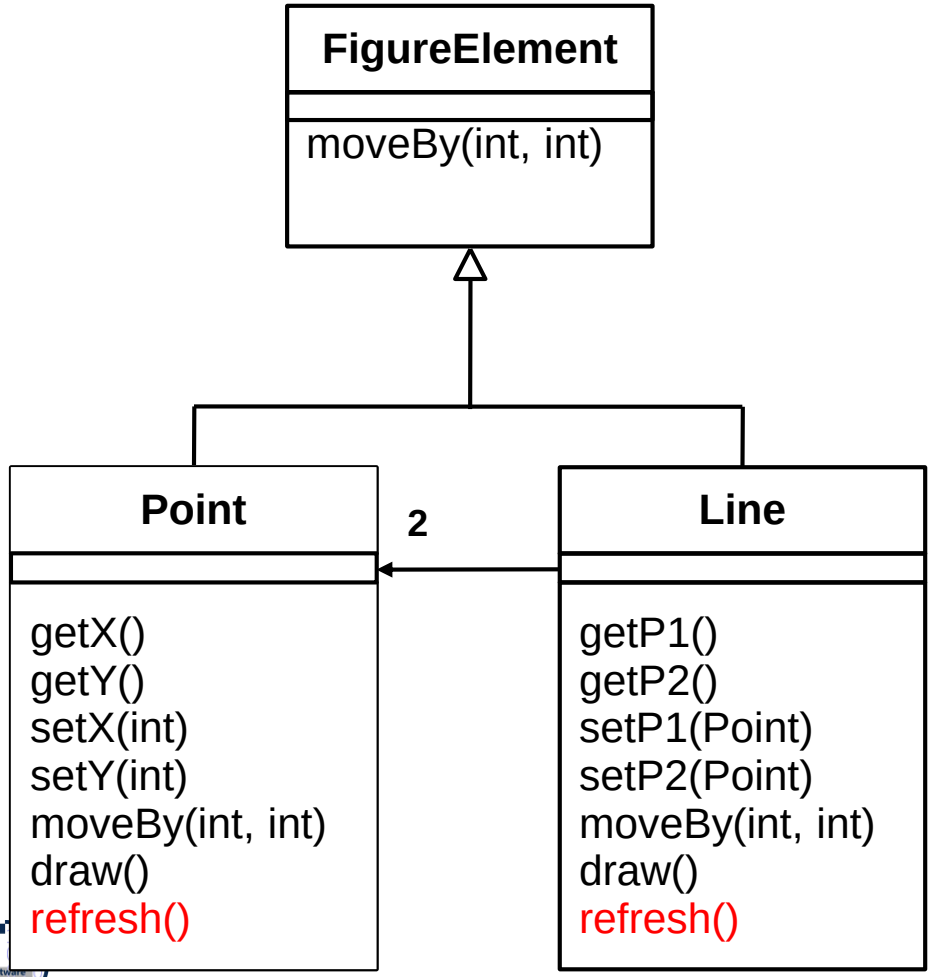
```
    :

  Helper.foo(x);

    :
```

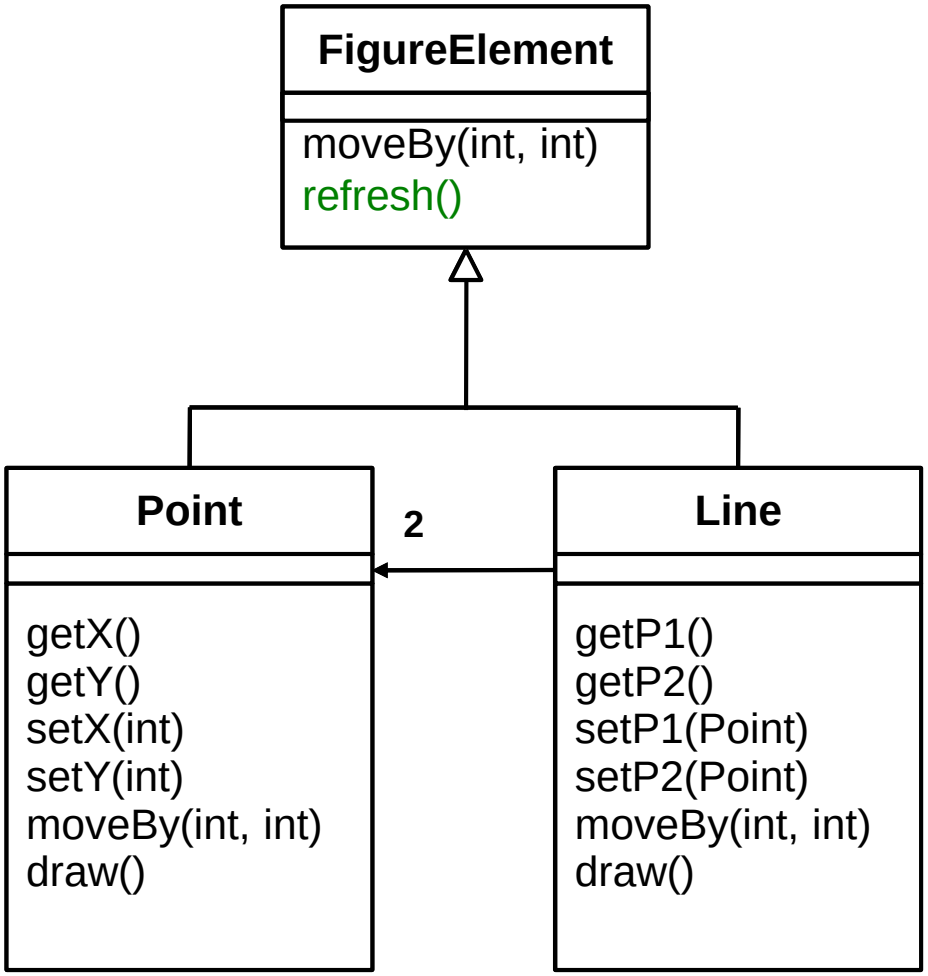
```
class Helper {
    :
  public static void foo(int n) {
      System.out.println("foo called");
    …
  }
    :
}
```

A second example of Scattering and Tangling

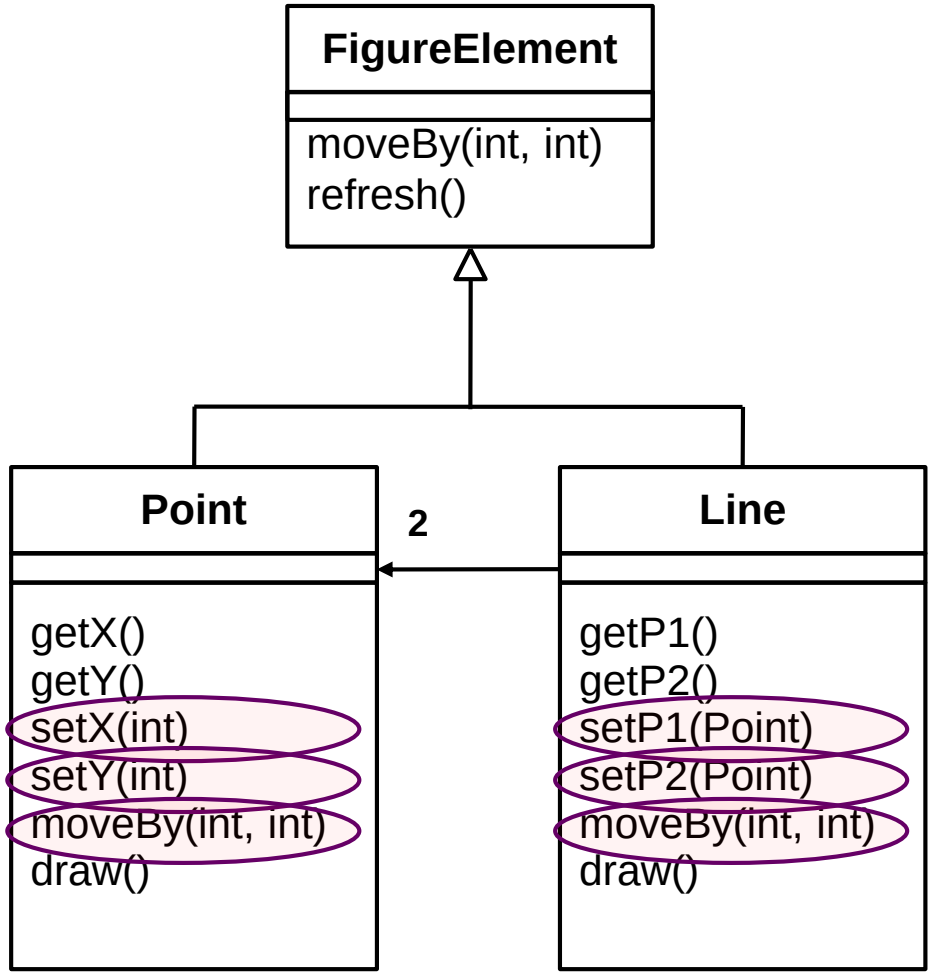
- all subclasses have an identical method

- inheritance can modularize this
- Refactoring **moveUpMethod**





A Final Example of S&T in the Implementation of Methods

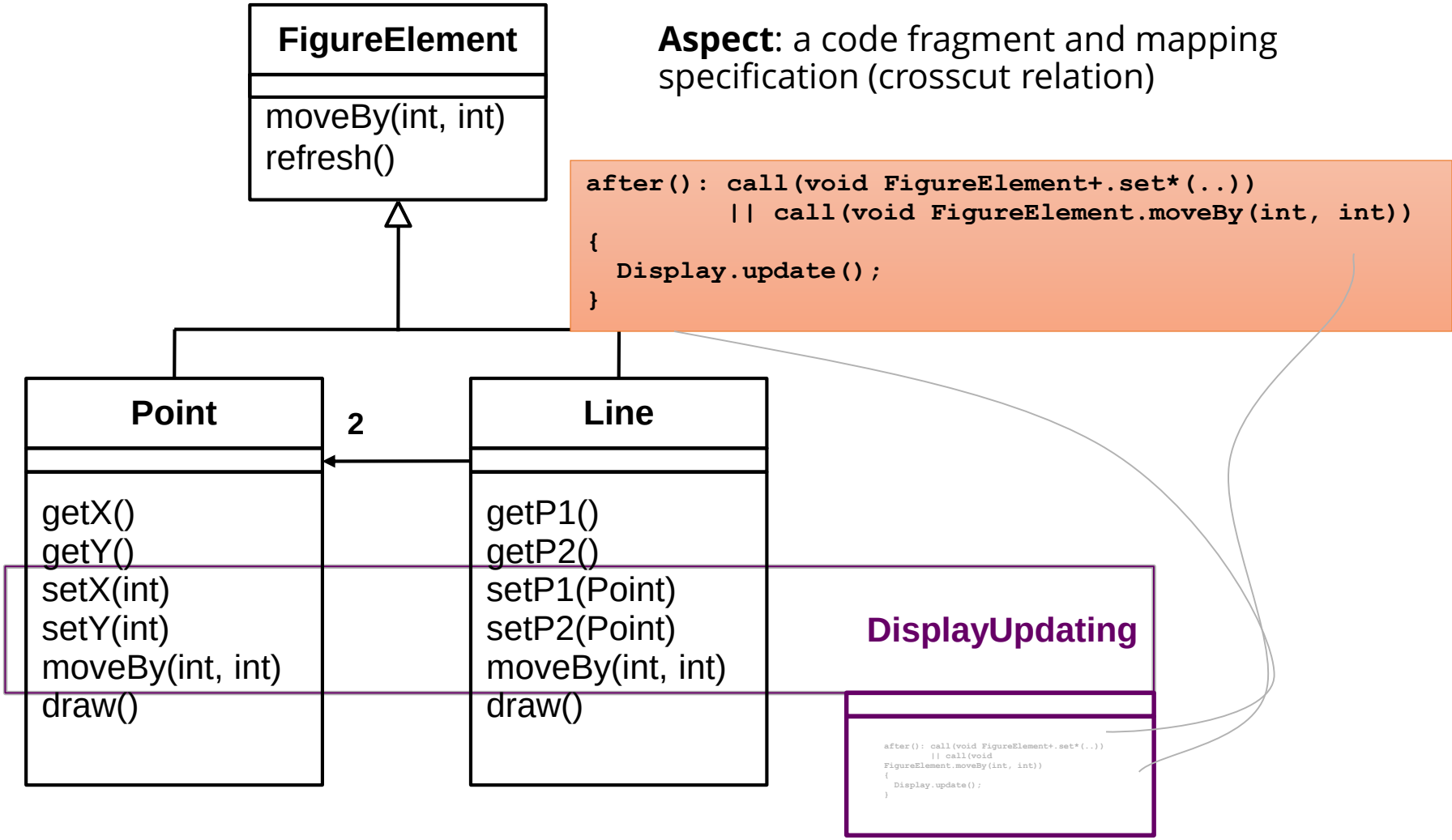


Some scatterings cannot easily be refactored.

Example:
All implementations of these methods end with call to:

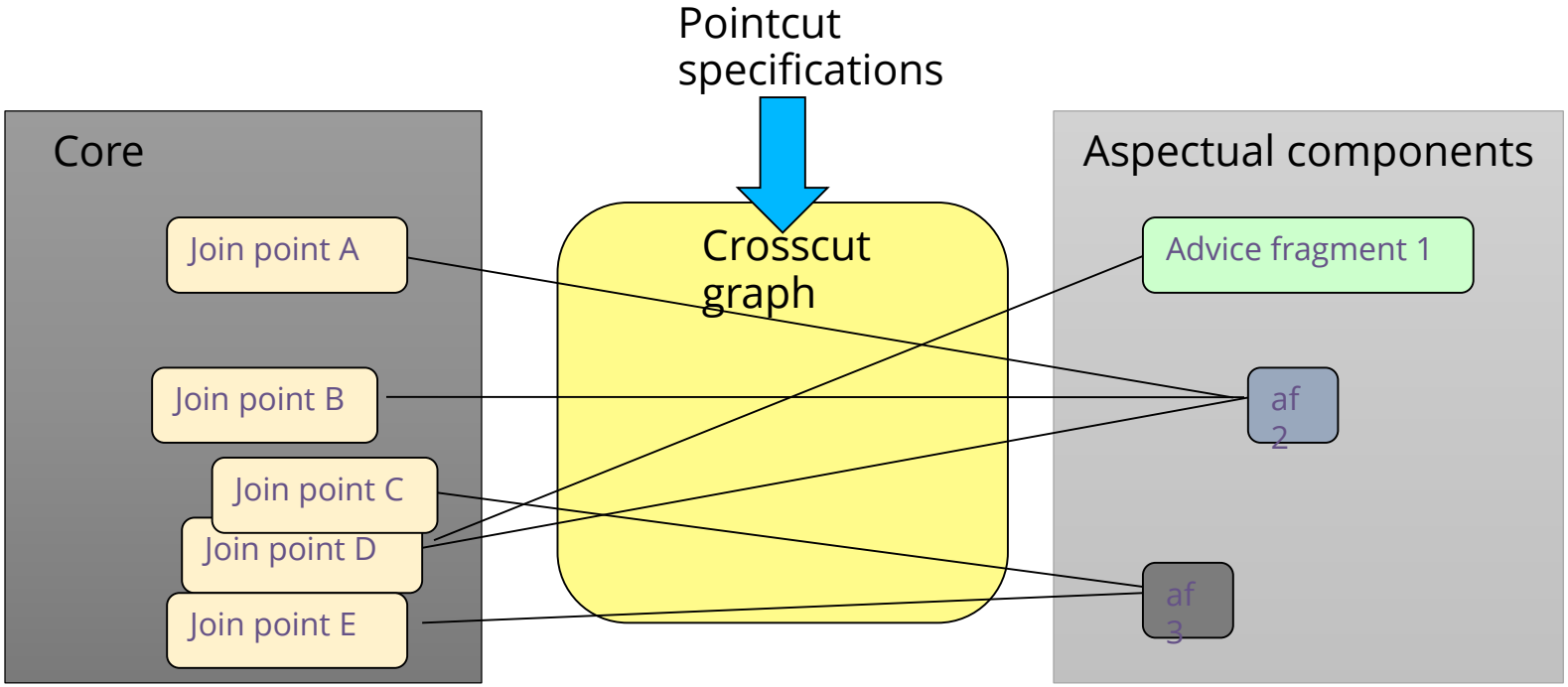
```
Display.update();
```

Needs AOP for a Solution



Crosscut Graphs

- **Crosscuts** are represented by crosscut graphs between core and aspect
- **Pointcut specifications** specify crosscut graphs
- **Aspects (aspectual components)** are specific components containing **advices (code fragments)** to be mixed into a **core** component



Superimposition of Aspects

- **Joinpoints** are places in the core where advices can be woven into
  - e.g., before/after the first/last statements in a method body
- **Pointcuts** denote a selection of joinpoints to which an advice is to be scattered
  - e.g., `before(): call(* *.set*(..))`
- **Aspects** always comprise a code fragment (advice) and a pointcut specification
- All pointcuts of all aspects represent the crosscut graph

Superimposition of Aspects

- Aspect-orientation is ***asymmetric*** composition, i.e., a core is *extended* by an aspect
- Aspectual components are ***superimposed*** to the core, i.e., the unforeseen extension of the core component with extensions.
- Core components are **oblivious** with regard to the aspect**,** i.e., do not see that they are extended [Filman]

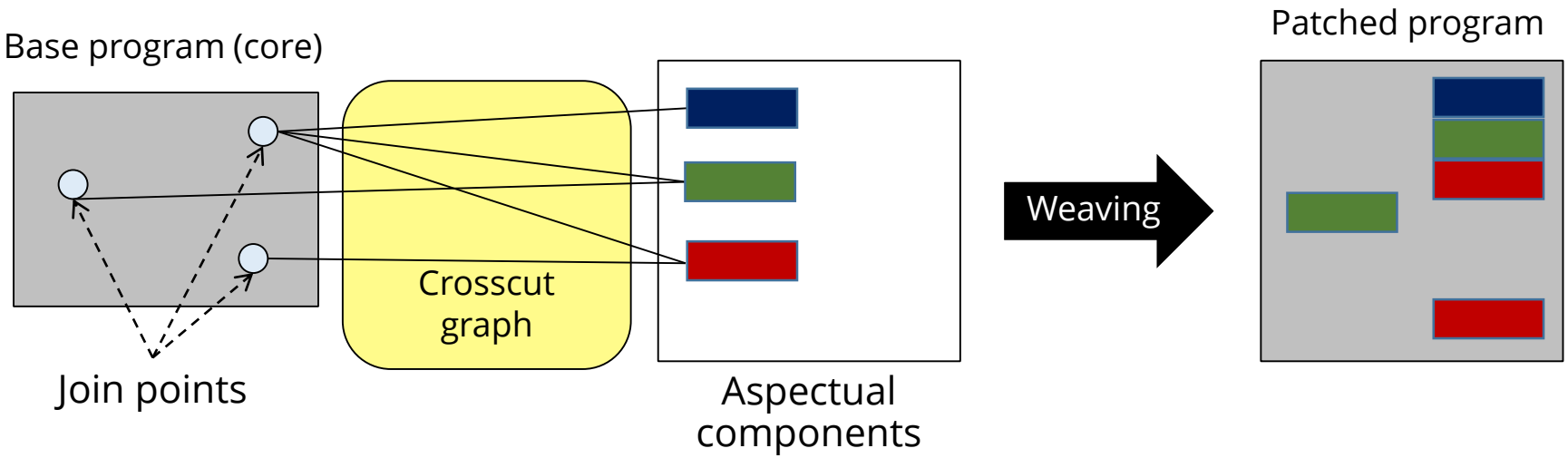
44.2 Aspect-Oriented Programming

The AOP Idea

- **Crosscutting** *(scattering* and *tangling)* is inherent in complex systems
  - The “tyranny of the dominant decomposition”
- AOP proposes to capture crosscutting concerns explicitly
  - in a modular way with *core* components and *aspectual* components
- **Typical examples** of crosscutting concerns
  - Logging
  - Persistence
  - Security
  - Styling

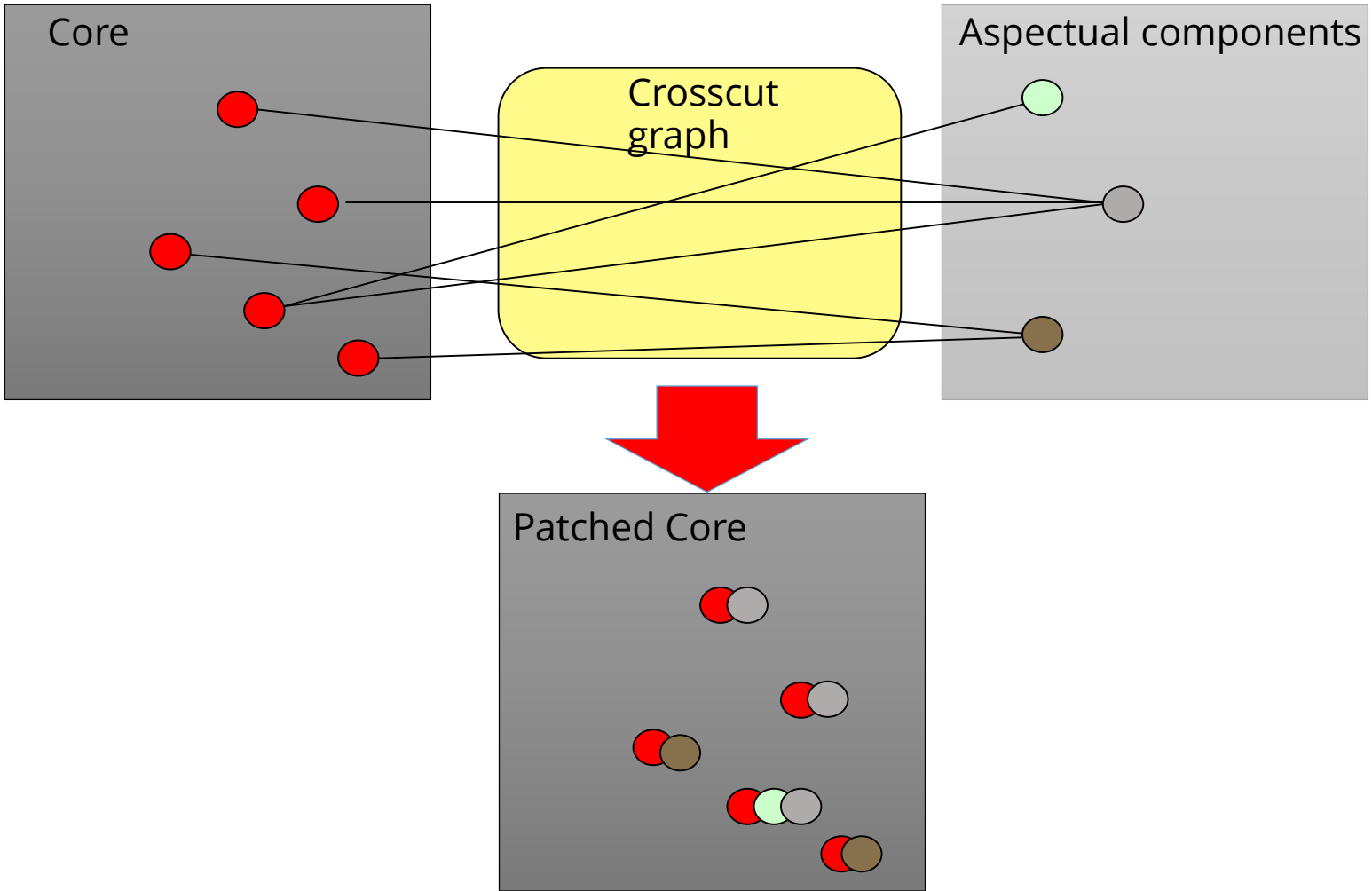
The AOP Idea (2)

- **Weaving** describes the composition, extending a core program at join points
  - At development time, aspects and classes are kept as two, separate artifacts.
  - At run-time, they need to be combined in some way for obtaining the final product.
- Weaving is **asymmetric composition**



Aspects are Woven by Interpretation of the Crosscut Graphs

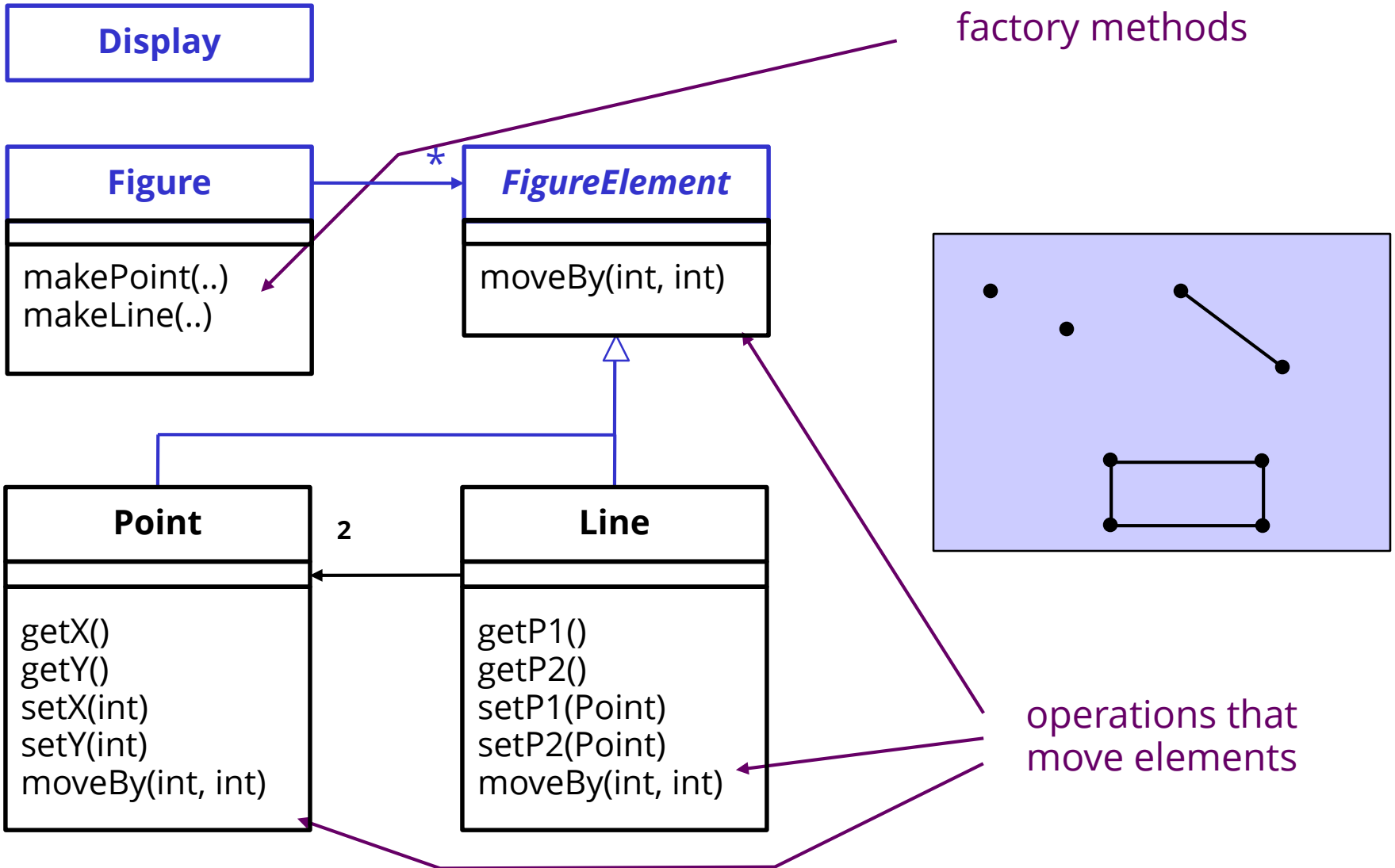
- Crosscut graphs are interpreted to insert advice fragments into core joinpoints



Aspect/J: a Weaver for Java

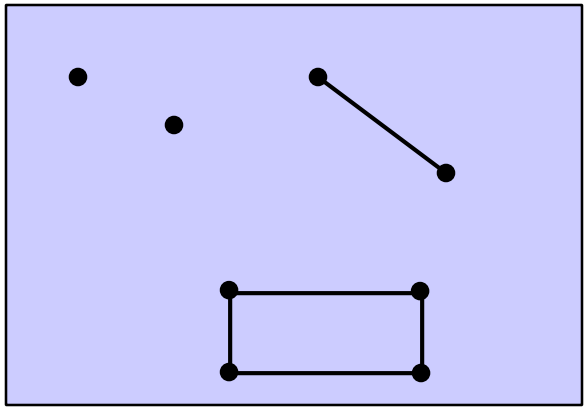
- First production-quality AOP-technology
- Allows specifying aspectual components for crosscutting concerns as separate entities: Aspects
- Two types of joinpoints:
  - **Static join points** are code positions, hooks, open for extension
  - **Dynamic join points** are some points in the execution trace of an application, open for extension

Example: A Simple Figure Editor



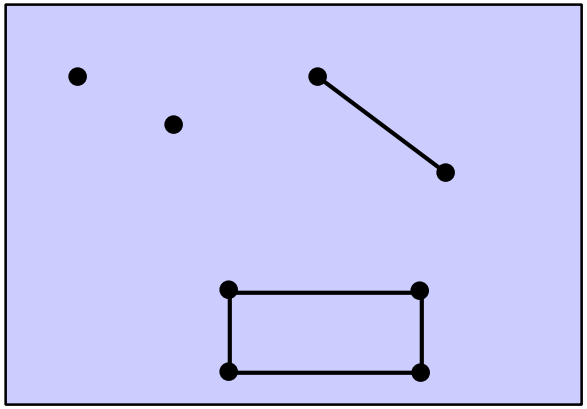
Example: A Simple Figure Editor (Java)


```
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 moveBy(int dx, int dy) { ... }
}

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 moveBy(int dx, int dy) { ... }
}
```



Problem: Display Updating

- Collection of figure elements
  - that move periodically
  - must refresh the display as needed



we will assume just a single display

Static Joinpoints in Aspect/J

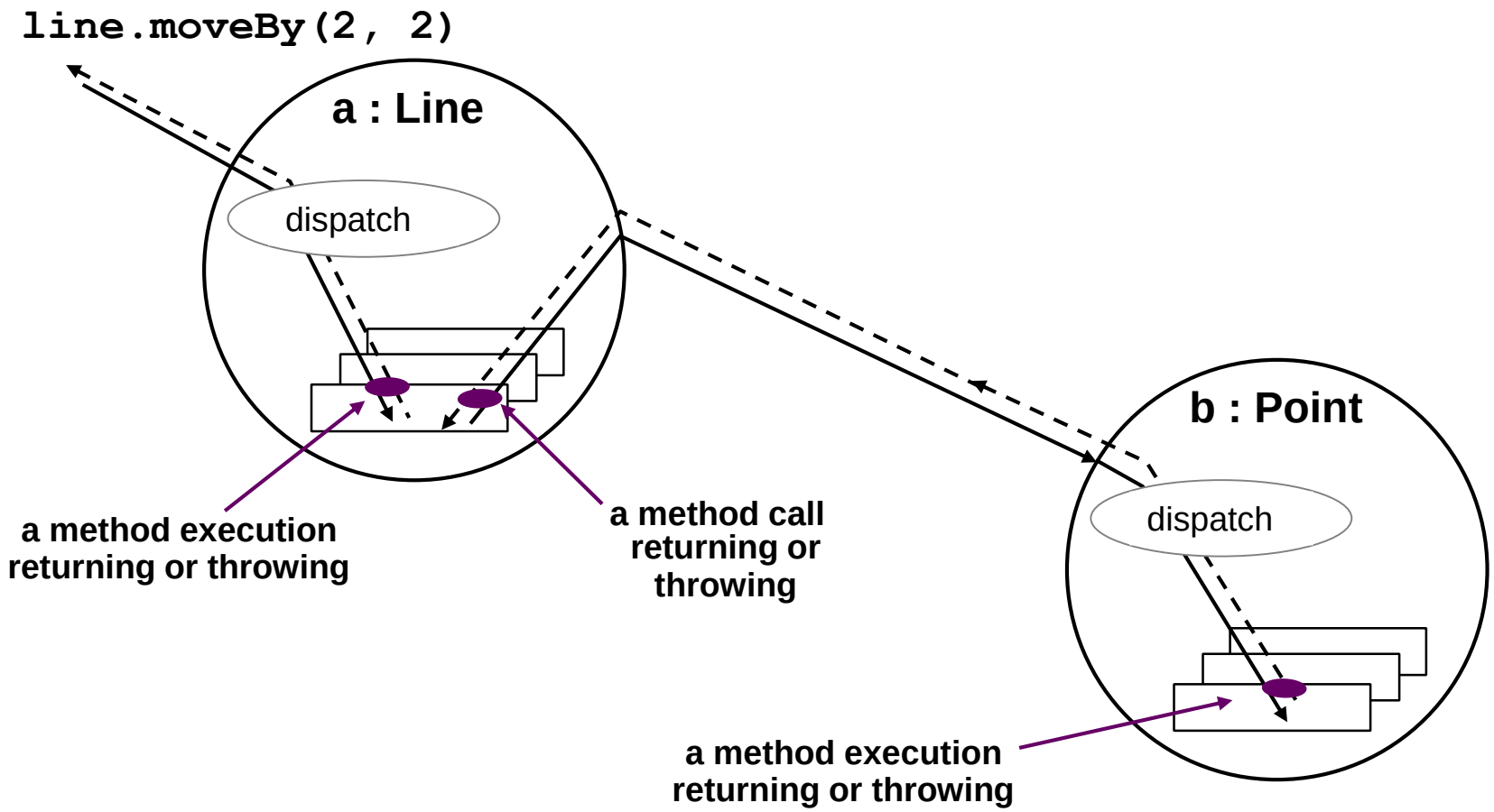
- **Static joinpoints** are code positions which can be addressed and extended by the weaver.
- An advice may extend a static joinpoint.

`target(Point)`	
`target(graphics.geom.Point)`	
`target(graphics.Point.moveBy)`	joinpoint method “moveBy”
`target(graphics.Point.move*)`	joinpoint methods with prefix “move”
`target(graphics.geom.*)`	any type in graphics.geom
`target(graphics.*)`	any type in any sub-package of graphics

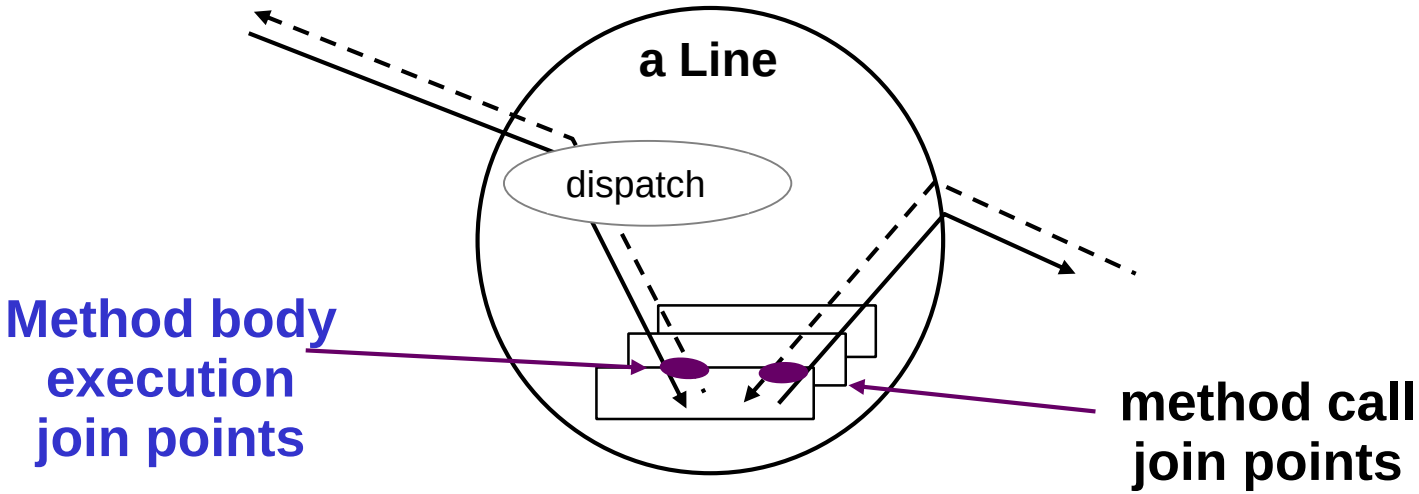
“*” is wild card
“..” is multi-part wild card

Dynamic Join Points in Aspect/J (Dynamic Hooks)

- A **dynamic join point** is a *hook (extension point)* in the execution trace of a program, also in dynamic call graph

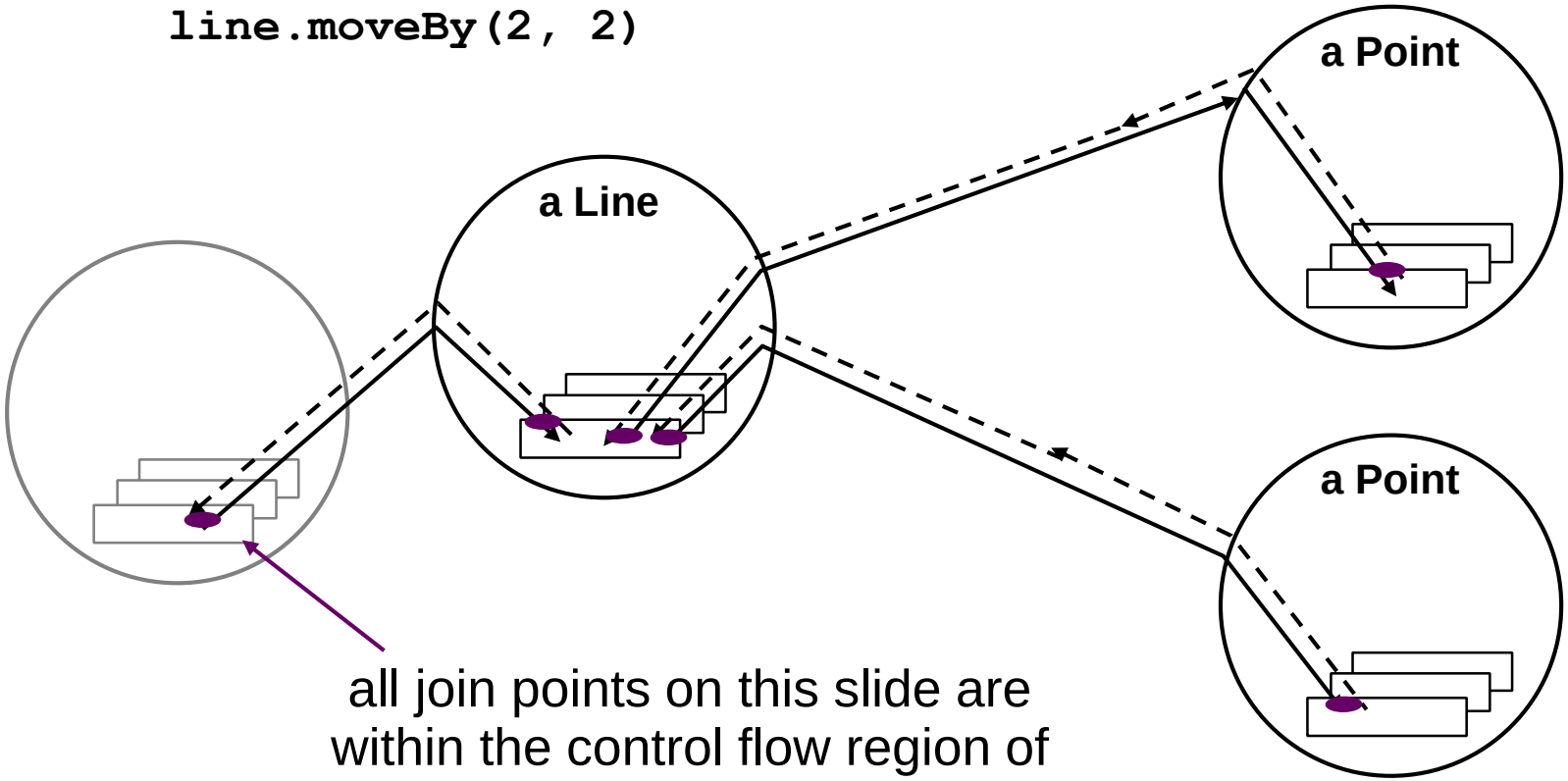


Dynamic Join Point Terminology



- The **join-point model** of Aspect/J defines several types of join points (join-point types)
  - method & constructor call
  - method & constructor execution
  - field get & set
  - exception handler execution
  - static & dynamic initialization

Join Point Terminology



all join points on this slide are within the control flow region of this join point

Primitive Pointcuts

- A **pointcut** is a specification *addressing a set of 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
  - "a means of identifying join points"

- Example: `call(void Line.setP1(Point))`

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

Pointcut Composition

- Pointcuts are logical expressions in Aspect/J, they compose like predicates, using &&, || and !

a “void Line.setP1(Point)” call

or

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

a “void Line.setP2(Point)” call

whenever a Line receives a
“void setP1(Point)” or “void setP2(Point)” method call

User-Defined Pointcuts

- User-defined (named) pointcuts can be used in the same way as primitive pointcuts

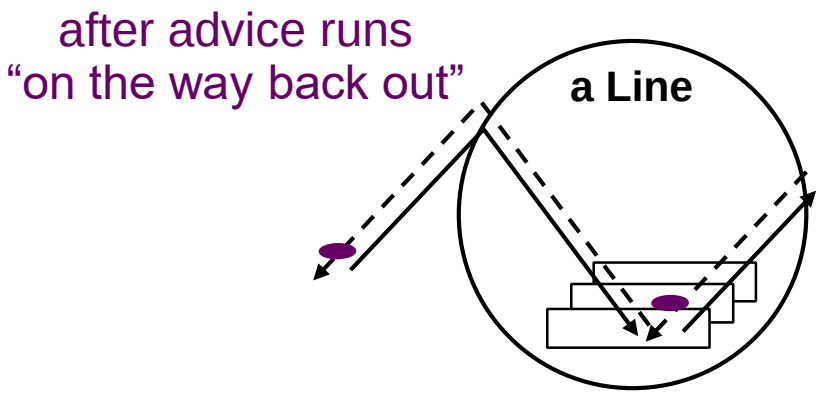
name parameters

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

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

After Advice

- An **after advice** is a fragment describing the action to take after computation under join points




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

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

A Simple Aspectual Component

- An **aspect (aspectual component, aspectual class)** defines a special class collecting all fragments related to one concern and which will crosscut core classes
  - With one or several **advices** (fragments plus composition expression)
  - With at least one pointcut expressing the crosscut graph

```
aspect DisplayUpdating {

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

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

The Effect of AspectJ Weaving


```
class Line {
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }

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

```
aspect DisplayUpdating {

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

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

Weaver

```
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();
  }
}
```

A multi-class aspect

- With pointcuts cutting across multiple classes

```
aspect DisplayUpdating {

  pointcut move():
    call(void FigureElement.moveBy(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() {
    Display.update();
  }
}
```

Using values at join points

- A pointcut can explicitly expose certain run-time values in parameters
- An advice can use the exposed value

Pointcut Parameter defined and used

```
pointcut move(FigureElement figElt):
  target(figElt) &&
  (call(void FigureElement.moveBy(int, int)) ||
   call(void Line.setP1(Point)) ||
   call(void Line.setP2(Point)) ||
   call(void Point.setX(int)) ||
   call(void Point.setY(int)));

after(FigureElement fe) returning: move(fe) {
  <fe is bound to the figure element>
}
```

Pointcut parameter

advice parameters

Join Point Qualifier “target”

- A **join point qualifier** does two things:
  - exposes information from the context of the join point (e.g., the state of the target object of a message)
  - tests a predicate on join points (e.g., a dynamic type test - any join point at which a target object is an instance of type name)
- target(<type name> | <formal reference>)
  - target(Point)
  - target(Line)
  - target(FigureElement)
- “any join point” means it matches join points of all kinds:
  - method & constructor call join points
  - method & constructor execution join points
  - field get & set join points
  - exception handler execution join points
  - static & dynamic initialization join points

Wildcarding in pointcuts

"*" is wild card
".." is multi-part wild card

```
target(Point)
target(graphics.geom.Point)
target(graphics.geom.*)        any type in graphics.geom
target(graphics..*)            any type in any sub-package of graphics

call(void Point.setX(int))
call(public * Point.*(..))     any public method on Point
call(public * *(..))           any public method on any type

call(void Point.getX())
call(void Point.getY())
call(void Point.get*())
call(void get*())              any getter

call(Point.new(int, int))
call(new(..))                  any constructor
```

Other Primitive Pointcuts

- `handler(Exception)`
  - any Exception handler of the respective Exception type

- `get(int Point.x)`
- `set(int Point.x)`
  - field reference or assignment join points

Context & multiple classes


```
aspect DisplayUpdating {

  pointcut move(FigureElement figElt):
    target(figElt) &&
    (call(void FigureElement.moveBy(int, int)) ||
     call(void Line.setP1(Point))              ||
     call(void Line.setP2(Point))              ||
     call(void Point.setX(int))                ||
     call(void Point.setY(int)));

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

Without AspectJ

- no locus of “display updating”
  - evolution is cumbersome
  - changes in all classes

```
class Line {
  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 {
  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);
  }
}
```

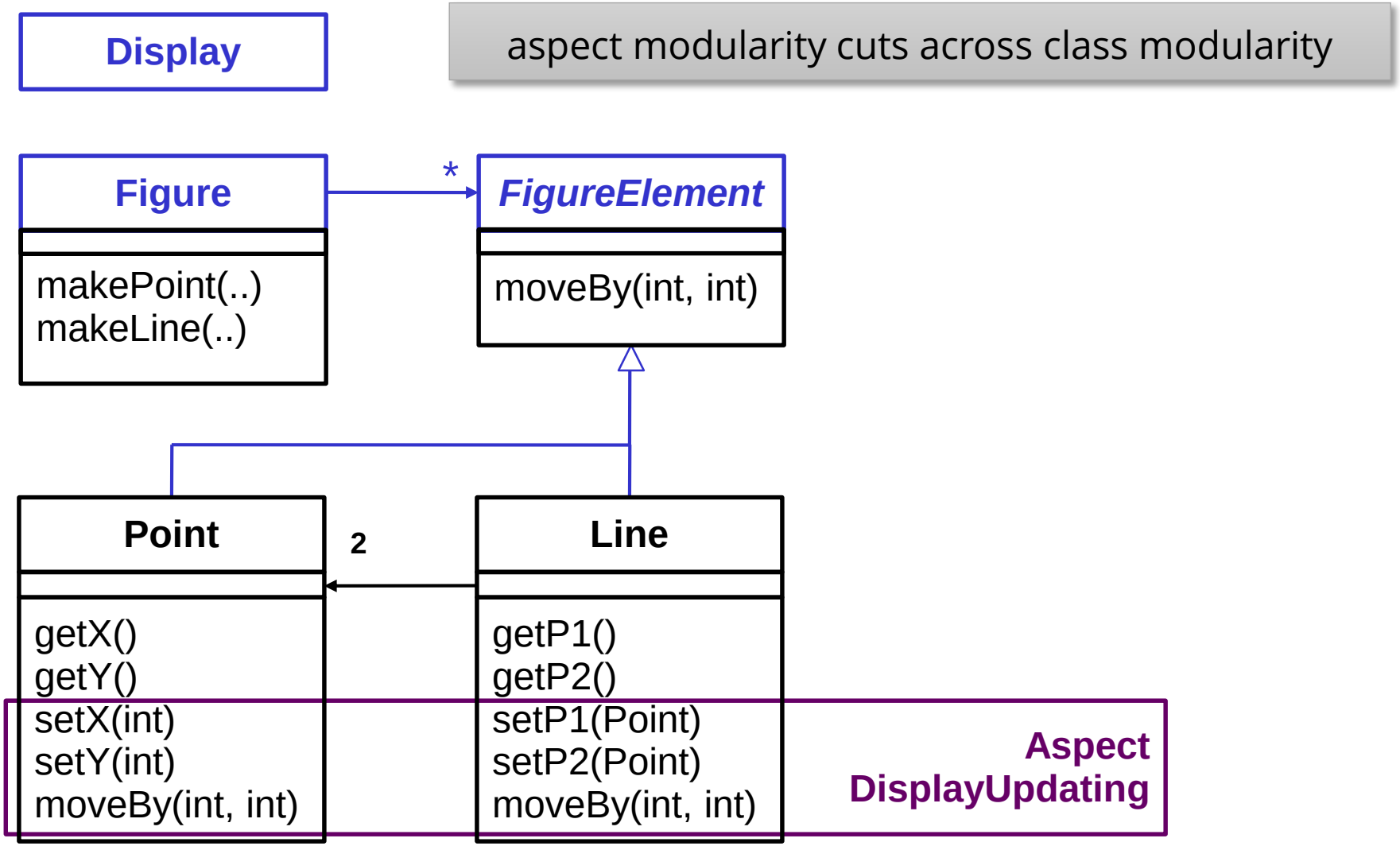
With AspectJ

- clear display updating module
  - all changes in single aspect
  - evolution is modular

```
aspect DisplayUpdating {

  pointcut move(FigureElement figElt):
    target(figElt) &&
    (call(void FigureElement.moveBy(int, int) ||
     call(void Line.setP1(Point))             ||
     call(void Line.setP2(Point))             ||
     call(void Point.setX(int))               ||
     call(void Point.setY(int)));


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

```
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;
  }
}

class Point {
  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;
  }
}
```

Aspects Crosscut Classes



44.3 Composition Operators of Aspect/J

Types of Advice Composition Operators

- **before** — before proceeding at join point
- **after returning** — a value to a method-call join point
- **after throwing** — a throwable (exception) 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

Special Methods (Hooks in Advices)

- For each around advice with the signature

```
<Tr> around(T1 arg1, T2 arg2, …)
```

- there is a special method with the signature

```
<Tr> proceed(T1, T2, …)
```

- available only in an around advice, meaning *"run what would have run if this around advice had not been defined"*

Aspect/J Introductions of Members

- An aspect can also introduce new attributes and methods to 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);
    }
}
```

44.4 Evaluation

44.6 Evaluation: Aspects as Composition System

Component model

Source components (core with join-points)

Aspects (advices and pointcuts)

Composition technique

Weaving

Asymmetric composition (extension)

Crosscut graph definitions (with pointcuts)

Expression-based

Composition language

How to get started?

- http://www.aosd.net/

- **Aspect/J**
  - uses compile-time bytecode weaving,
    - supports weaving aspects to existing *.class files (based on BCEL)
  - Aspect/J was taken over by IBM as part of the Eclipse project: http://www.eclipse.org/aspectj

- **AspectC++**
  - is an aspect-oriented extension to the C++ programming language. https://www.aspectc.org/

- **PostSharp**
  - is an aspect-oriented extension for .NET https://www.postsharp.net/

The End

- Many slides courtesy to Wim Vanderperren, Vrije Universitet Brussel, and the Aspect/J team