24) Aspect-Oriented Programming with Aspect/J

Prof. Dr. Uwe Aßmann

Florian Heidenreich

Technische Universität Dresden

Institut für Software- und Multimediatechnik

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

Version 11-0.1, Juli 5, 2011

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.
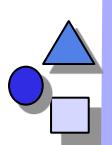
Other literature

- 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

24.1 The Problem of Crosscutting
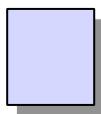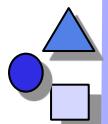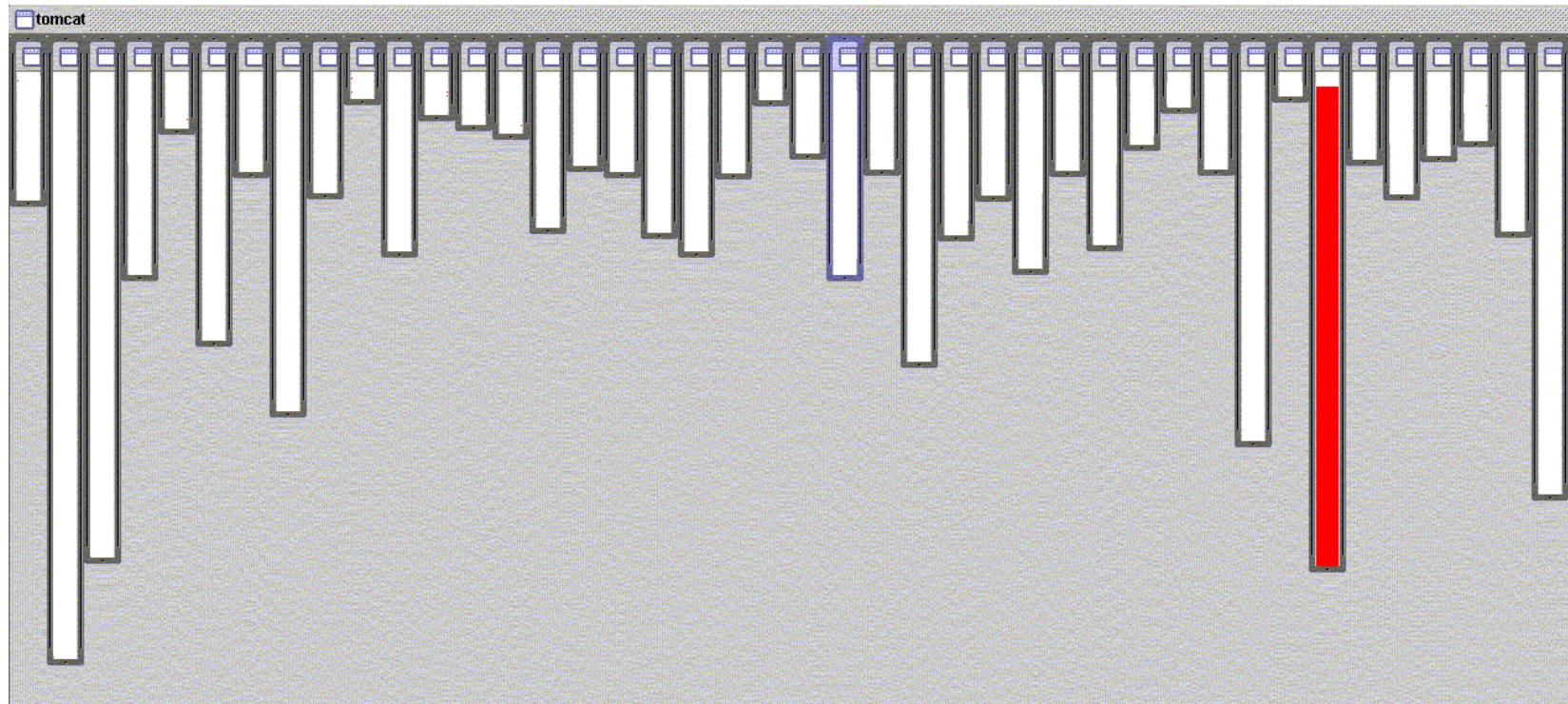
[Picture taken from the aspectj.org website]

Good modularity:

handled by code in one class
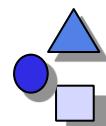
URL pattern matching in org.apache.tomcat
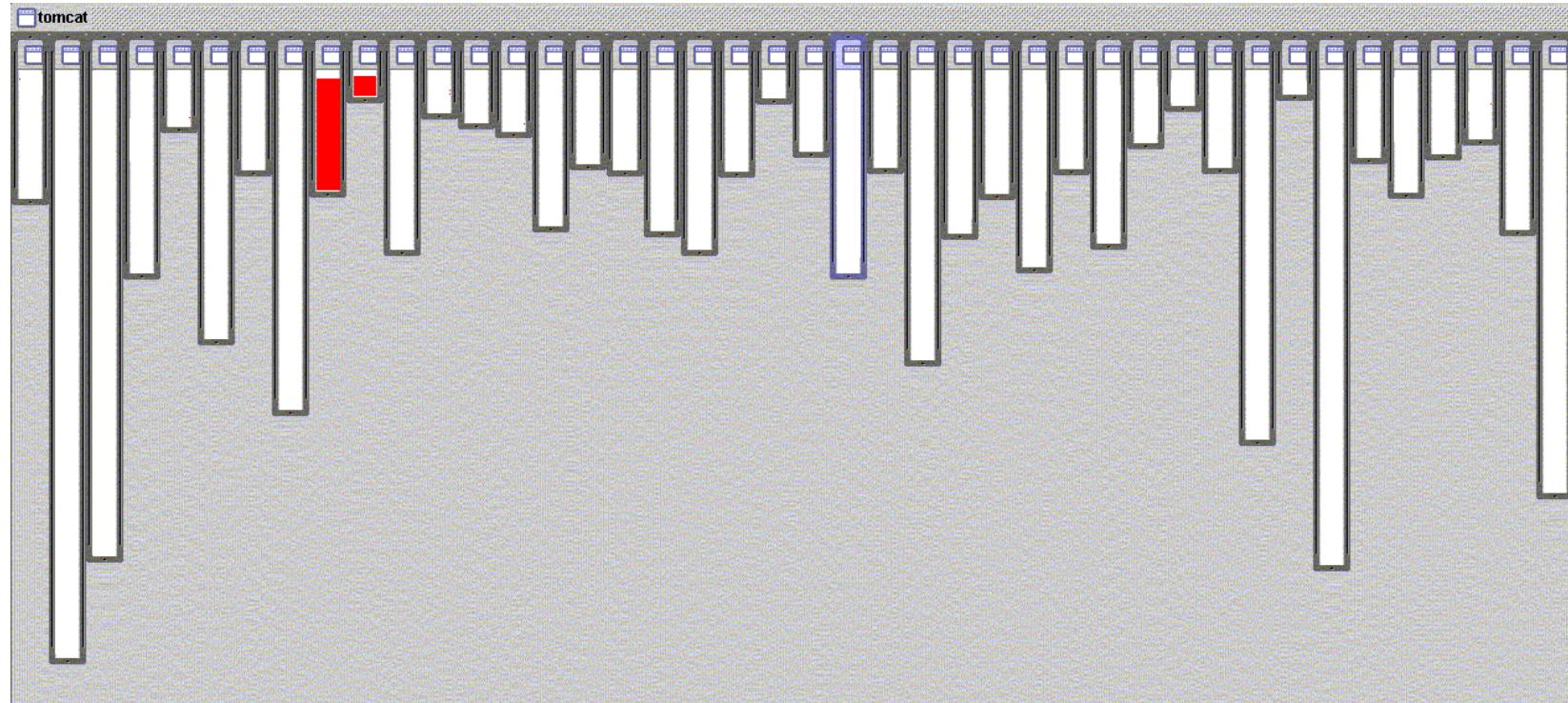
[Picture taken from the aspectj.org website]

Good modularity:

handled by code in two classes related by inheritance
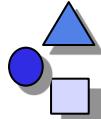
Logging in org.apache.tomcat

[Picture taken from the aspectj.org website]
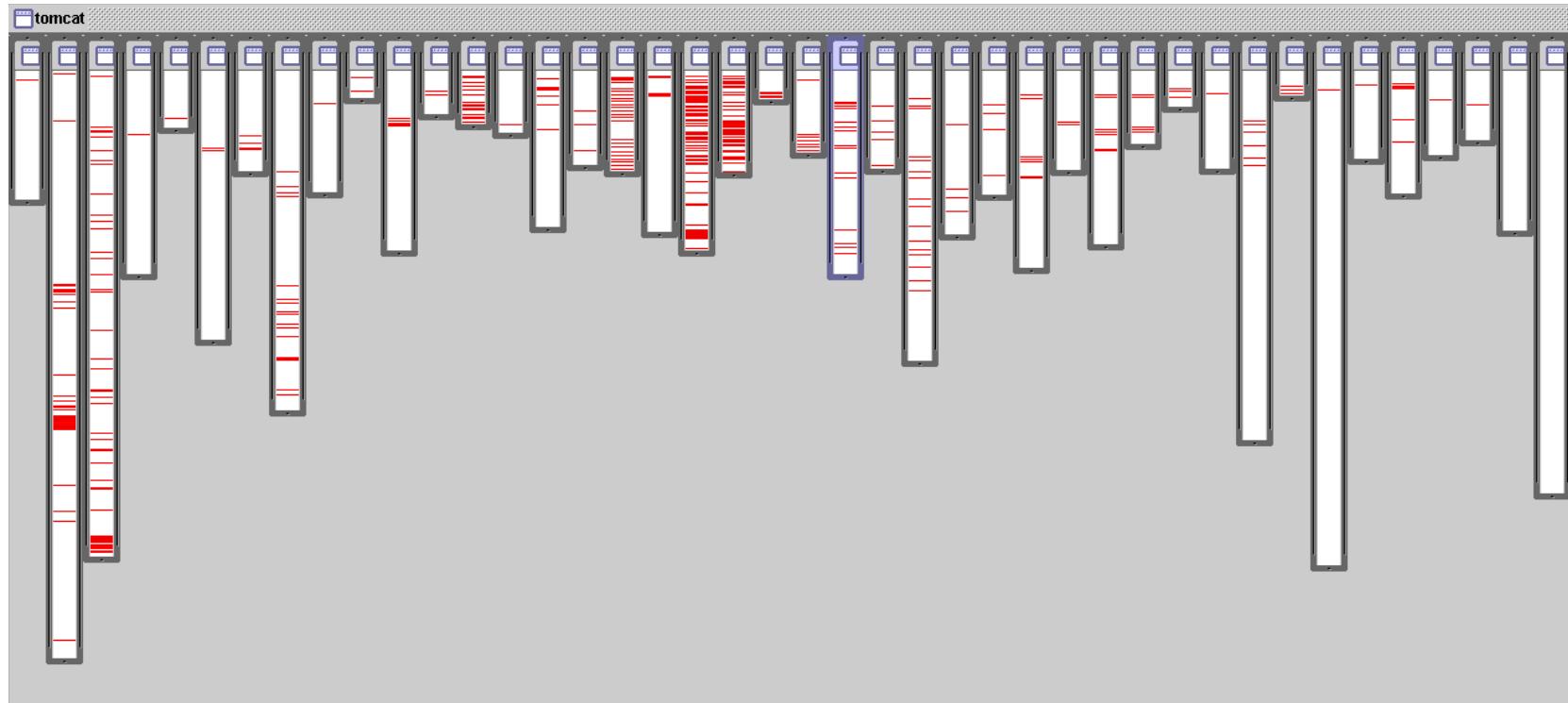
BAD modularity:

handled by code that is scatered over almost all classes
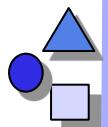
Comparison

Bad modularity

► **scattering** – code addressing one concern is spread around in the code

► **tangling** – code in one region addresses multiple 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
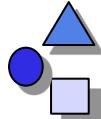
A first example for scattering

► every call to foo is preceded by a log call (scattering)

```
      :
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) {

    …

  }
    :

}
```
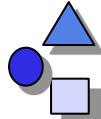
Solution: Refactoring of Scattered Calls

► 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");

    …

  }
    :

}
```
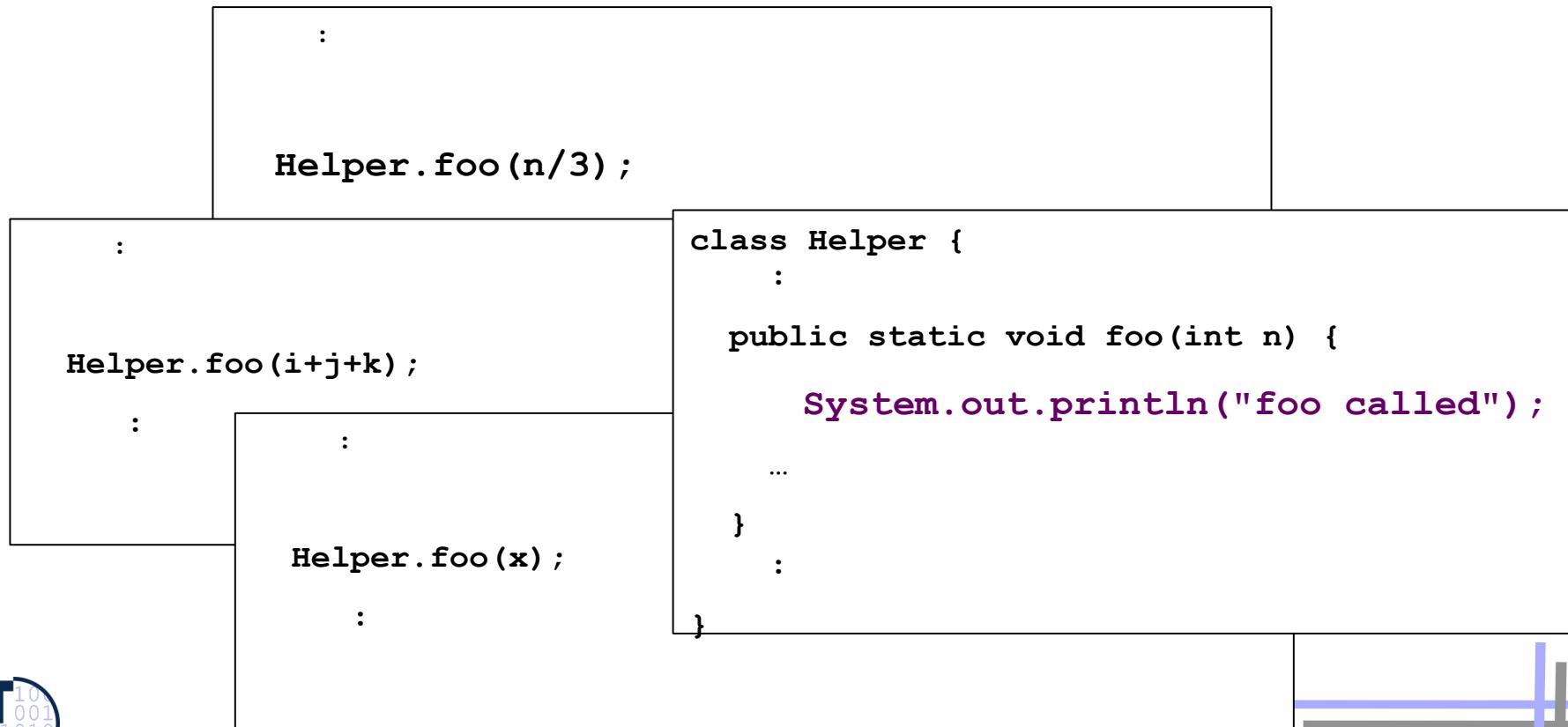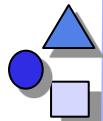
A second example of S&T

► all subclasses have an identical method

```
        ┌─────────────────────┐
        │   FigureElement     │
        ├─────────────────────┤
        │   moveBy(int, int)  │
        │                     │
        └─────────────────────┘
                 △
        ┌────────┴────────┐
┌──────────────┐  2  ┌──────────────┐
│    Point     │◄────│     Line     │
├──────────────┤     ├──────────────┤
│ getX()       │     │ getP1()      │
│ getY()       │     │ getP2()      │
│ setX(int)    │     │ setP1(Point) │
│ setY(int)    │     │ setP2(Point) │
│ moveBy(int, int) │ │ moveBy(int, int) │
│ draw()       │     │ draw()       │
│ refresh()    │     │ refresh()    │
└──────────────┘     └──────────────┘
```
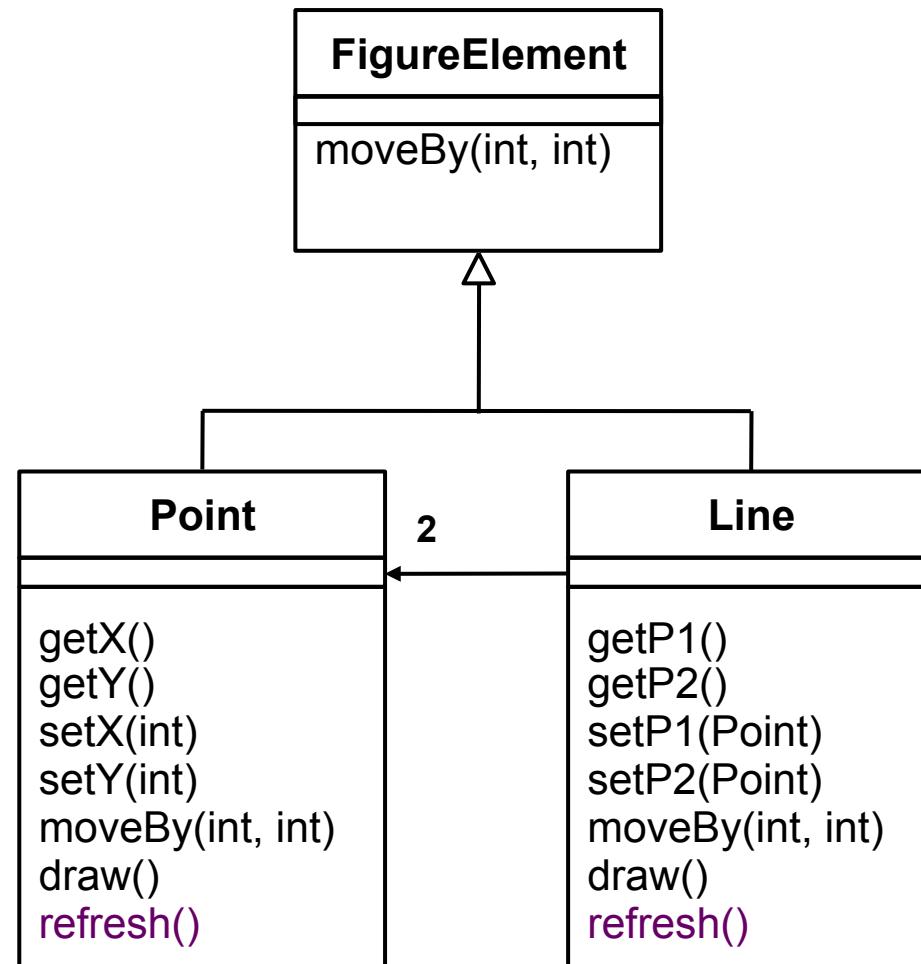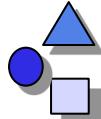
Solution: Refactoring (MoveUp Method)

▶ inheritance can modularize this

```
┌─────────────────────────┐
│      FigureElement      │
├─────────────────────────┤
│  moveBy(int, int)       │
│  refresh()              │
└─────────────────────────┘
```

```
┌──────────────────┐          2          ┌──────────────────┐
│      Point       │ ◄─────────────────  │       Line       │
├──────────────────┤                     ├──────────────────┤
│  getX()          │                     │  getP1()         │
│  getY()          │                     │  getP2()         │
│  setX(int)       │                     │  setP1(Point)    │
│  setY(int)       │                     │  setP2(Point)    │
│  moveBy(int, int)│                     │  moveBy(int, int)│
│  draw()          │                     │  draw()          │
└──────────────────┘                     └──────────────────┘
```

A final example of S&T

FigureElement

moveBy(int, int)
refresh()

Point

getX()
getY()
setX(int)
setY(int)
moveBy(int, int)
draw()

2

Line

getP1()
getP2()
setP1(Point)
setP2(Point)
moveBy(int, int)
draw()
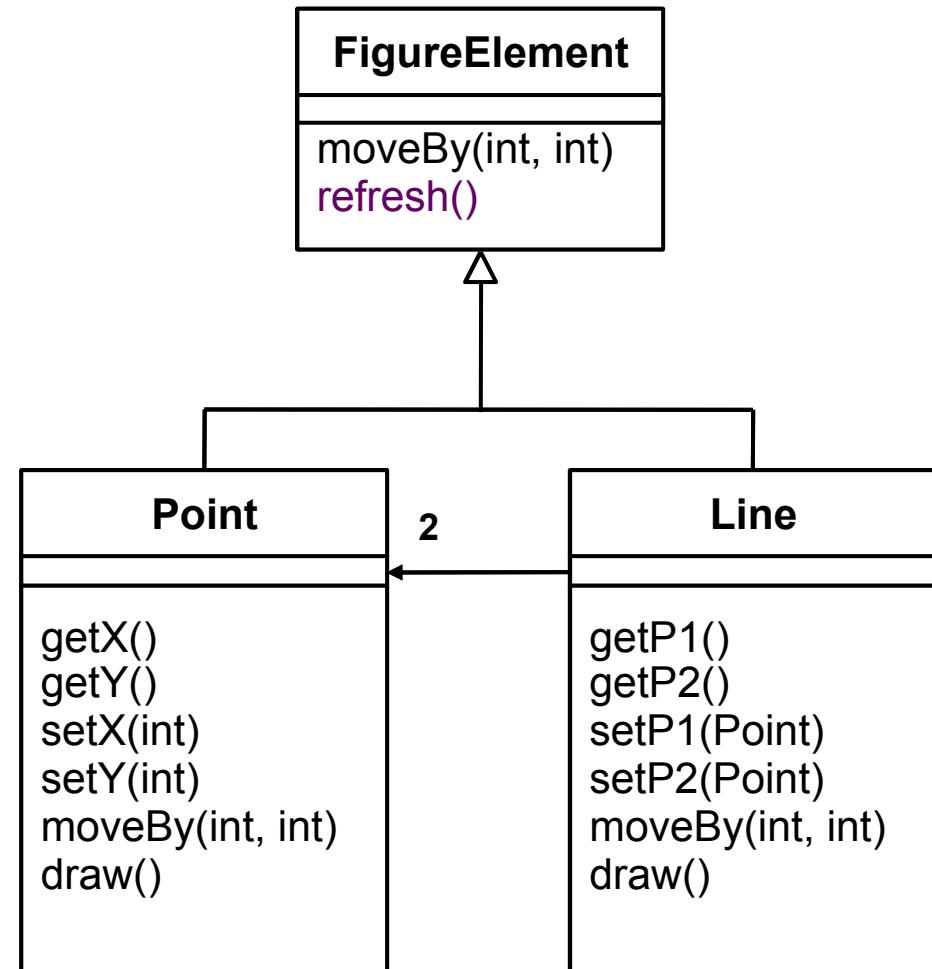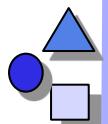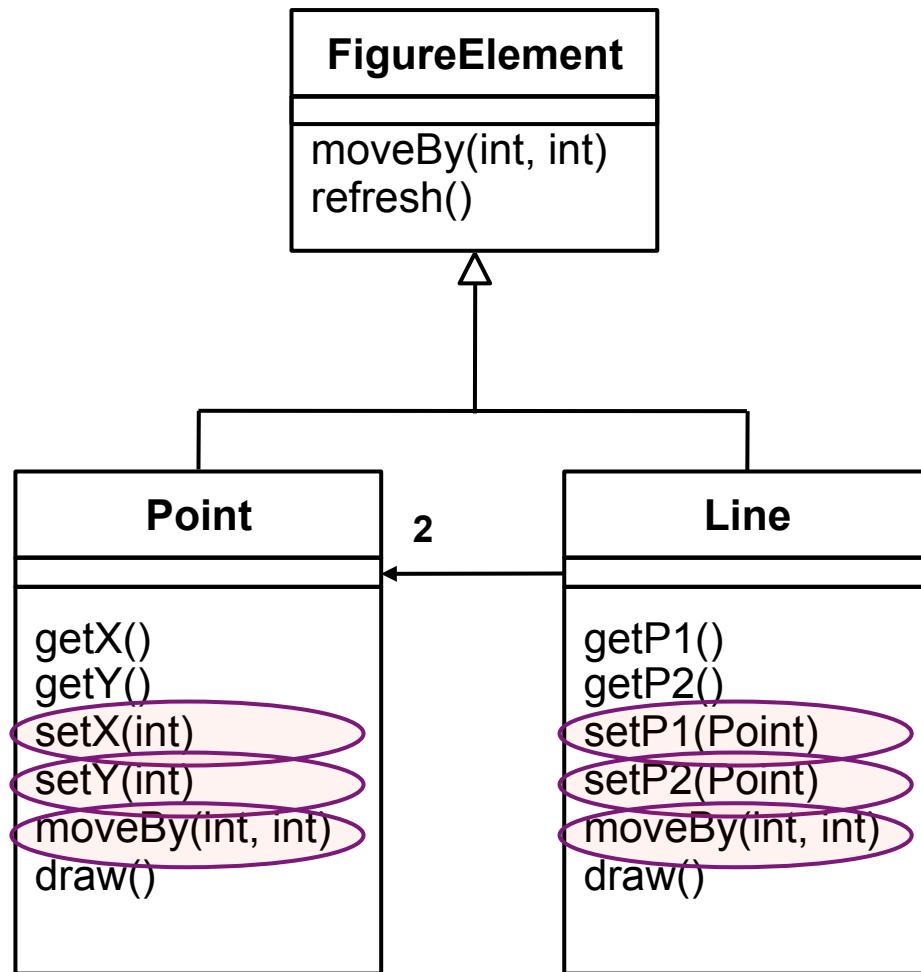
All implementations of
these methods end with
call to:

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

Needs AOP for a Solution

FigureElement

---

moveBy(int, int)
refresh()

Aspect: a separate hyperslices with concern mapping (crosscut relation)

```
after(): call(void FigureElement+.set*(..))
        || call(void FigureElement.moveBy(int, int))
{
  Display.update();
}
```

Point   **Line**   2

---   ---

getX()   getP1()
getY()   getP2()
setX(int)   setP1(Point)
setY(int)   setP2(Point)
moveBy(int, int)   moveBy(int, int)
draw()   draw()

DisplayUpdating

```
after(): call(void FigureElement+.set*(..))
        || call(void FigureElement.moveBy
(int, int))
{
  Display.update();
}
```

24.2 Aspect-Oriented Programming

The AOP Idea

► *Crosscutting* is inherent in complex systems

- The "tyranny of the dominant decomposition"

- Crosscutting concerns have a clear purpose   What

- have some regular interaction points   Where

► AOP proposes to capture crosscutting concerns explicitly...

- in a modular way

- with programming language support

- and with tool support

► AOP improves View-Based Programming

- Beyond name merging (open definitions), *cross-cuts* (cross-cutting relationships) can be defined such that many definitions are extended by an extension

- AOP also relies on open definitions. A core program is open in any of its join points. Join points specify the "points of extension"
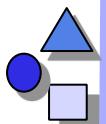
The AOP Idea (2)

- ▸ **Aspects** are separate, independent hyperslices, in which a **crosscutting concern mapping** relates fragment groups (advices) to concerns
- ▸ **Weaving** describes the composition, extending a core program at join points
  - ▸ At software development time, aspects and classes are kept as two, separate dimensions.
  - ▸ At run-time, both dimension need to be combined in some way for obtaining the final product.
 - ▸ Weaving is **non-symmetric composition** (hyperslice composition is symmetric)

Base program (core)

Join points

Weaving

Concern mapping
(pointcuts)
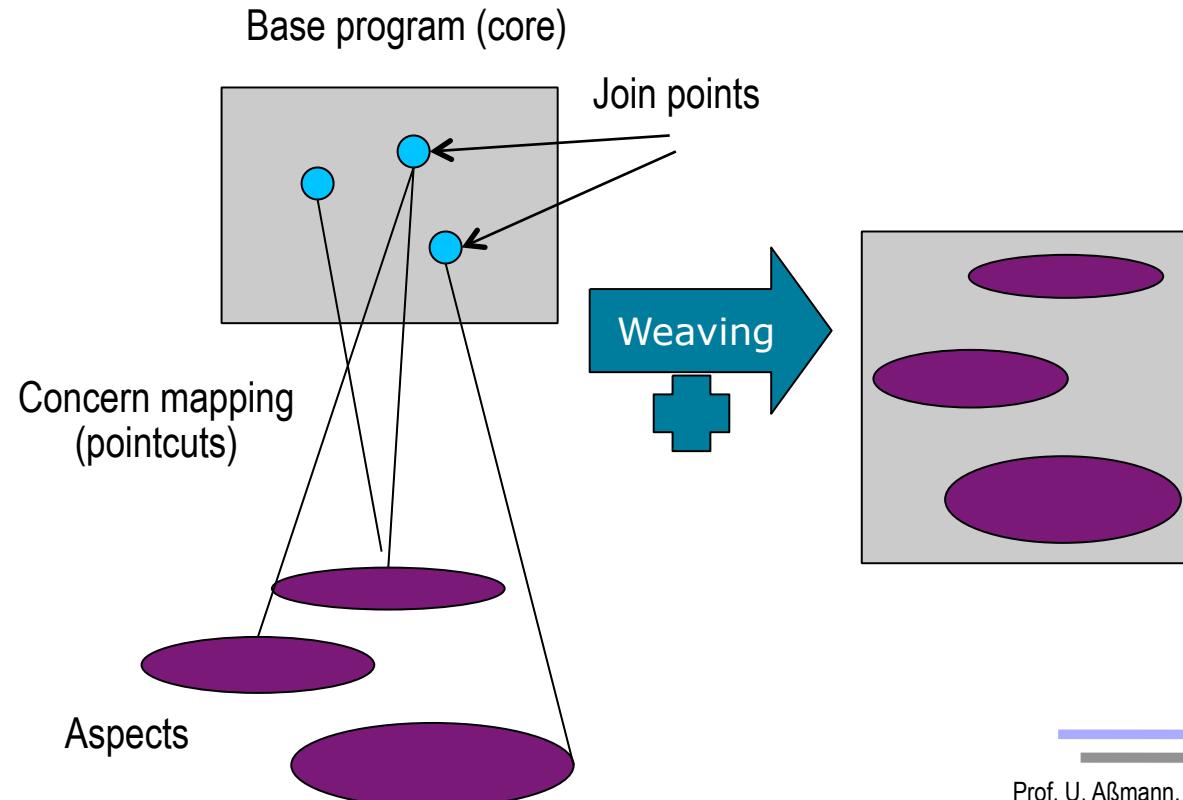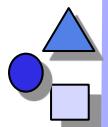
Aspects

AspectJ: a Weaver for Java

► First production-quality AOP-technology

► Allows specifying hyperslices for crosscutting concerns as separate entities: Aspects

- **Static join points** are code positions, hooks, open for extension

- **Dynamic join point**: some point in the execution of an application, open for extension

- **Pointcut:** a set of logically related join points

- **Advice:** a some behavior that should become active whenever a dynamic join point is encountered

- **Weaving:** a technology for bringing aspects and base code together

```
// aspects are hyperslices plus integrated concern mapping
aspect  <concern> {
   // introductions: fragments added to classes of the core
   // advices: fragments for extensions
   // pointcuts: concern mapping from advices to
   //     joinpoints of the core
}
```

Example: A Simple Figure Editor

Display

factory methods

Figure
—
makePoint(..)
makeLine(..)

*

FigureElement
—
moveBy(int, int)

Point
—
getX()
getY()
setX(int)
setY(int)
moveBy(int, int)

2

Line
—
getP1()
getP2()
setP1(Point)
setP2(Point)
moveBy(int, int)

operations that
move elements
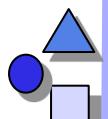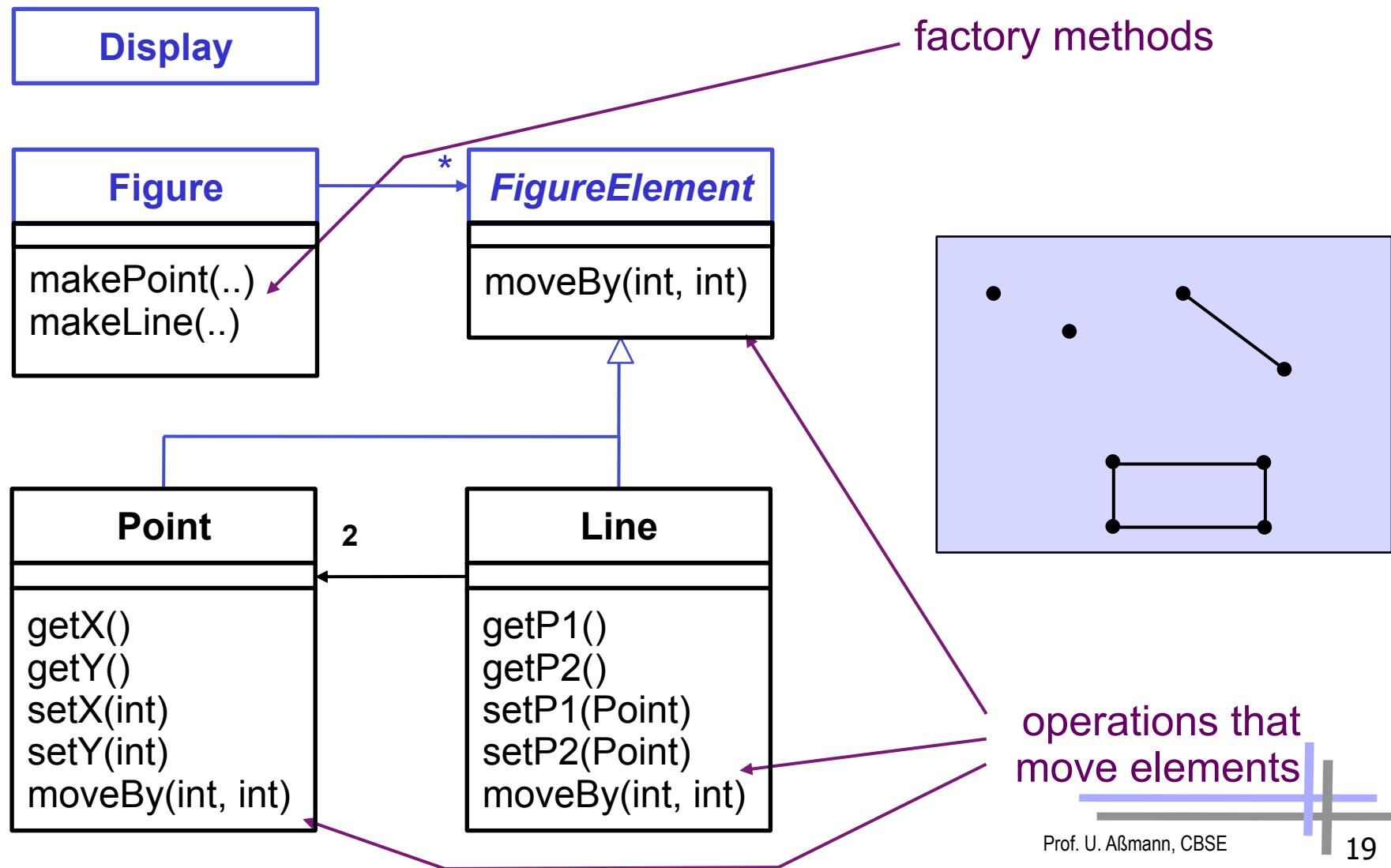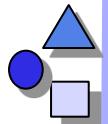
Example: A Simple Figure Editor

```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) { ... }
}
```
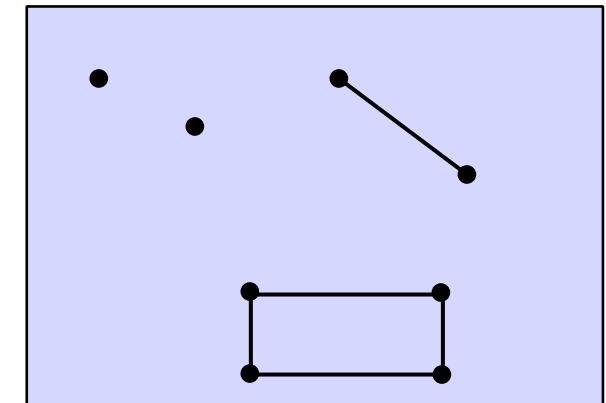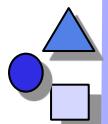
Display Updating

► Collection of figure elements

  ▪ that move periodically

 ▪ must refresh the display as needed
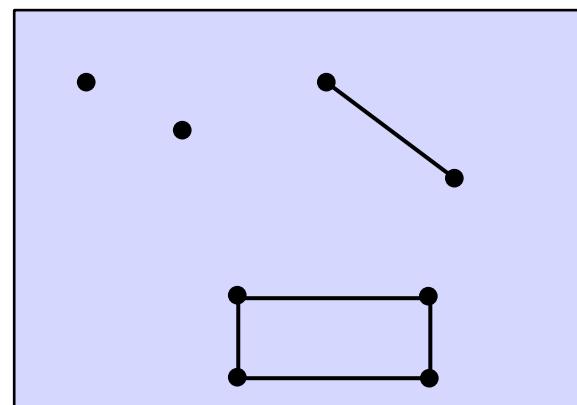
*we will initially assume
just a single display*
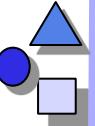
Aspect/J Dynamic Join Points (Dynamic Hooks)

▶ A *dynamic join point* is a key point in dynamic call graph

`line.moveBy(2, 2)`

a Line

dispatch

a method execution
returning or throwing

a method call
returning or
throwing

a Point

dispatch

a method execution
returning or throwing
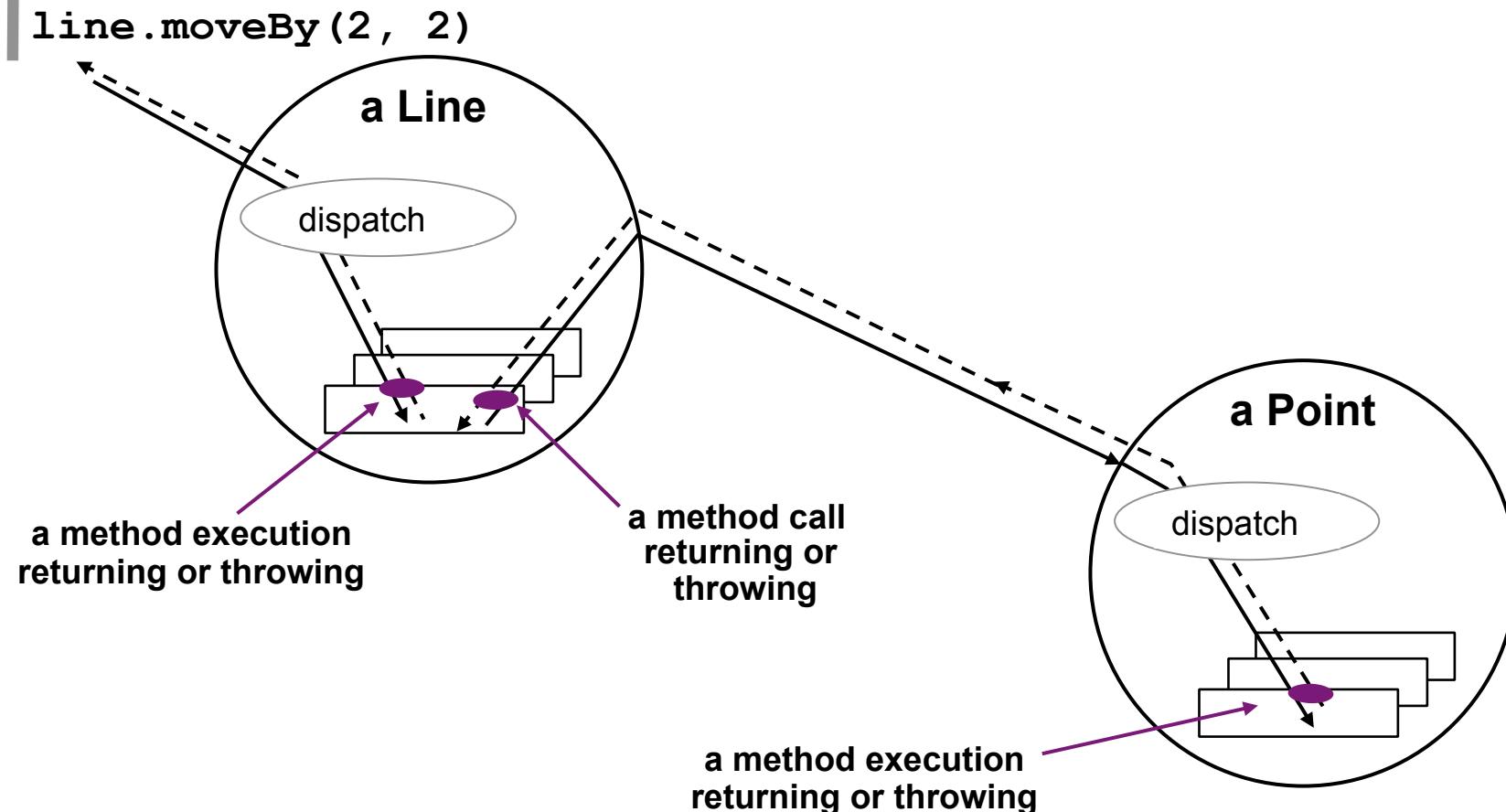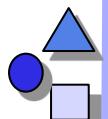
Dynamic Join Point Terminology

a Line

dispatch

**Method body execution join points**

method call join points
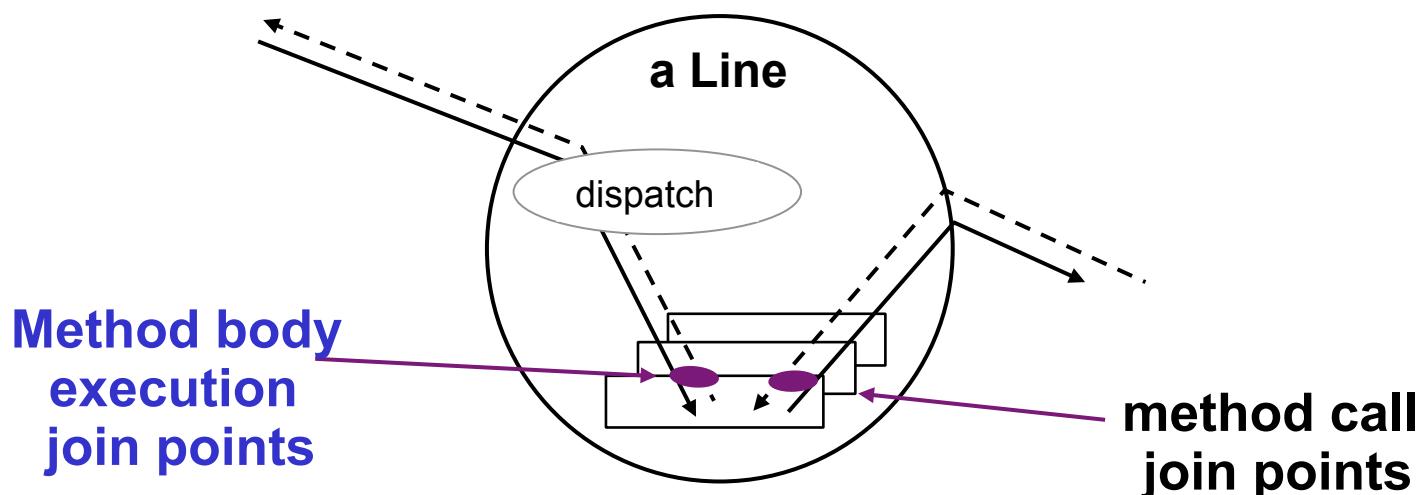
▶ several kinds of join points

- method & constructor call
- method & constructor execution
- field get & set
- exception handler execution
- static & dynamic initialization
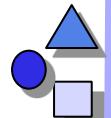
Join Point Terminology

```
line.moveBy(2, 2)
```

a Point

a Line

a Point

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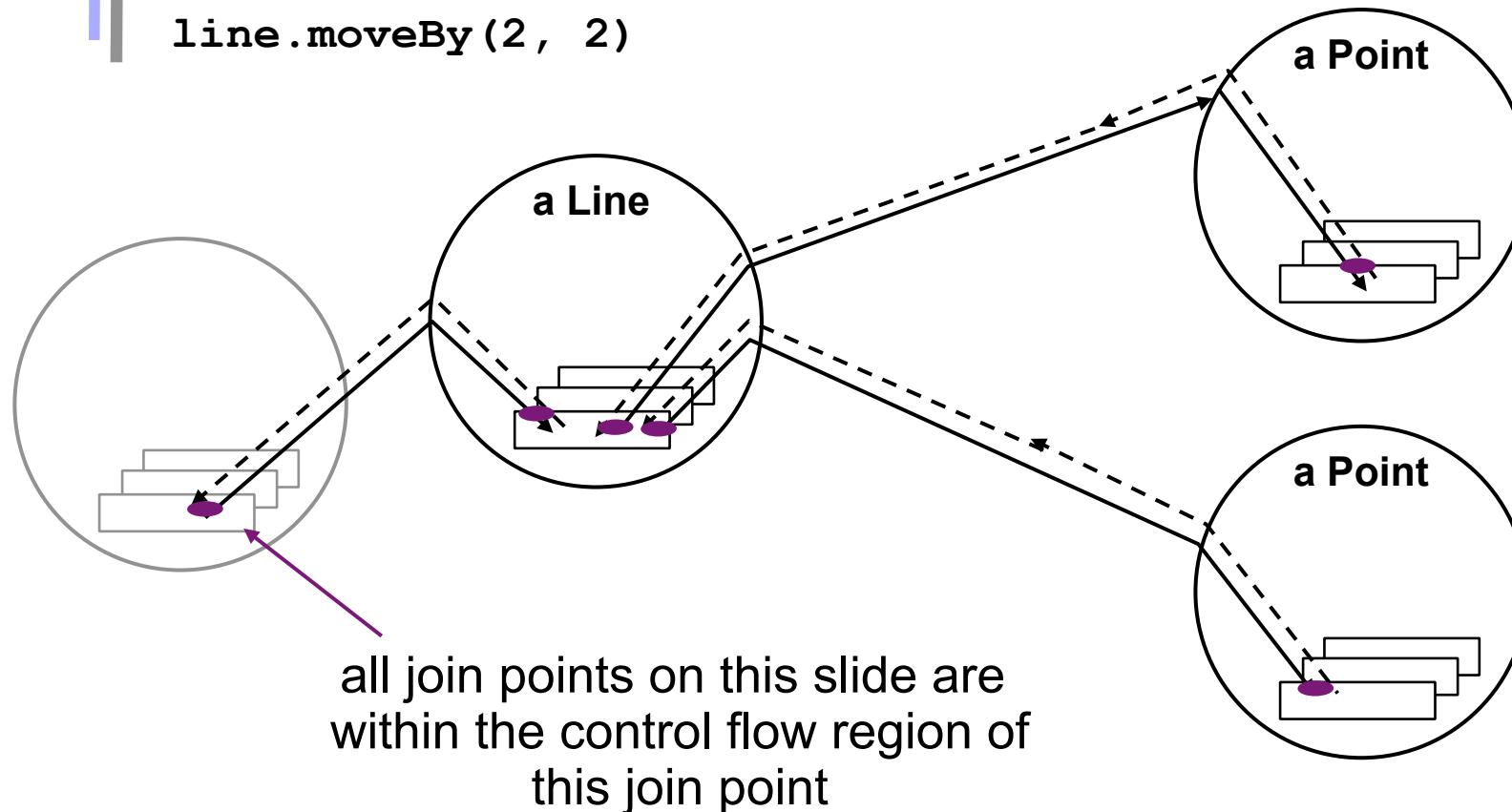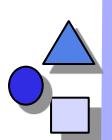
Primitive Pointcuts

► A **pointcut** is an addressing expression 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

- "a means of identifying join points"

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

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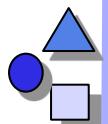
Pointcut Composition

► Pointcuts 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
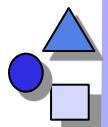
User-Defined Pointcuts

► User-defined (named) pointcuts

- defined using the pointcut construct

- 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
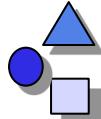
After Advice

► An *after advice* is the action to take after computation under join points

after advice runs
"on the way back out"

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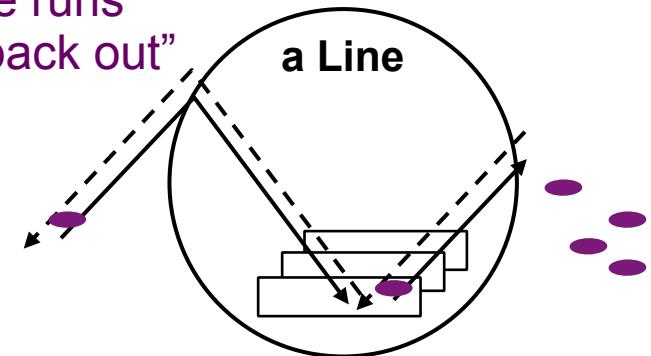
```
pointcut move():
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point));

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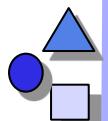
A Simple Aspect

an aspect defines a special class that can crosscut other classes

```
aspect DisplayUpdating {

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

  after() returning: move() {
    Display.update();
  }
}
```
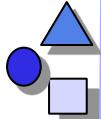
Without AspectJ

```java
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();
  }
}
```

► what you would expect

- update calls are tangled through the code
- "what is going on" is less explicit
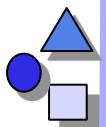
Pointcuts

► can cut across multiple classes

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

► Can use interface signatures

```
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));
```
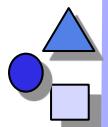
A multi-class aspect

```
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();
  }
}
```
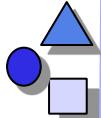
Using values at join points

▶ A pointcut can explicitly expose certain values

▶ An advice can use the exposed value

parameter
mechanism
being 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>
}
```
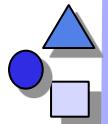
Parameters of user-defined pointcut designator

► Variable is bound by user-defined pointcut declaration

- Pointcut supplies value for variable
- Value is available to all users of user-defined pointcut

pointcut parameters

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

typed variable in place of type name

```
after(Line line): move(line) {
  <line is bound to the line>

}
```

Parameters of advice

► Variable is bound by advice declaration

- Pointcut supplies value for variable

- Value is available in advice body

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

advice parameters

typed variable in place
of type name

```
after(Line line): move(line) {
  <line is bound to the line>
}
```
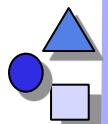
Explaining parameters…

► 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

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



after(Line line): move(line) {
  <line is bound to the line>

}
```
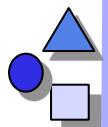
Join Point Qualifier "Target"

A **join point qualifier** does two things:

- exposes information from the context of the join point (e.g, target object of a message)
- tests a predicate on join points (e.g., a dynamic type test - any join point at which 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
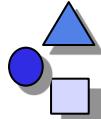
Getting target object in a polymorphic pointcut

`target(<supertype name>) &&`

▶ does not further restrict the join points

▶ does pick up the target object

```
pointcut move(FigureElement figElt):
  target(figElt) &&
  (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) {
  <fe is bound to the figure element>
}
```
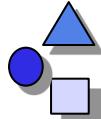
Context & multiple classes

```
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);
  }
}
```
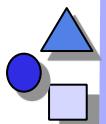
Without AspectJ

```java
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);
  }
}
```

► no locus of "display updating"

- evolution is cumbersome

- changes in all classes

- have to track & change all callers
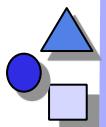
With AspectJ

```java
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;
  }
}
```

```java
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);
  }
}
```

► clear display updating module

- all changes in single aspect
- evolution is modular
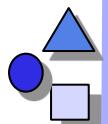
Aspects Crosscut Classes

Display

aspect modularity cuts across
class modularity

Figure   *   *FigureElement*

Figure
makePoint(..)
makeLine(..)

FigureElement
moveBy(int, int)

Point
getX()
getY()
setX(int)
setY(int)
moveBy(int, int)

2

Line
getP1()
getP2()
setP1(Point)
setP2(Point)
moveBy(int, int)
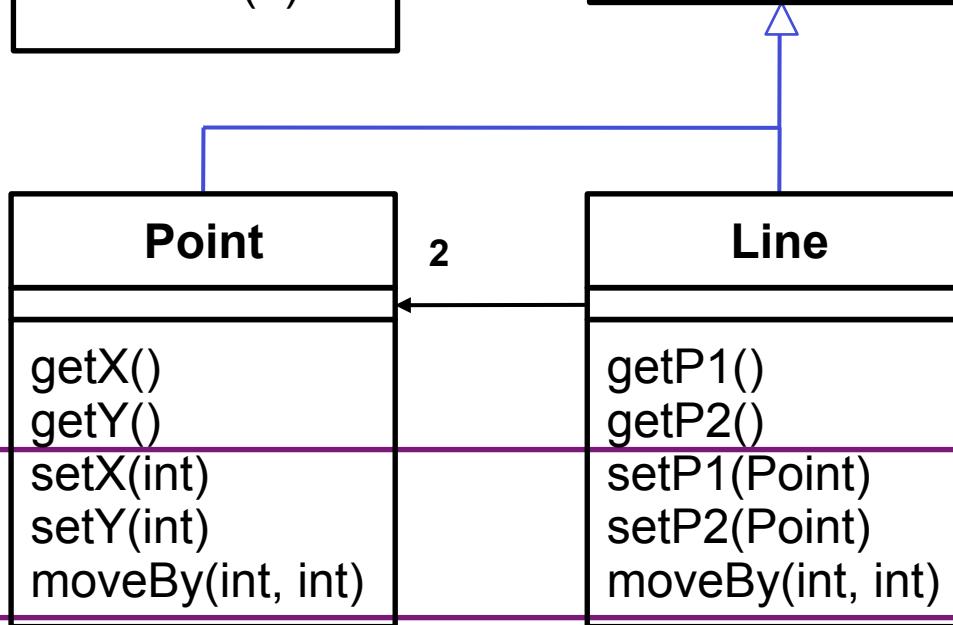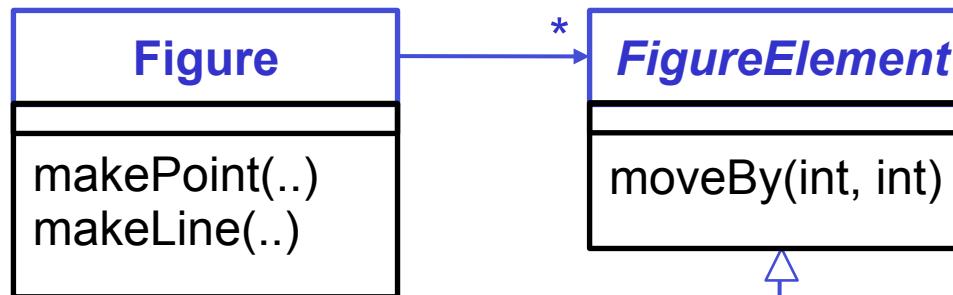
**Aspect
DisplayUpdating**

24.3 Composition Operators and Point-Cuts
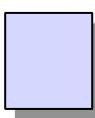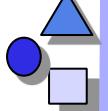
Types of Advice Composition Operators

- ► before   before proceeding at join point

- ► after returning   a value to join point

- ► after throwing   a throwable 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
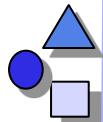
Example: Contract checking with Aspects

► Simple application of before/after/around composition operators

► pre-conditions (assumptions)

  ▪ check whether parameter is valid

► post-conditions (guarantees)

  ▪ check whether values were set

► Invariants

  ► Check conditions that should be true everywhere

► condition enforcement

  ▪ force parameters to be valid and consistent
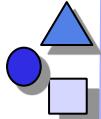
Pre-Condition (Assumption)

```
aspect PointBoundsPreCondition {

  before(int newX):
      call(void Point.setX(int)) && args(newX) {
    assert(newX >= MIN_X);
    assert(newX <= MAX_X);
  }
  before(int newY):
      call(void Point.setY(int)) && args(newY) {
    assert(newY >= MIN_Y);
    assert(newY <= MAX_Y);
  }

  private void assert(boolean v) {
    if ( !v )
      throw new RuntimeException();
  }
}
```

what follows the ':' is always a pointcut – primitive or user-defined
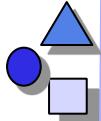
Post-condition

```
aspect PointBoundsPostCondition {

  after(Point p, int newX) returning:
      call(void Point.setX(int)) && target(p) && args(newX) {
    assert(p.getX() == newX);
  }


  after(Point p, int newY) returning:
      call(void Point.setY(int)) && target(p) && args(newY) {
    assert(p.getY() == newY);
  }

  private void assert(boolean v) {
    if ( !v )
      throw new RuntimeException();
  }
}
```
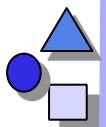
Condition enforcement

```
aspect PointBoundsEnforcement {
  void around(int newX):
      call(void Point.setX(int)) && args(newX) {
    proceed(// before the join point
            clip(newX, MIN_X, MAX_X)
          );
    // after the join point
    System.out.println("after");
  }
  void around(int newY):
      call(void Point.setY(int)) && args(newY) {
    proceed(clip(newY, MIN_Y, MAX_Y));
  }
  private int clip(int val, int min, int max) {
    return Math.max(min, Math.min(max, val));
  }
}
```
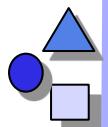
Special Methods

- ► 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 around advice, meaning *"run what would have run if this around advice had not been defined"*
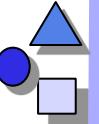
Property-based crosscutting ("Listener Aspects")

```
package
com.xerox.print;
public class C1 {
  …
  public void foo()
    A.doSomething(…
    …
  }
  …
}
```

```
package
com.xerox.scan;
public class C2 {
  …
  public int frotz(
    A.doSomething(…
    …
  }
  public int bar()
    A.doSomething(…
    …
  }
  …
}
```

```
package
com.xerox.copy;
public class C3 {
  …
  public String s1() {
    A.doSomething(…);
    …
  }
  …
}
```

► crosscuts of methods with a common property

  ▪ public/private, return a certain value, in a particular package

► logging, debugging, profiling

  ▪ log on entry to every public method
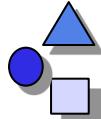
Property-based crosscutting

```
aspect PublicErrorLogging {

  Log log = new Log();

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

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

neatly captures public interface of mypackage

- ► consider code maintenance

- ► another programmer adds a public method
  - . i.e. extends public interface – this code will still work

- ► another programmer reads this code
  - . "what's really going on" is explicit
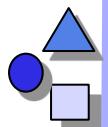
Wildcarding in pointcuts

"*" 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
```
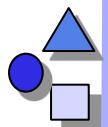
Other Primitive Pointcuts

`this(<type name>)`
any join point at which currently executing object is an instance of type name

`within(<type name>)`
any join point at which currently executing code is contained within type name

`withincode(<method/constructor signature>)`
any join point at which currently executing code is specified method or constructor

`get(int Point.x)`
`set(int Point.x)`

field reference or assignment join points
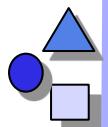
Other Primitive Pointcuts

`execution(void Point.setX(int))`

method/constructor execution join points (actual running method)

`initialization(Point)`

object initialization join points

`staticinitialization(Point)`

class initialization join points (as the class is loaded)

`cflow(`*`pointcut designator`*`)`

all join points within the dynamic control flow of any join point in pointcut designator

`cflowbelow(`*`pointcut designator`*`)`

all join points within the dynamic control flow below any join point in pointcut designator, excluding thisJoinPoint
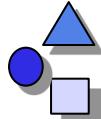
Example: Only top-level moves

```
aspect DisplayUpdating {

  pointcut move(FigureElement fe):
    target(fe) &&
    (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)));

  pointcut topLevelMove(FigureElement fe):
    move(fe) && !cflowbelow(move(FigureElement));

  after(FigureElement fe) returning: topLevelMove(fe) {
    Display.update(fe);
  }
}
```
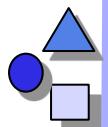
Aspect/J Introductions

- An aspect can 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); }
}
```
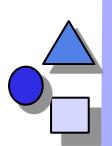
Aspect Weaving

► AspectJ uses compile-time bytecode weaving,

- but also inserts code that matches dynamic join points (dynamic weaving)
- supports weaving aspects to existing *.class files (based on BCEL)
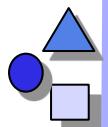
Other approaches (1)

▶ http://www.aosd.net/

▶ 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.

AspectJ is a seamless aspect-oriented extension to Java that enables the modular implementation of a wide range of crosscutting concerns.

AspectWerkz is a dynamic, lightweight and high-performant AOP/AOSD framework for Java.

JAC is a Java framework for aspect-oriented distributed programming.

JBoss-AOP is the Java AOP architecture used for the JBOSS application server.

Nanning is an Aspect Oriented Framework for Java based on dynamic proxies and aspects implemented as ordinary Java-classes.
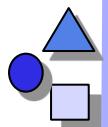
Other approaches (2)

AspectR is aspect-oriented programming for Ruby that allows you to wrap code around existing methods in your classes.

AspectS is an early prototype that enables aspect-oriented programming in the Squeak/Smalltalk environment.

CaesarJ is an aspect-oriented programming language that focusses on multi-view decomposition and aspect reusability.

DemeterJ and DJ facilitate the structure-shy encapsulation of traversal-related behavioral concerns.

Hyper/J supports "multi-dimensional" separation and integration of concerns in standard Java software.

JAsCo is an aspect-oriented programming language tailored for component based software development.

JMangler is a framework for load-time transformation of Java programs, which supports conflict-free composition of independently developed aspects (implemented as JMangler transformer components) and their joint application to existing base classes.

MixJuice is an extension to Java, based on the difference-based module mechanism.

**Mozart Programming System** is an advanced development platform for intelligent, distributed applications.

…
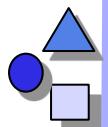
Towards AOSD

- ► Aspects are important in the whole lifecycle
  - ▪ requirements (*early aspects*)
  - ▪ analysis
  - ▪ design (*model aspects*)
  - ▪ implementation (*code aspects*)
  - ▪ test
- ► Aspect-aware technology
- ► Aspect-aware tools
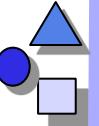
24.4 Evaluation: Aspects as Composition System

Component model

Source components

Greybox components (aspects with advices and introductions)

Composition technique

Simple set of composition operators

Crosscut graph definitions (with pointcuts)

Expression-based

Composition language
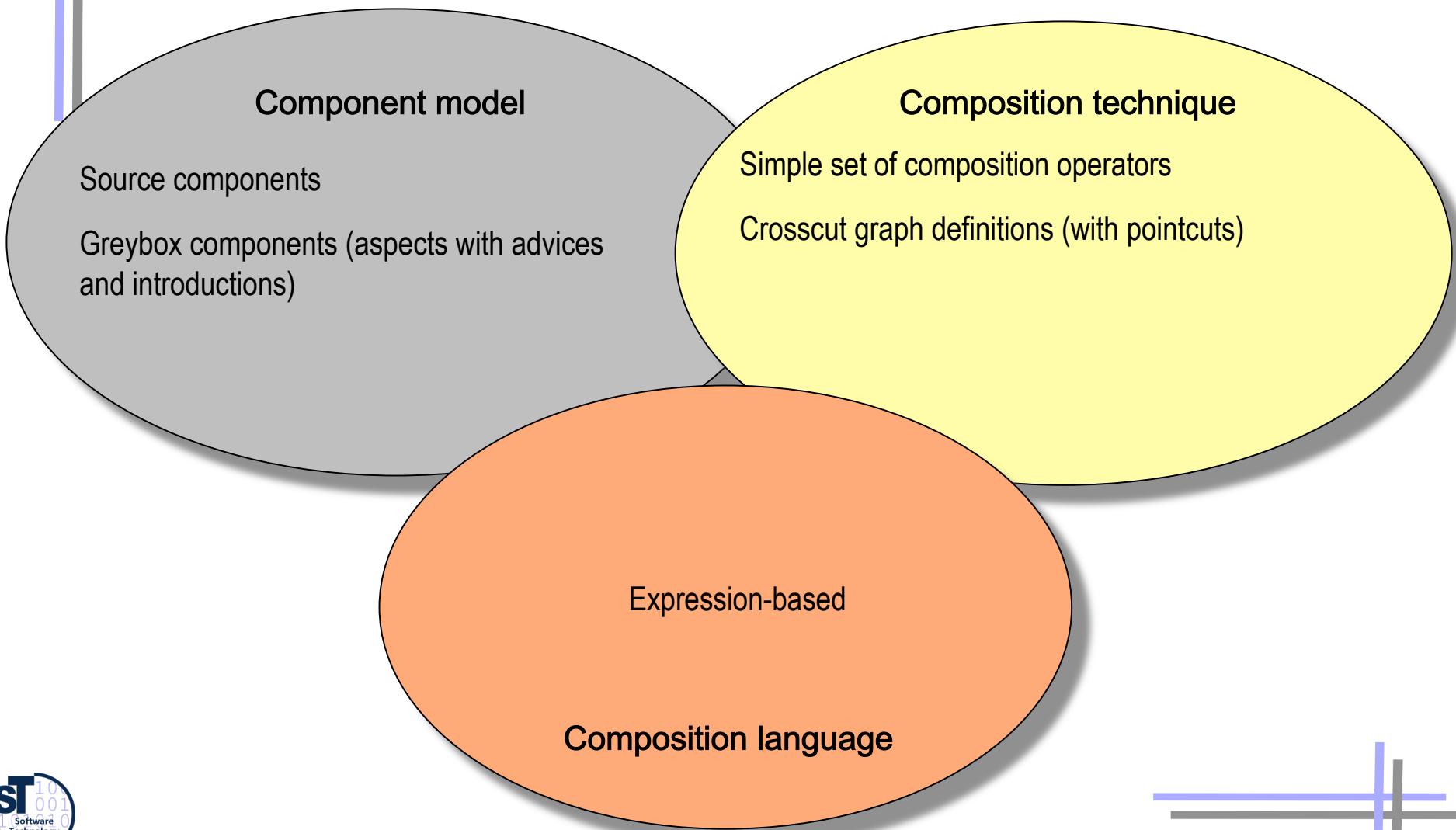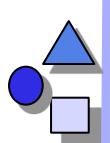
The End

► Slides stem from Wim Vanderperren, Vrije Universitet Brussel, and the Aspect/J team