Aspect-Oriented Programming Radical Research in Modularity

Gregor Kiczales

University of British Columbia
Software Practices Lab

Expressiveness

- The code looks like the design
- “What’s going on” is clear
- The programmer can say what they want to

> ***Programs must be written for people to read, and only incidentally for machines to execute.***
>
> **[SICP, Abelson, Sussman w/Sussman]**

Share An Emerging Debate

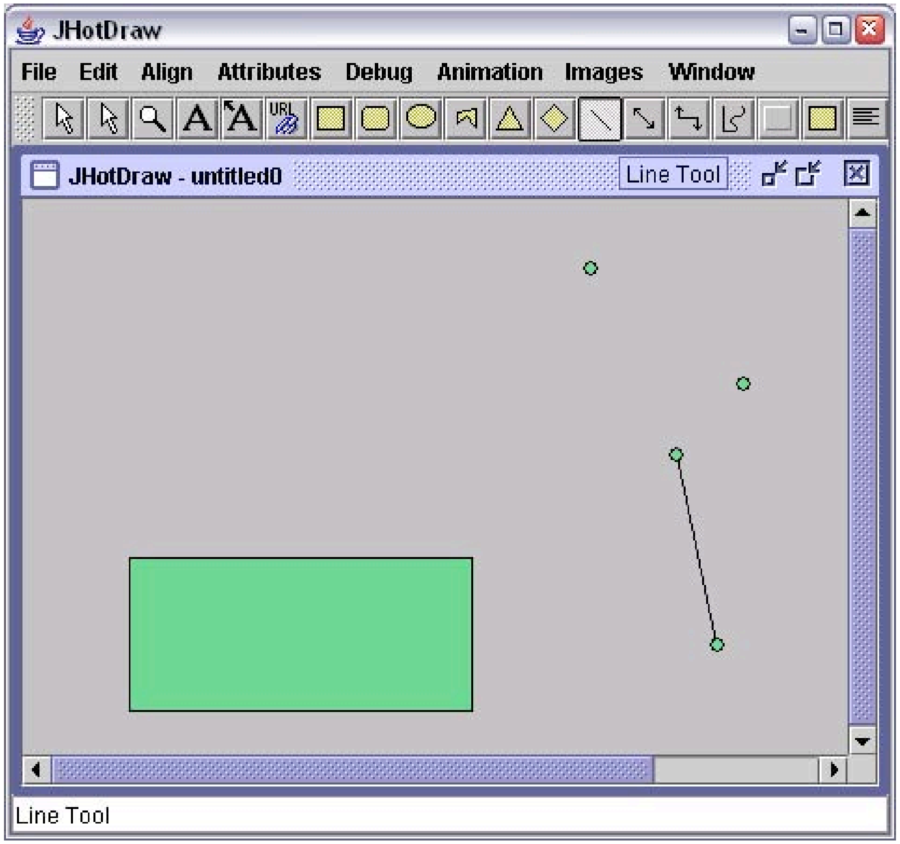
- About modularity and abstraction
  - foundational concepts of the field
  - but perhaps built on invalid implicit assumptions
    - generality of hierarchy
    - dynamicity of software configurations
    - source to machine code correspondence
    - developer's sphere of control

- Consider these definitions:

  A module is a localized unit of source code with a well-defined interface.

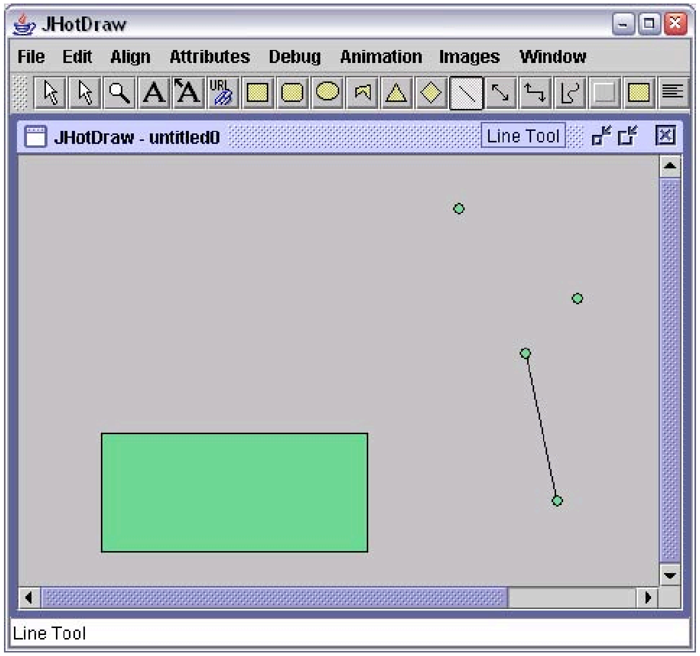
  Abstraction means hiding irrelevant details behind an interface.

Simple Drawing tool (i.e. JHotDraw)



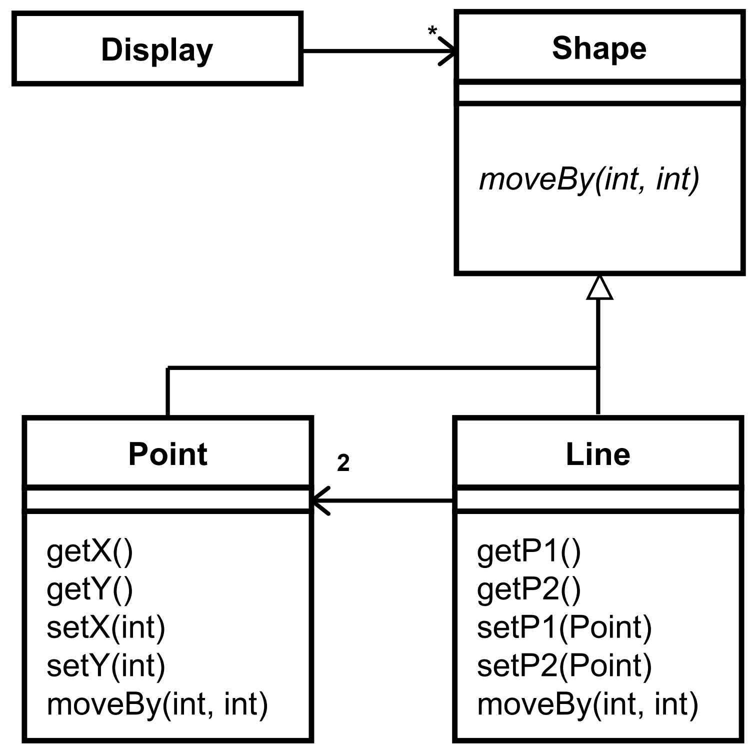
Key Design Elements

- Shapes
  - simple (Point)
  - compound (Line…)
  - display state
  - displayed form
- Display
- …
- Display update signaling
  - when shapes change
  - update display
  - aka Observer Pattern



Using Objects

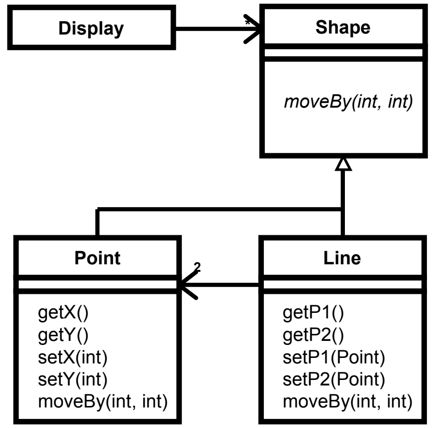
- Shapes
- Display
- Update signaling



Using Objects

- Shapes
- Display
- Update signaling
    - Expressive
        - code looks like the design
        - “what’s going on” is clear
    - Modular
        - localized units
        - well defined interfaces
    - Abstract
        - focus on more or less detail

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

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

  void moveBy(int dx, int dy) {
    x = x + dx; y = y + dy;
  }

  void setX(int x) {
    this.x = x;
  }

  void setY(int y) {
    this.y = y;

  }
}
```

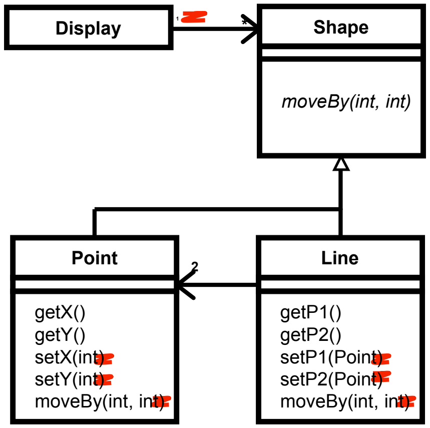


Using Objects

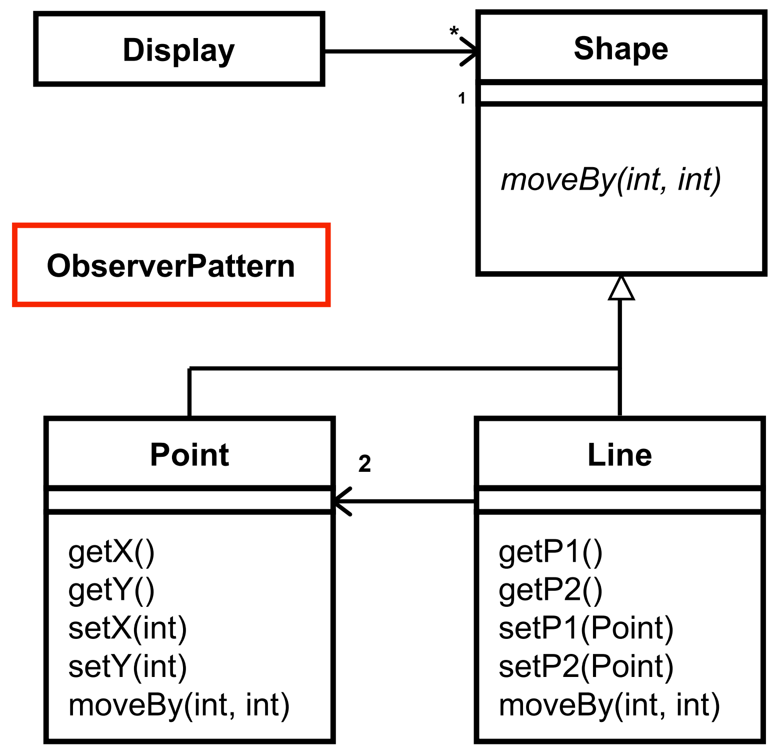
- Shapes
- Display
- Update signaling

  - Expressive
    - Point, Line harder to read
    - structure of signaling
      - not localized, clear, declarative
  - Modular? Abstract?
    - signaling clearly not localized
    - Point, Line polluted
    - revisit this later

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

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

  void moveBy(int dx, int dy) {
    x = x + dx; y = y + dy;
    display.update(this);
  }
  void setX(int x) {
    this.x = x;
    display.update(this);
  }
  void setY(int y) {
    this.y = y;
    display.update(this);
  }
}
```

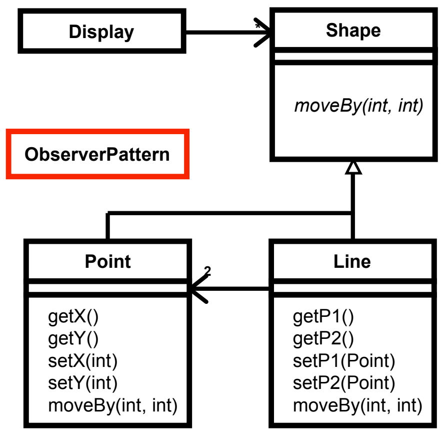


Using Aspect-Oriented Programming



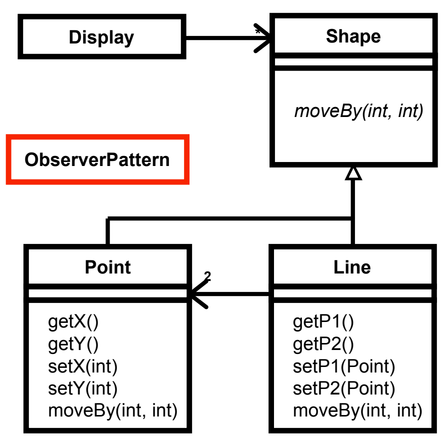
Using Aspect-Oriented Programming

```
aspect UpdateSignaling {

  private Display Shape.display;

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

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



Using Aspect-Oriented Programming

```
aspect UpdateSignaling {

  private Display Shape.display;

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

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

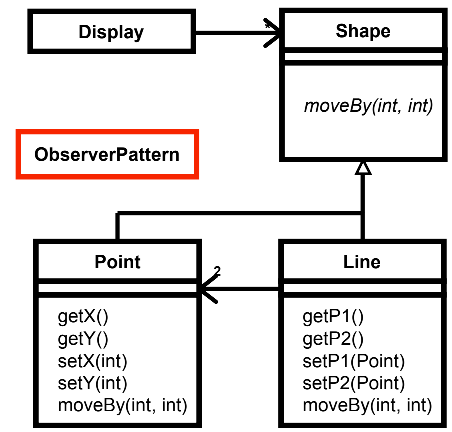


Using Aspect-Oriented Programming

- Shapes
- Display
- Update signaling

```
aspect UpdateSignaling {

 private Display Shape.display;

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

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

  - Expressive
    - “what’s going on” is clear
  - Modular
    - localized units
    - well defined interfaces
  - Abstract
    - focus on more or less detail



Outline

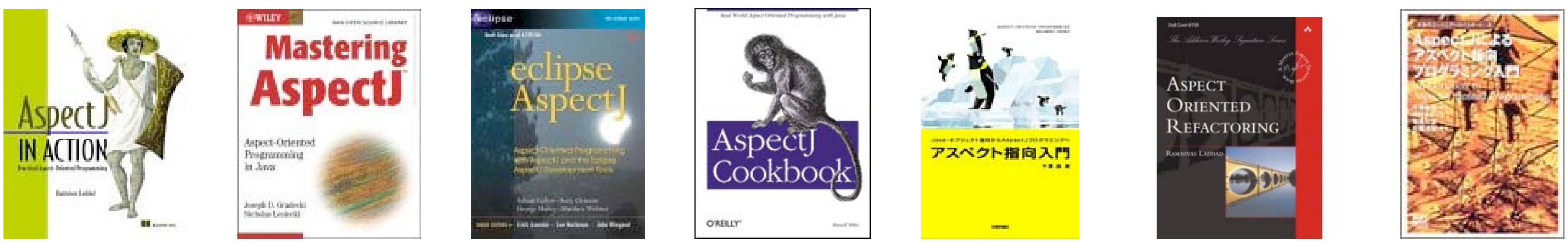
- Introduction
- OOP/AOP Example

- Intro to AOP
- Other Examples
- Is AOP Code Modular, Abstract

- Join Point Models
- Future Possibilities

AOP w/AspectJ

- AspectJ is
  - seamless extension to Java
  - Eclipse open source project
  - de-facto standard on Java platform
  - model for other AOP tools
  - supported by IBM, Interface 21, BEA



MIT Technology Review 10 Leading technologies 2000

2002 World Technology Network Finalist

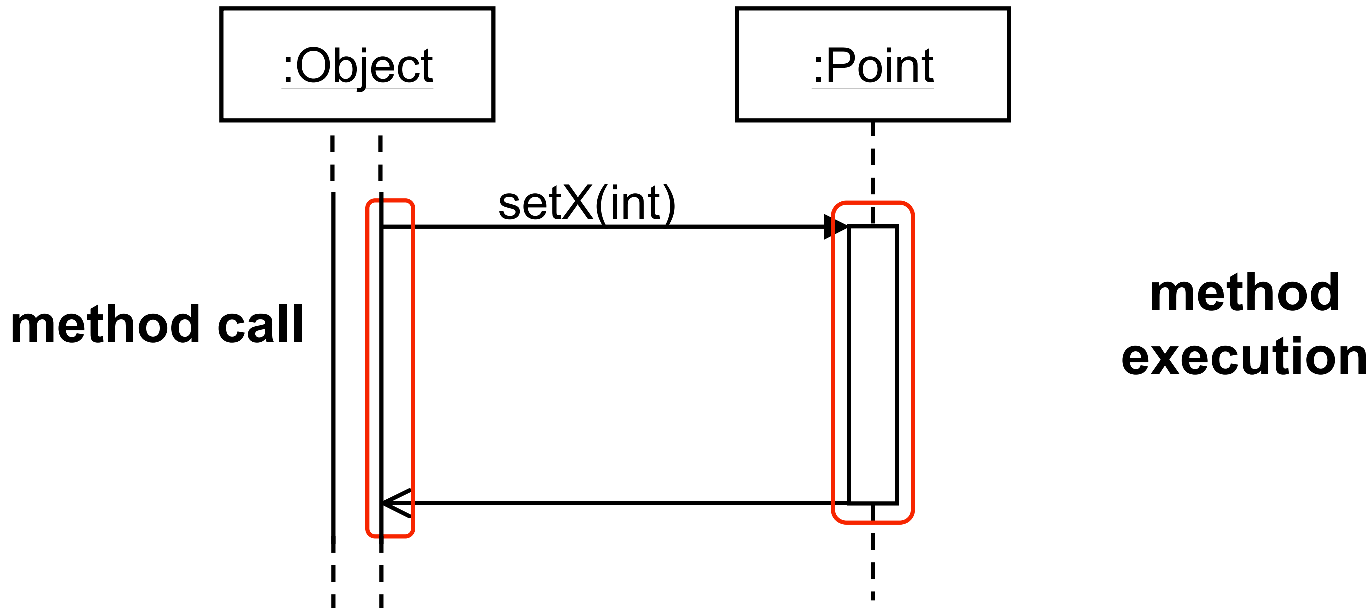
EDITORS' CHOICE JavaWorld FINALIST 2003

MIT Technology Review TR100 2004

software development 14th annual product excellence award JOLT

Dynamic Join Points

points of aspect correspondence



- 11 kinds of dynamic join point
  - well defined points in flow of execution
    - method, constructor, and advice execution
    - method & constructor call
    - field get & set
    - exception handler execution
    - static, object pre- and object initialization

Pointcuts

means of identifying dynamic join points

a pointcut is a predicate on dynamic join points that:

- can match or not match any given join point
- says "what is true" when the pointcut matches
- can optionally expose some of the values at that join point

```
execution(void Line.setP1(Point))
```

matches method execution join points with this signature

Pointcut Composition

pointcuts compose like predicates, using &&, || and !

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

whenever a Line executes a
"void setP1(Point)" or "void setP2(Point)" method

Primitive Pointcuts

- **call, execution, adviceexecution**
- **get, set**
- **handler**
- **initialization, staticinitialization**

kinded

match one kind of DJP
using signature

- **within, withincode**
- **this, target, args**
- **cflow, cflowbelow**

non-kinded

match all kinds of DJP
using variety of properties

User-Defined Pointcuts

user-defined (aka named) pointcuts

- defined with pointcut declaration
- can be used in the same way as primitive pointcuts

name → parameters

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

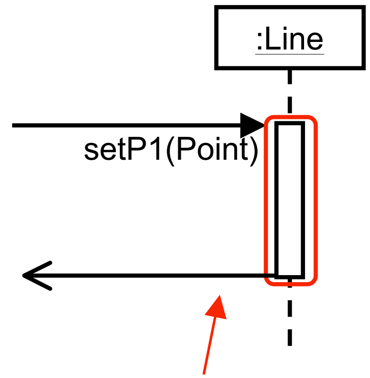
Every powerful language has three mechanisms for [combining simple ideas to form more complex ideas]:

* primitive expressions, which represent the simplest entities the language is concerned with,
* means of combination, by which compound elements are built from simpler ones, and
* means of abstraction, by which compound elements can be named and manipulated as units.

[SICP, Abelson, Sussman w/ Sussman]

After Advice

means of semantic effect at dynamic join points



after advice runs on the way back out

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

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

A Simple Aspect

ObserverPattern v1

```
aspect ObserverPattern {

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

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

box means complete running code

How to Read This Code

ObserverPattern v1

Here is the ObserverPattern aspect of the system.

```
aspect ObserverPattern {

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

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

Some points in the system's execution are a "change".

Specifically, these method executions.

After returning from change points-update the display.

Without AspectJ

ObserverPattern v1

```
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 write if you didn’t have AspectJ;
NOT what AspectJ produces
OR meaning of AspectJ code

- what you would expect
  - update calls are scattered and tangled
  - “what is going on” is less explicit

How Do You Think About Objects?

- Objects
  - Define their own behavior
  - Have fields and methods
  - Clear interface

- A datastructure w/
  - Vector of fields
  - Pointer to method table
- Dispatch code
  - Method call → table entry
- Macrology to
  - Make fields look like vars
  - Method calls look nice

Abstraction

- Objects
  - Define their own behavior
  - Have fields and methods
  - Clear interface

Helps to

- do OO *design*

- scale use of objects to large systems

- A datastructure w/
  - Vector of fields
  - Pointer to method table
- Dispatch code
  - Method call → table entry
- Macrology to
  - Make fields look like vars
  - Method calls look nice

Helps understand

- *one way* to implement OOP

- potential performance costs

- language semantics issues

Abstraction

Helps to

- do AO *design*

- scale use of aspects to large systems

Helps understand

- *one way* to implement AOP

- potential performance costs

- language semantics issues

- Aspects
  - Define their own behavior
  - Have pointcuts, advice …
  - Clear interface
- A datastructure w/
  - Vector of fields
  - Pointer to method table
- Code transformations
  - Find join point shadows
  - Insert interceptor calls

Abstraction

- Objects
  - Define their own behavior
  - Have fields and methods
  - Clear interface
- A datastructure w/
  - Vector of fields
  - Pointer to method table
- Dispatch code
  - Method call → table entry
- Macrology to
  - Make fields look like vars
  - Method calls look nice

- Aspects
  - Define their own behavior
  - Have pointcuts, advice …
  - Clear interface
- A datastructure w/
  - Vector of fields
  - Pointer to method table
- Code transformations
  - Find join point shadows
  - Insert interceptor calls

A Multi-Class Aspect

ObserverPattern v2

```
aspect ObserverPattern {

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

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

Using Naming Convention

ObserverPattern v2b

```
aspect ObserverPattern {

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

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

Using Attributes

ObserverPattern v2c

```
aspect ObserverPattern {

  pointcut change():
    execution(@Change * *(..)));

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

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

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

  @Change
  void moveBy(int dx, int dy) {
    p1.moveBy(dx, dy);
    p2.moveBy(dx, dy);
  }
  @Change
  void setP1(Point p1) {
    this.p1 = p1;
  }
  @Change
  void setP2(Point p2) {
    this.p2 = p2;
  }
}
```

Values at Join Points

ObserverPattern v3

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

```
aspect ObserverPattern {

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

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

Crosscutting Structure

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

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

  void moveBy(int dx, int dy) {
    p1.moveBy(dx, dy);
    p2.moveBy(dx, dy);
  }
  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 moveBy(int dx, int dy) {
    x = x + dx; y = y + dy;
  }
  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

```
aspect ObserverPattern {

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


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

- Aspect and classes crosscut
- Pointcut cuts interface
  - through Point and Line
  - advice programs against interface
  - interface structure is declarative

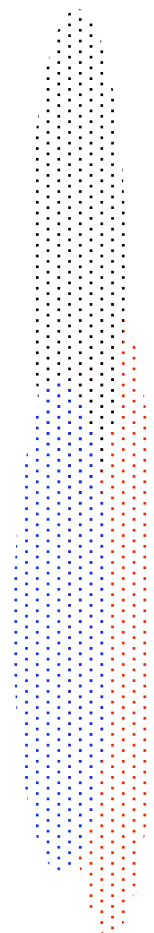
Crosscutting

c1 and c2 crosscut wrt a common representation iff projections overlap, but do not contain [Masuhara, ECOOP03]

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

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

  void
    th
  }
  void
    th
  }
}
```

```
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;
  }
}
```

```
aspect ObserverPattern {

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

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



```
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();
  }
}
```

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

Scattering and Tangling

```
class Shape {
  private Display display;

  abstract void moveBy(int, int);
}
class Line extends Shape {
 private Point p1, p2;

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

  void moveBy(int dx, int dy) {
    p1.moveBy(dx, dy);
    p2.moveBy(dx, dy);
    display.update(this);
  }

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

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

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

  void moveBy(int dx, int dy) {
    x = x + dx;
    y = y + dy;
    display.update(this);
  }

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

Observer pattern is

scattered –
spread around

tangled –
mixed in with other concerns

Only Top-Level Changes

ObserverPattern v4

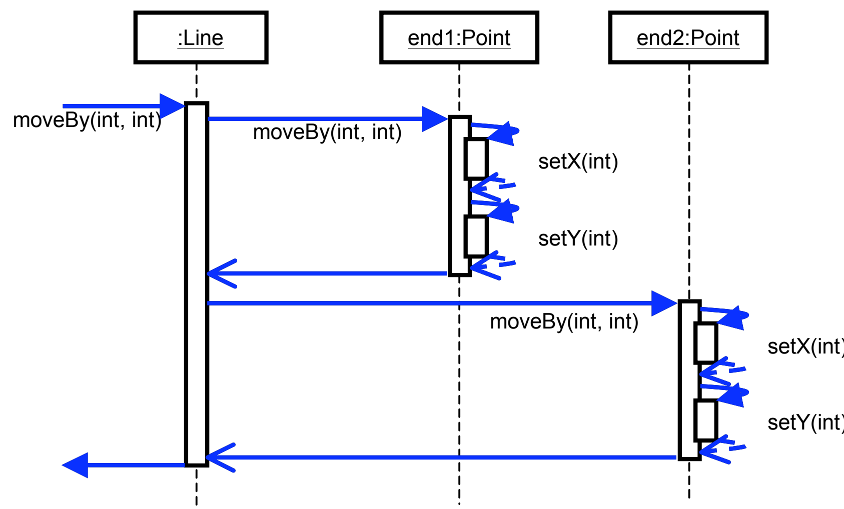
```
aspect ObserverPattern {

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

  pointcut topLevelChange(Shape shape):
    change(shape) && !cflowbelow(change(Shape));

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

Compositional Crosscutting

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

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

  void moveBy(int dx, int dy) {
    p1.moveBy(dx, dy);
    p2.moveBy(dx, dy);
  }
  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 moveBy(int dx, int dy) {
    x = x + dx; y = y + dy;
  }
  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

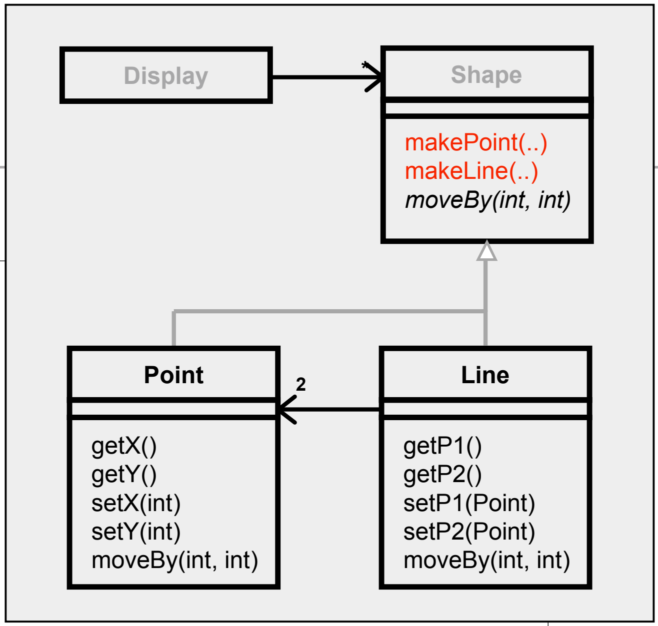
```
aspect ObserverPattern {

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

  pointcut topLevelChange(Shape shape):
    change(shape) && !cflowbelow(change(Shape));

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



Outline

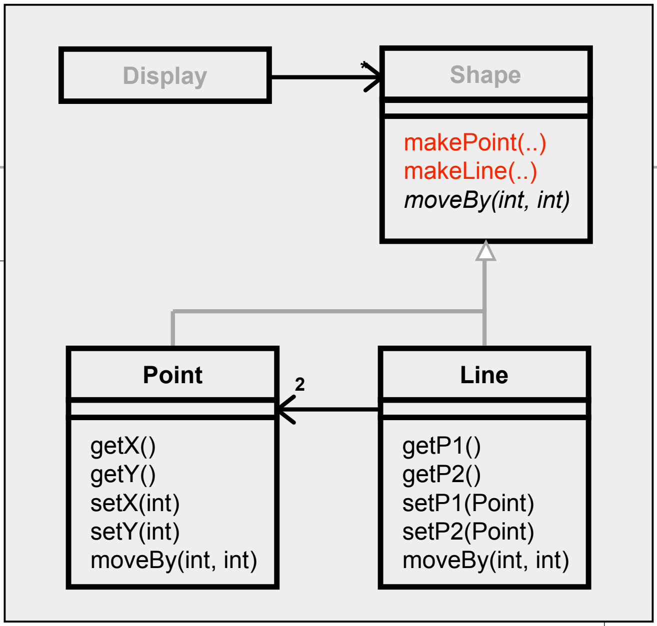
- Introduction
- OOP/AOP Example
- Intro to AOP
- Other Examples
- Is AOP Code Modular, Abstract
- Join Point Models
- Future Possibilities

Design Invariants




```
aspect FactoryEnforcement {

  pointcut newShape():
    call(Shape+.new(..));

  pointcut inFactory():
    withincode(Shape+ Shape.make*(..));

  pointcut illegalNewShape():
    newShape() && !inFactory();


  before(): illegalNewShape() {
    throw new RuntimeError("Must call factory method…");
  }
}
```

Design Invariants




```
aspect FactoryEnforcement {

  pointcut newShape():
    call(Shape+.new(..));

  pointcut inFactory():
    withincode(Shape+ Shape.make*(..));

  pointcut illegalNewShape():
    newShape() && !inFactory();


  declare error: illegalNewShape():
    "Must call factory method to create figure elements.";

}
```

(Simple) Authentication State FSM

```
public aspect AccessibilityFSM {

  private enum State { INIT, AUTHENTICATED, REJECTED };

  private State curr = State.INIT; // global state

  pointcut authenticate(): ...;

  pointcut access(): ...;

  after() returning: authenticate() { curr = State.AUTHENTICATED; }
  after() throwing:  authenticate() { curr = State.REJECTED;      }

  before(): access() {
    if( curr != State.AUTHENTICATED )
      throw new AccessException();
    }
}
```

FFDC [Colyer et. al. AOSD 2004]

```
public aspect FFDC {

  private Log log = <appropriate global log>;

  after() throwing (Error e):
                 execution(* com.ibm..*(..)) {
    log.log(e);
  }
}
```

- Logs every error as soon as its thrown
- Consistent policy makes logs meaningful
- Real FFDC implementations are more complex

From a Spacewar Game

```
class Ship {
  ...
  public void fire() { ... }
  public void rotate(int direction) { ... }
  public void fire() { ... }
  ...
  static aspect EnsureShipIsAlive {

    pointcut helmCommand(Ship ship):
      this(ship) &&
      ( execution(void Ship.rotate(int))      ||
        execution(void Ship.thrust(boolean)) ||
        execution(void Ship.fire()) );

    void around(Ship ship): helmCommand(ship) {
      if ( ship.isAlive() ) {
        proceed(ship);
      }
    }
  }
}
```

One Display per Shape

ObserverPattern v5

```
aspect ObserverPattern {

  private Display Shape.display;

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

 pointcut change(Shape
    this(shape) &&
    (execution(void Sh
     execution(void Sh

  after(Shape s) retur
    s.display.update(s
  }
}
```

private with respect to aspect

- inter-type declarations
- aka open classes [Cannon 78]
- declares members of other types
  - fields, methods
- display field
  - is in objects of type Shape
  - but belongs to ObserverPattern aspect

From a Compiler

```
/**
 * Implements the crosscutting relationships concerning the different kinds of
 * labels that different kinds of statements (and one expr) have.  The declare
 * parents block can be read as table of what ASTs have what labels.
 *
 */
aspect HasLabel {

  private interface Label      {} //enclosing loop's label

WhileStat    implements       TopLabel,                                    DoneLabel;
ForStat      implements       TopLabel, IncrLabel,                         DoneLabel;
BreakStat    implements Label                                                       ;
ContinueStat implements Label                                                       ;
BinaryExpr   implements                           TrueLabel,               DoneLabel;
IfStat       implements                           TrueLabel, FalseLabel, DoneLabel;

  declare parents: IfStat       implements                      TrueLabel, FalseLabel, DoneLabel;


  private String Label.label;
  public  String Label.getLabel() { return label; }
  private void   Label.setLabel(String label) { this.label = label; }

  ...

}
```

- dflow
- remote
- ffdc

Outline

- Introduction
- OOP/AOP Example
- Intro to AOP
- Other Examples
- Is AOP Code Modular, Abstract
- Join Point Models
- Future Possibilities

Is the AOP Code Modular, Abstract?

- Reactionary
- Experientially
- Refers to relations
- Business options
- [Kiczales, Mezini ICSE05]

Is the AOP Code Modular, Abstract?

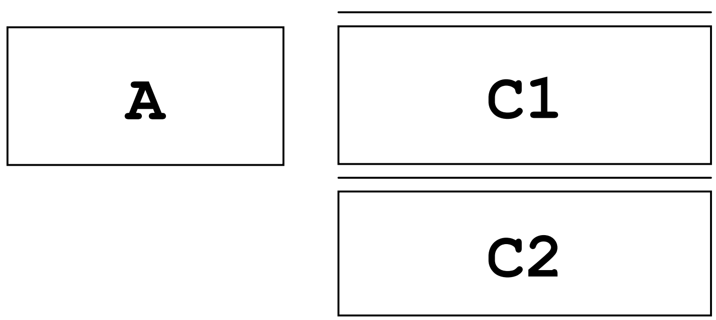
- Remember original definitions

A module is a localized unit of source code with a well-defined interface.

Abstraction means hiding irrelevant details behind an interface.

"AOP is Anti-Modular"

- "it changes the behavior of my code"



- A can affect behavior visible at interface to C1
- But C2 can do that also
- That's the nature of modularity:
  - A module implements its behavior in terms of other well-defined behaviors

The VI Argument

- In non-AOP programmers can easily chase module references
  - to know what has to be consulted
  - to determine complete behavior of C1
  - we don’t want to have to use tool support
- But
  - include files are ‘easy’ to chase down?
  - write enterprise code w/o tools?

…

- Nuance of original definitions

  A module is a ***statically*** *localized unit of source code with a well-defined interface.*

  Abstraction means hiding irrelevant ***for all time*** *details behind an interface.*

- Anti-modular and VI arguments reduce to
  - idea that modularity implies hierarchy
    - designer/implementer/owner of a module has complete responsibility for everything at that level and down
    - implicitly controls all contexts of use

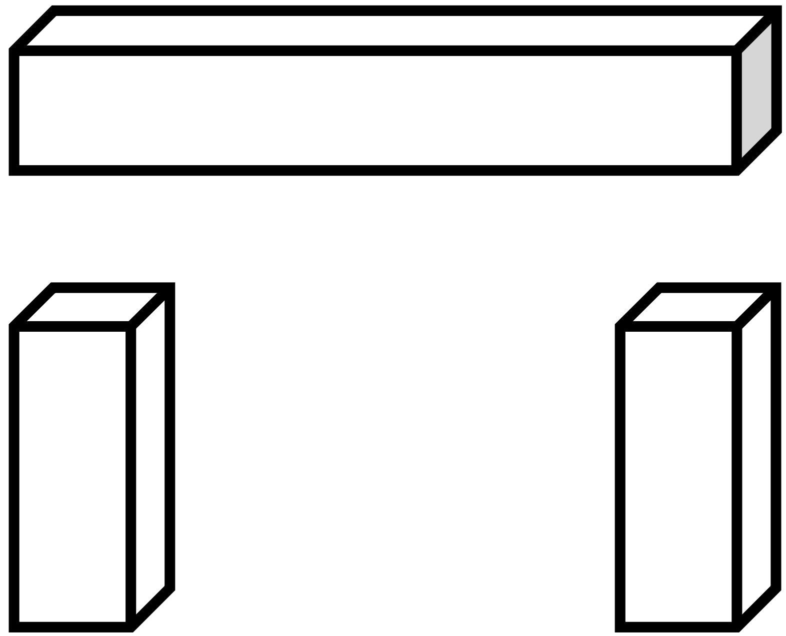
Crosscutting Concerns are Real

- Crosscutting concerns are a fact of life
- Even simple ObserverPattern
  - cannot be implemented modularly w/o AOP

→ hierarchical (de)composition alone isn't enough
→ without AOP, users will scatter code

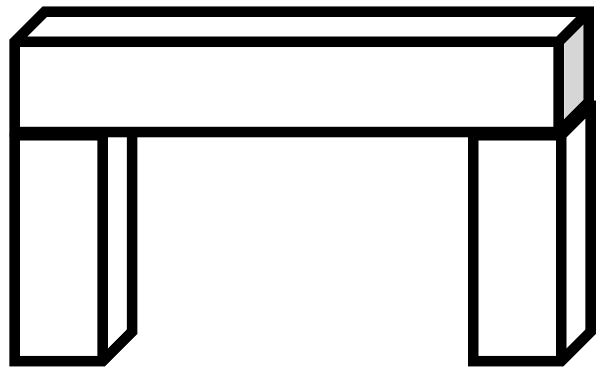
- CVS tells no lies

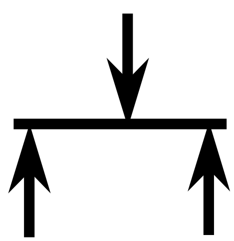
Crosscutting In Other Domains



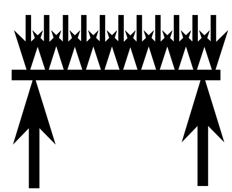
putting 3 blocks together

Crosscutting Models

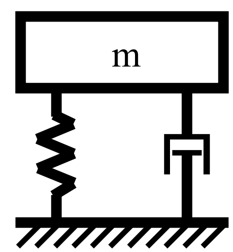




simple
statics



more detailed
statics



simple
dynamics

Without AspectJ

ObserverPattern v5

```
class Shape {
  private Display display;

  abstract void moveBy(int, int);
}
```

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

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

  void moveBy(int dx, int dy) {
    p1.moveBy(dx, dy);
    p2.moveBy(dx, dy);
    display.update(this);
  }

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

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

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

  void moveBy(int dx, int dy) {
    x = x + dx;
    y = y + dy;
    display.update(this);
  }

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

- Replaying the same evolution
- Through 4 versions
- In plain OO (Java)

"display updating" is not modular

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

With AspectJ

ObserverPattern v5

```
class Shape {

  abstract void moveBy(int, int);
}
```

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

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

  void moveBy(int dx, int dy) {
    p1.moveBy(dx, dy);
    p2.moveBy(dx, dy);

  }

  void setP1(Point p1) {
    this.p1 = p1;

  }
  void setP2(Point p2) {
    this.p2 = p2;

  }
}
```

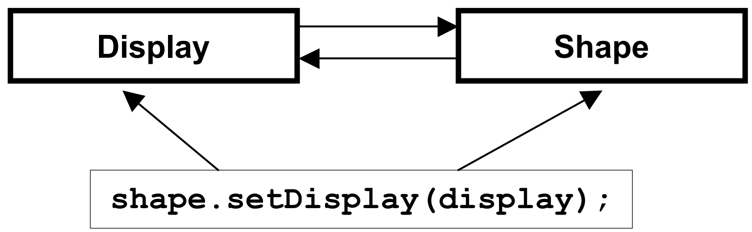
```
class Point extends Shape {
  private int x = 0, y = 0;

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

  void moveBy(int dx, int dy) {
    x = x + dx;
    y = y + dy;

  }

  void setX(int x) {
    this.x = x;

  }
  void setY(int y) {
    this.y = y;

  }
}
```

```
aspect ObserverPattern {

 private Display Shape.display;

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

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

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

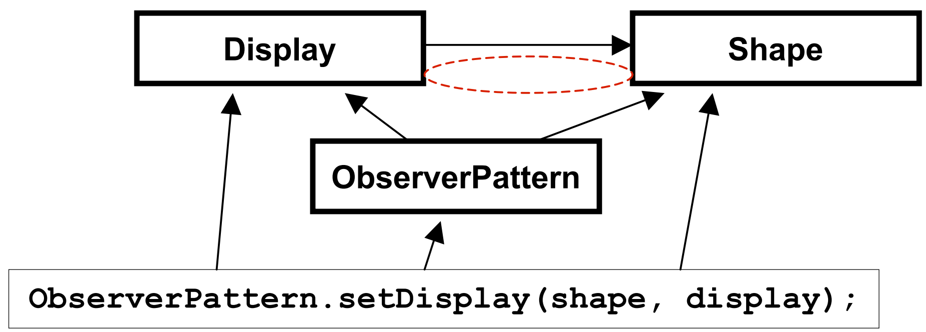
ObserverPattern is modular

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

Comparing *refers to* relations



Plain Java



w/ AspectJ 1

Selling Different Service Aspects

- Major turning point
  - during internal exploration of AspectJ @ IBM
- Product-line potential of
  - FFDC and related serviceability aspects
- “So we could sell different logging policies?”

[Kiczales, Mezini, ICSE 05]

- Starts w/ AspectJ style AOP
- Provides more flexible definition of module
  - modules are statically localized
  - but interfaces are more dynamic
    - constructed based on complete system configuration
- Shows that modular reasoning
  - is possible
  - works better than non AOP if there are crosscutting concerns

IDE support

- AJDT (AspectJ Development Tool)
- An Eclipse Project
- Goal is JDT-quality AspectJ support
  - highlighting, completion, wizards…
  - structure browser
    - immediate
    - outline
    - overview

Outline

- Introduction
- OOP/AOP Example
- Intro to AOP
- Other Examples
- Is AOP Code Modular, Abstract
- Join Point Models
- Future Possibilities

[Smith, On the Origin of Objects[1]]

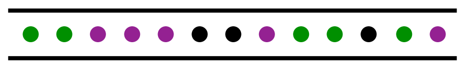
- How is it that we can see the world in different ways?
- Registration is
  - process of ‘parsing’ objects out of fog of undifferentiated stuff
  - constantly registering and re-registering the world
  - mediates different perspectives on a changing world
  - enables moving in and out of connection with the world
- Critical properties of registration
  - multiple routes to reference
    - morning star, evening star
  - ability to exceed causal reach
    - person closest to average height in Gorbachev's office now
  - indexical reference
    - the one in front of him

1. On this slide, object means in the real-world.

Traditional Mechanisms

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

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

  void moveBy(int dx, int dy) {
    p1.moveBy(dx, dy);
    p2.moveBy(dx, dy);
  }
  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 moveBy(int dx, int dy) {
    x = x + dx; y = y + dy;
  }
  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```



stream of instructions

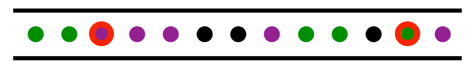
- Modular program structures
- Give rise to execution stream
- Only one place has static direct causal access to given point
  – via single module that gives rise to it
  – equivalent to static hierarchy assumption

Join Point Models

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

```
aspect ObserverPattern {

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

  pointcut topLevelChange(Shape shape):
    change(shape) && !cflowbelow(change(Shape));

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

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

  void moveBy(int dx, int dy) {
    x = x + dx; y = y + dy;
  }
  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```



stream of instructions

- Pointcuts
  - pick out dynamic join points in instruction stream
  - unconstrained by original program modularity
  - ‘register’ instructions in own form
  - create a crosscutting modularity

Join Point Models

- (De)compose software in different ways
- Register aspects out of fog of undifferentiated points
  - means of identifying JPs (aka pointcut) registers
  - aspects/slices/concerns… group over that
- Connect and have effect through that registration
  - means of semantic effect (aka advice)

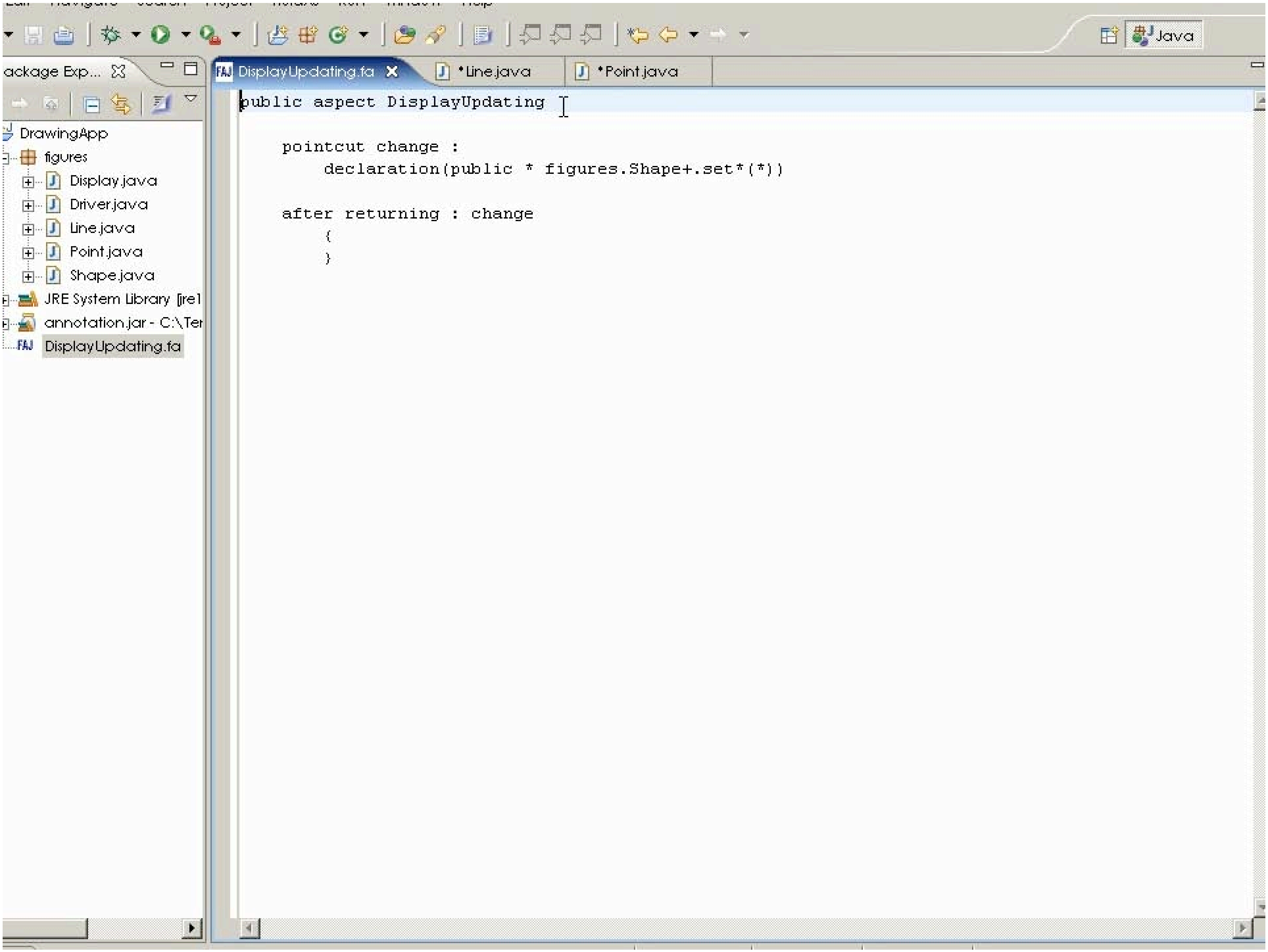
- Critical properties of registration
  - multiple routes to reference
    - void setX(int nx) { … }, call(void setX(int)), cflow(…)
  - exceed causal reach
    - within(com.sun..*), !within(com.mycompany.mysystem)
  - indexical reference
    - cflow(…)

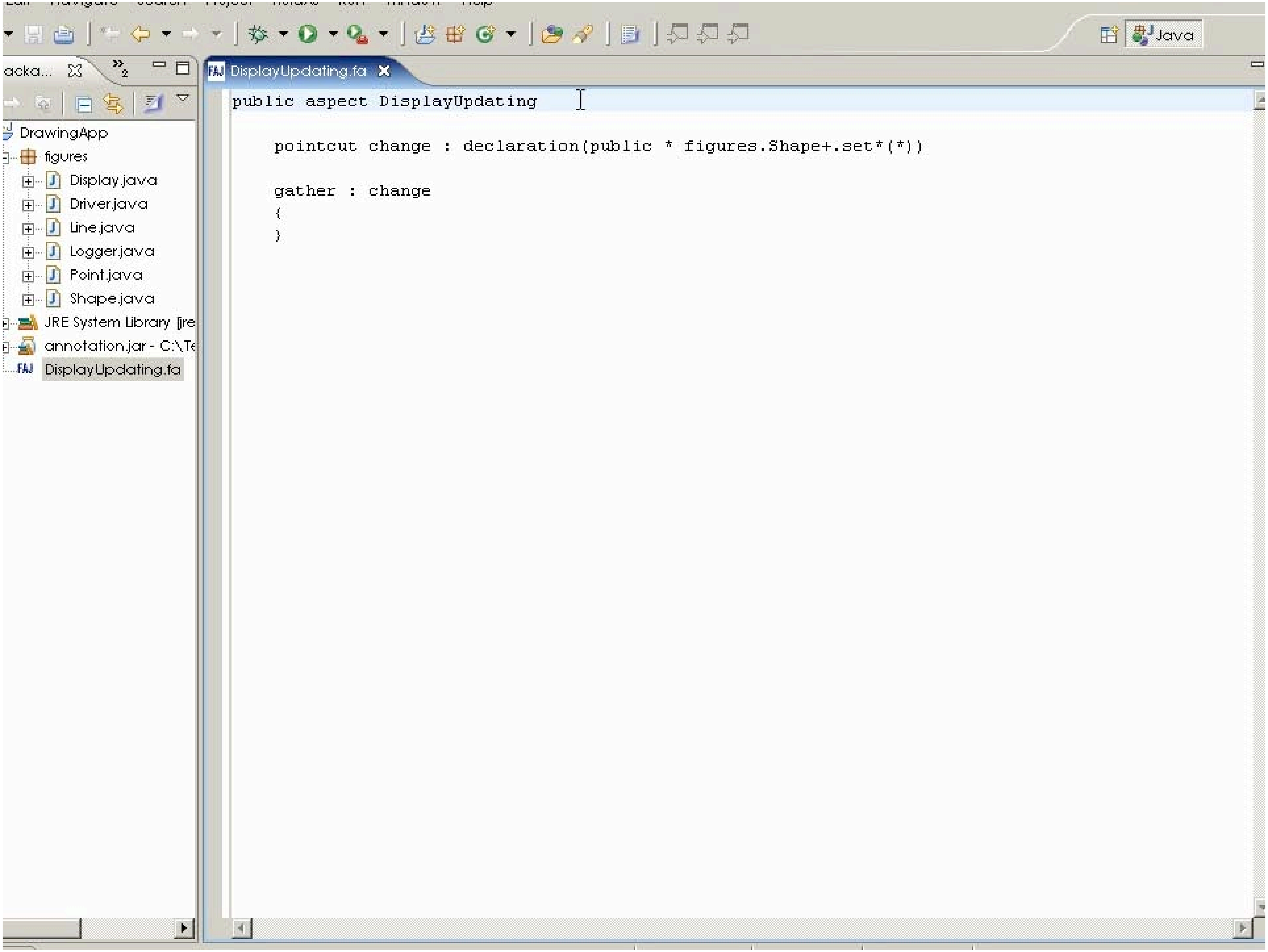
Outline

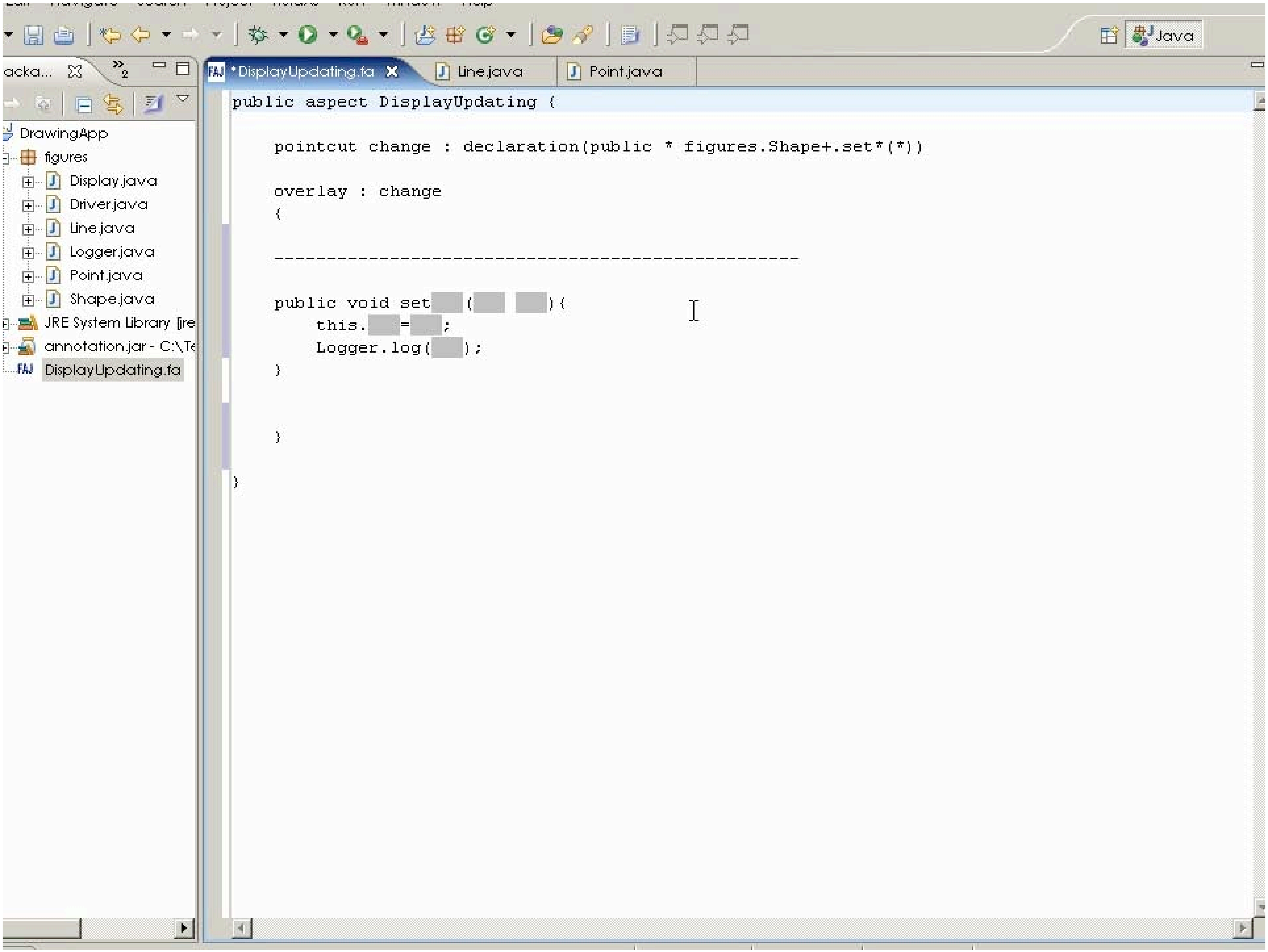
- Introduction
- OOP/AOP Example

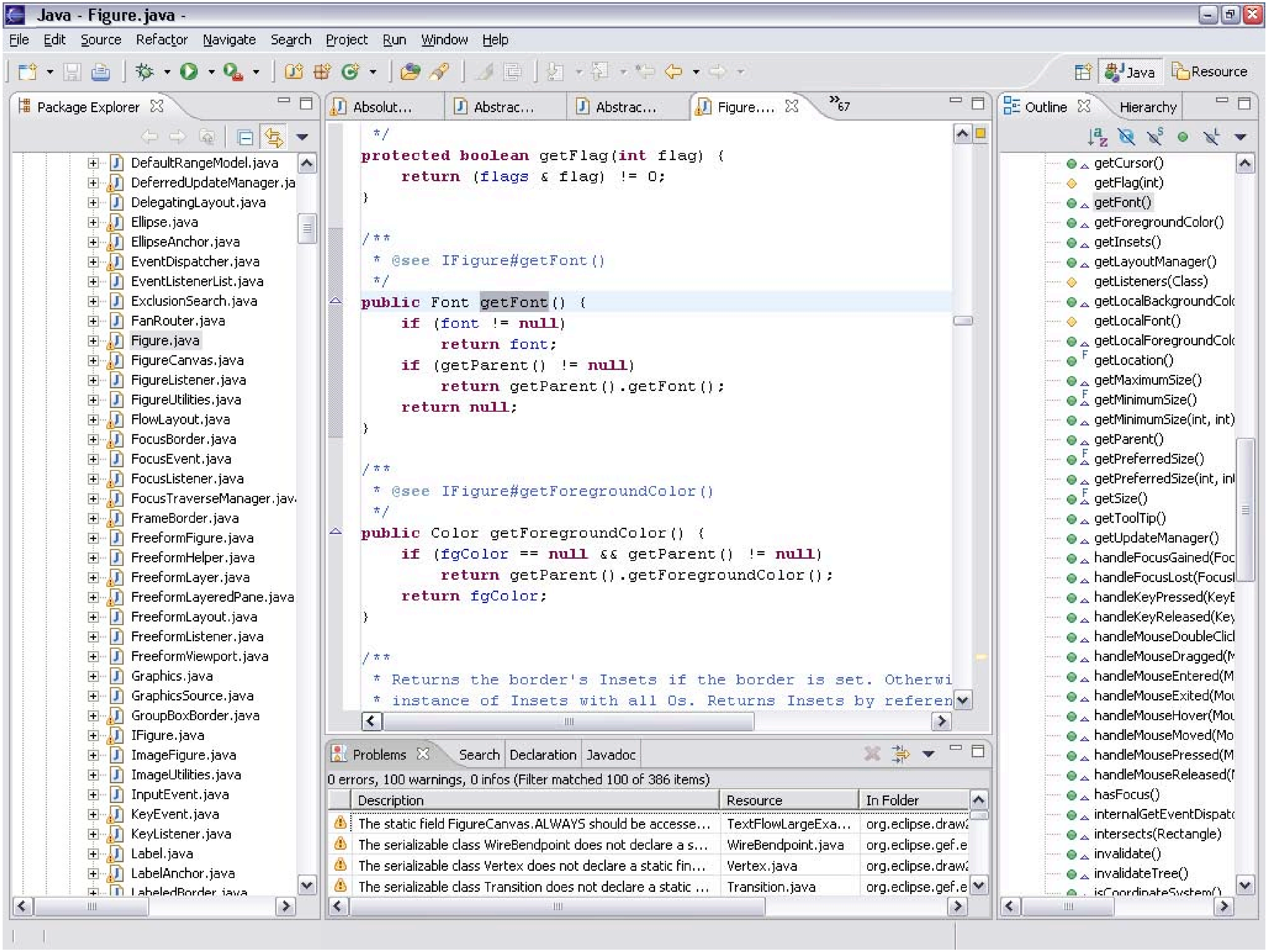
- Intro to AOP
- Other Examples
- Is AOP Code Modular, Abstract

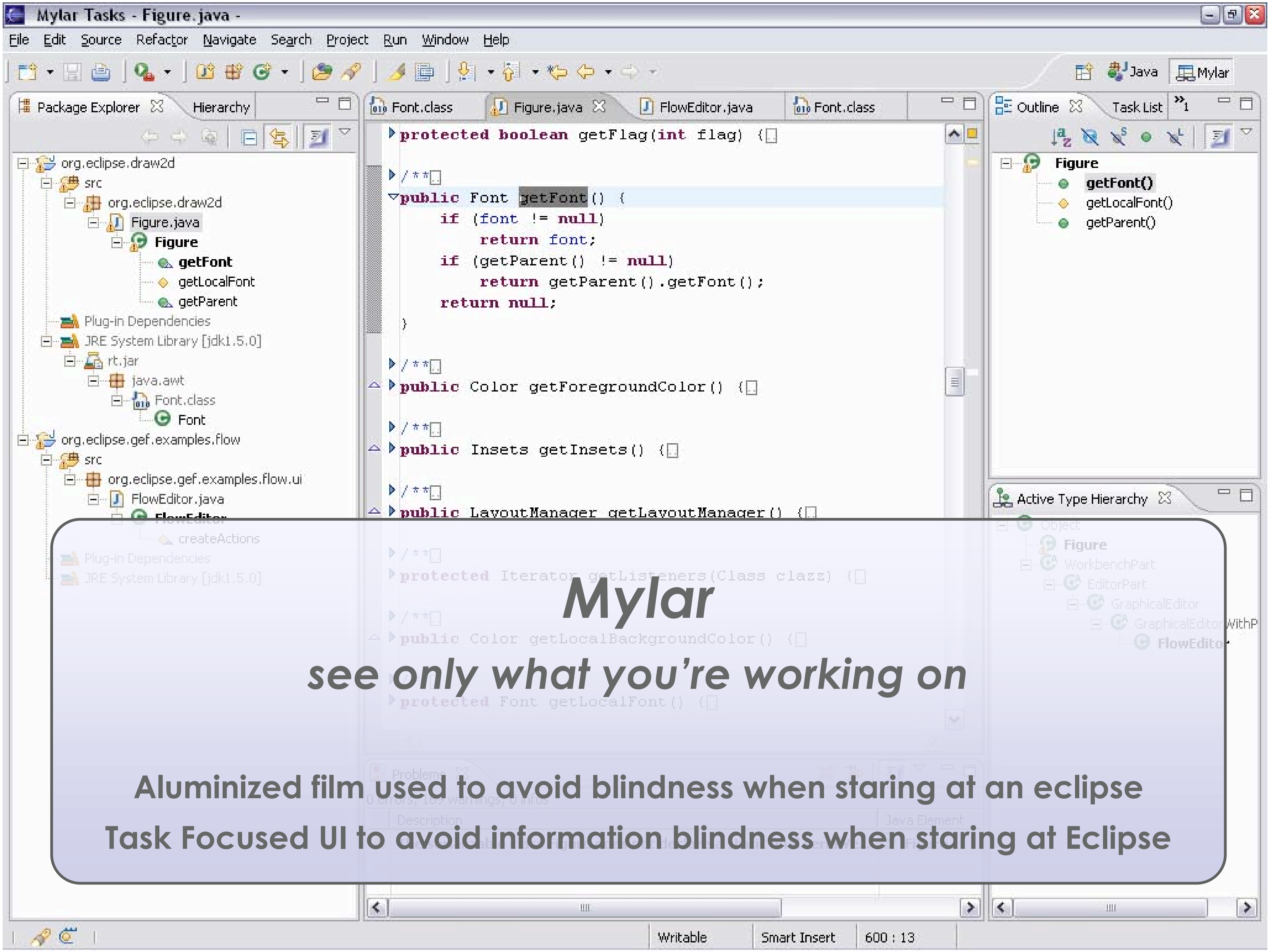
- Join Point Models
- Future Possibilities


ackage Exp...
DisplayUpdating.fa
*Line.java
*Point.java
Java
DrawingApp
figures
Display.java
Driver.java
Line.java
Point.java
Shape.java
JRE System Library [jre1
annotation.jar - C:\Ter
DisplayUpdating.fa
public aspect DisplayUpdating
pointcut change :
declaration(public * figures.Shape+.set*(*))
after returning : change
{
}


Java
DisplayUpdating.fa
acka...
DrawingApp
figures
Display.java
Driver.java
Line.java
Logger.java
Point.java
Shape.java
JRE System Library [jre
annotation.jar - C:\Te
DisplayUpdating.fa
public aspect DisplayUpdating
pointcut change : declaration(public * figures.Shape+.set*(*))
gather : change
{
}


Java
*DisplayUpdating.fa
Line.java
Point.java
DrawingApp
figures
Display.java
Driver.java
Line.java
Logger.java
Point.java
Shape.java
JRE System Library
annotation.jar - C:\T
DisplayUpdating.fa
public aspect DisplayUpdating {
pointcut change : declaration(public * figures.Shape+.set*(*))
overlay : change
{
----
public void set ( ){
this. = ;
Logger.log( );
}
}
}


Java - Figure.java -
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Resource
Package Explorer
DefaultRangeModel.java
DeferredUpdateManager.ja
DelegatingLayout.java
Ellipse.java
EllipseAnchor.java
EventDispatcher.java
EventListenerList.java
ExclusionSearch.java
FanRouter.java
Figure.java
FigureCanvas.java
FigureListener.java
FigureUtilities.java
FlowLayout.java
FocusBorder.java
FocusEvent.java
FocusListener.java
FocusTraverseManager.jav
FrameBorder.java
FreeformFigure.java
FreeformHelper.java
FreeformLayer.java
FreeformLayeredPane.java
FreeformLayout.java
FreeformListener.java
FreeformViewport.java
Graphics.java
GraphicsSource.java
GroupBoxBorder.java
IFigure.java
ImageFigure.java
ImageUtilities.java
InputEvent.java
KeyEvent.java
KeyListener.java
Label.java
LabelAnchor.java
LabeledBorder.java
Absolut...
Abstrac...
Abstrac...
Figure....
»67
*/
protected boolean getFlag(int flag) {
return (flags & flag) != 0;
}
/**
* @see IFigure#getFont()
*/
public Font getFont() {
if (font != null)
return font;
if (getParent() != null)
return getParent().getFont();
return null;
}
/**
* @see IFigure#getForegroundColor()
*/
public Color getForegroundColor() {
if (fgColor == null && getParent() != null)
return getParent().getForegroundColor();
return fgColor;
}
/**
* Returns the border's Insets if the border is set. Otherwi
* instance of Insets with all 0s. Returns Insets by referen
Problems
Search
Declaration
Javadoc
0 errors, 100 warnings, 0 infos (Filter matched 100 of 386 items)
Description
Resource
In Folder
The static field FigureCanvas.ALWAYS should be accesse...
TextFlowLargeExa...
org.eclipse.draw
The serializable class WireBendpoint does not declare a s...
WireBendpoint.java
org.eclipse.gef.e
The serializable class Vertex does not declare a static fin...
Vertex.java
org.eclipse.draw
The serializable class Transition does not declare a static ...
Transition.java
org.eclipse.gef.e
Outline
Hierarchy
getCursor()
getFlag(int)
getFont()
getForegroundColor()
getInsets()
getLayoutManager()
getListeners(Class)
getLocalBackgroundCol
getLocalFont()
getLocalForegroundCol
getLocation()
getMaximumSize()
getMinimumSize()
getMinimumSize(int, int)
getParent()
getPreferredSize()
getPreferredSize(int, in
getSize()
getToolTip()
getUpdateManager()
handleFocusGained(Foc
handleFocusLost(Focus
handleKeyPressed(KeyE
handleKeyReleased(Key
handleMouseDoubleClic
handleMouseDragged(M
handleMouseEntered(M
handleMouseExited(Mou
handleMouseHover(Mou
handleMouseMoved(Mo
handleMousePressed(M
handleMouseReleased(
hasFocus()
internalGetEventDispat
intersects(Rectangle)
invalidate()
invalidateTree()



Mylar

see only what you're working on

Aluminized film used to avoid blindness when staring at an eclipse

Task Focused UI to avoid information blindness when staring at Eclipse

Radical Research in Modularity

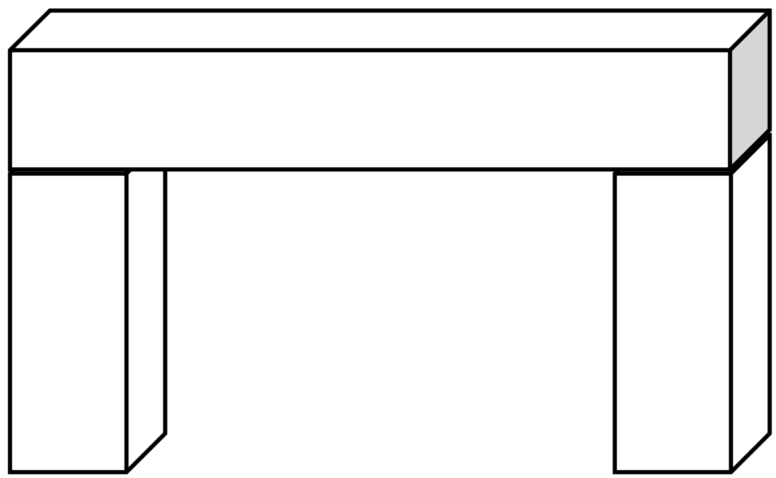
- AOP ala AspectJ can make programs
- Hierarchical structure insufficient
  - does not support all needed (de)composition
  - even a simple example shows this
- Crosscutting structure is inherent
  - and can be supported modularly

- A module should be able to be
  - any unit of concern
  - at any time, we should support
    - identification, localization, interface construction…
- Abstraction should be
  - ability to set aside currently irrelevant details

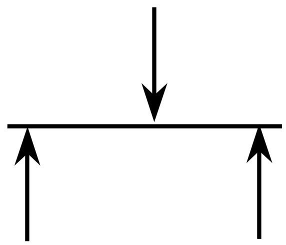
- For example
  - AspectJ style AOP
    - static modules, dynamically constructed interfaces
  - Fluid AOP, Mylar
    - dynamic modules, dynamic interfaces

- This might put some more ‘soft’ in software?

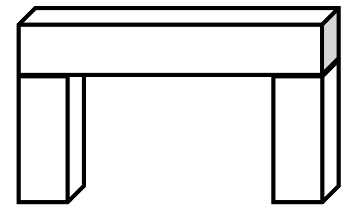
a simple bridge



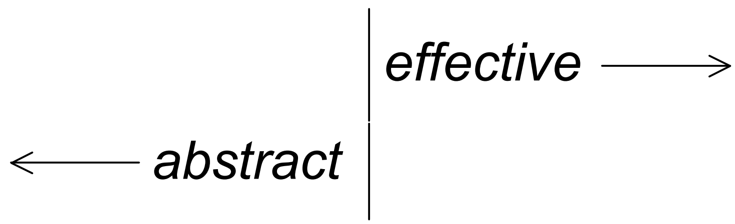
models, programs and systems



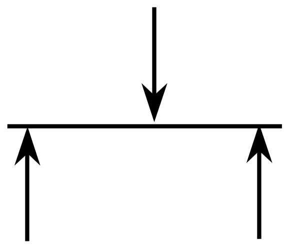
model



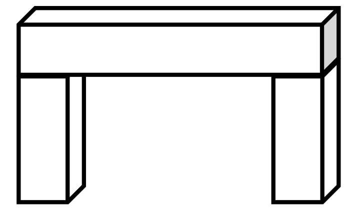
system



models, programs and systems



```
i = 1
while (i < 4) {
  print(i)
  i = i + 1
}
```



model

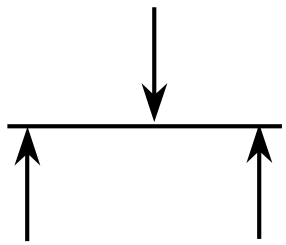
system

effective ⟶

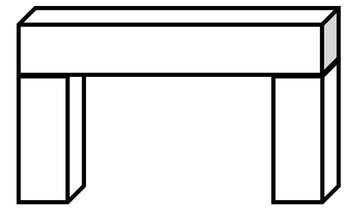
⟵ *abstract*

programs live in this magic space

models, programs and systems



```
i = 1
while (i < 4) {
  print(i)
  i = i + 1
}
```



model

system

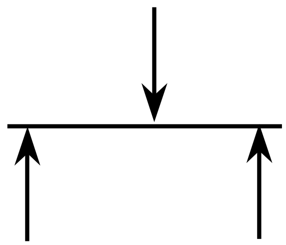
effective →

← *abstract*

programs live in this magic space

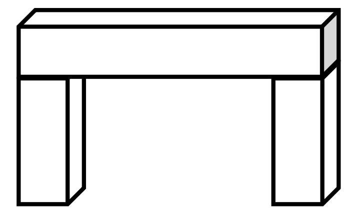
Brian's account talks (in part) about this space

models, programs and systems



model

```
i = 1
while (i < 4) {
  print(i)
  i = i + 1
}
```



system

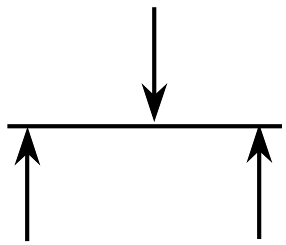
effective →

cc ← *abstract*

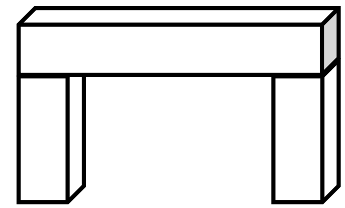
programs live in this magic space

Brian's account talks (in part) about this space

models, programs and systems



```
i = 1
while (i < 4) {
  print(i)
  i = i + 1
}
```



model

system

effective →

cc

← *abstract*

programs live in this magic space

Brian's account talks (in part) about this space

Review So Far

- Aspect is a unit of design, decomposition, composition
  - supported by mechanisms
  - a “learned intuitive way of thinking”
- Mechanisms
  - Pointcuts and advice
    - dynamic join points, pointcuts, advice
  - Inter-type declarations
- Different concepts for different structure of concerns
  - procedure holds computeRadius, setX…
  - class holds Point, Line…
  - aspect holds ObserverPattern…
- Aspects
  - modular units of implementation
  - look like modular units of design
  - improves design and code