AOP and observer pattern

Design pattern

- General reusable solution to a commonly occurring problem in software design
- Not a finished design that can be transformed directly into source code
- It is a template for solving a problem
- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

Observer design pattern

- In many programs, when a object changes state, other objects may have to be notified
  - Example: when an car in a game is moved
    - The graphics engine needs to know so it can re-render the item
    - The traffic computation routines need to re-compute the traffic pattern
    - The objects the car contains need to know they are moving as well
  - Another example: data in a spreadsheet
    - The display must be updated
    - Possibly multiple graphs that use that data need to re-draw themselves
- This pattern answers the question: How best to notify those objects when the subject changes?
  - And what if the list of those objects changes during run-time?

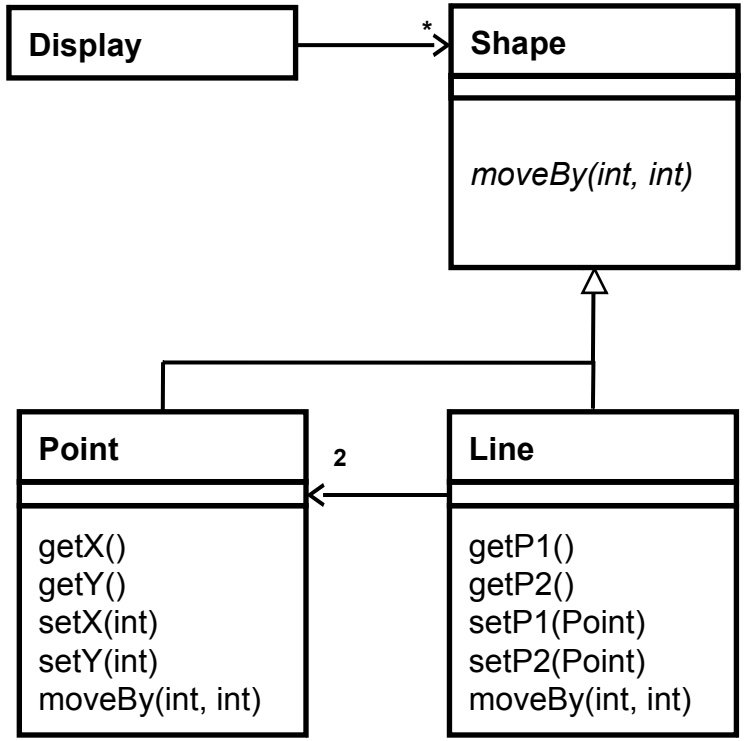
The key participants in observer pattern

- The Subject, which provides an (virtual) interface for attaching and detaching observers
- The Observer, which defines the (virtual) updating interface
- The ConcreteSubject, which is the class that inherits/extends/implements the Subject
- The ConcreteObserver, which is the class that inherits/extends/implements the Observer

- This pattern is also known as dependents or publish-subscribe

GUI example

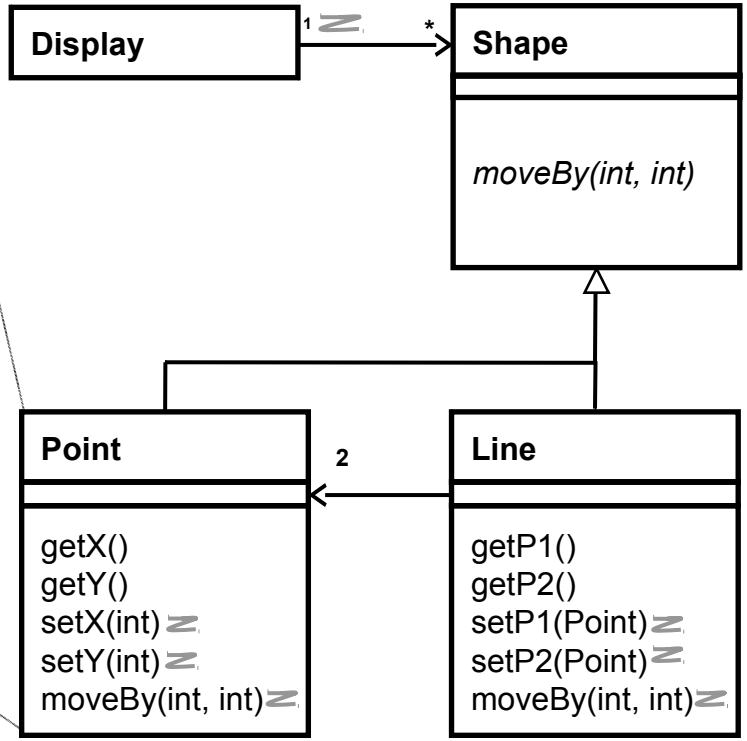
- Intuitive way of thinking ---
  - Points, Lines, Shapes,
  - encapsulation, class hierarchy
- Code is the natural reflection of the intuitive way of thinking



Poor code modularity

- Display needs to be updated every time a shape is moved

```
class Point extends Shape {
  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);
  }
}
```

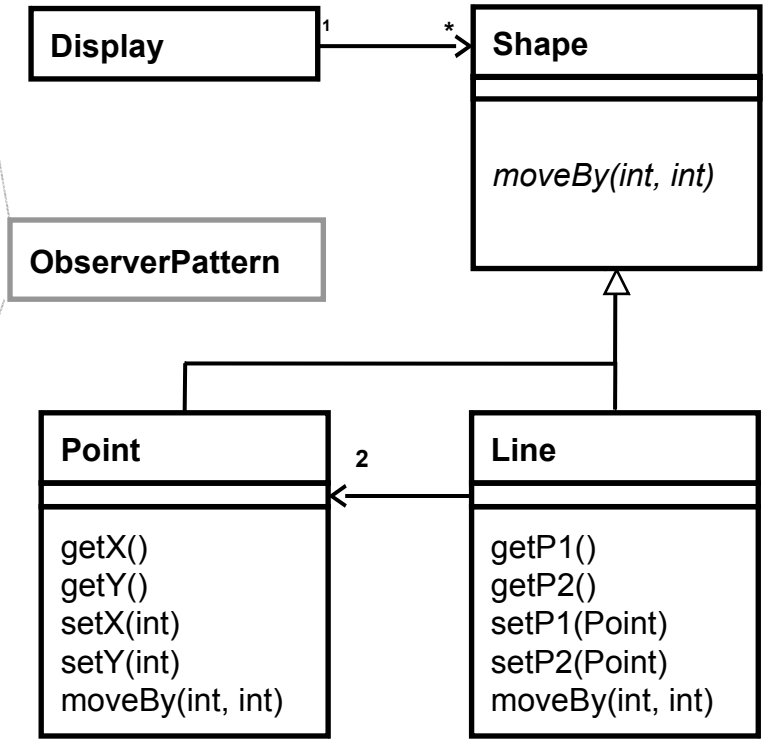


From Kiczales's slides

AOP makes code look like design

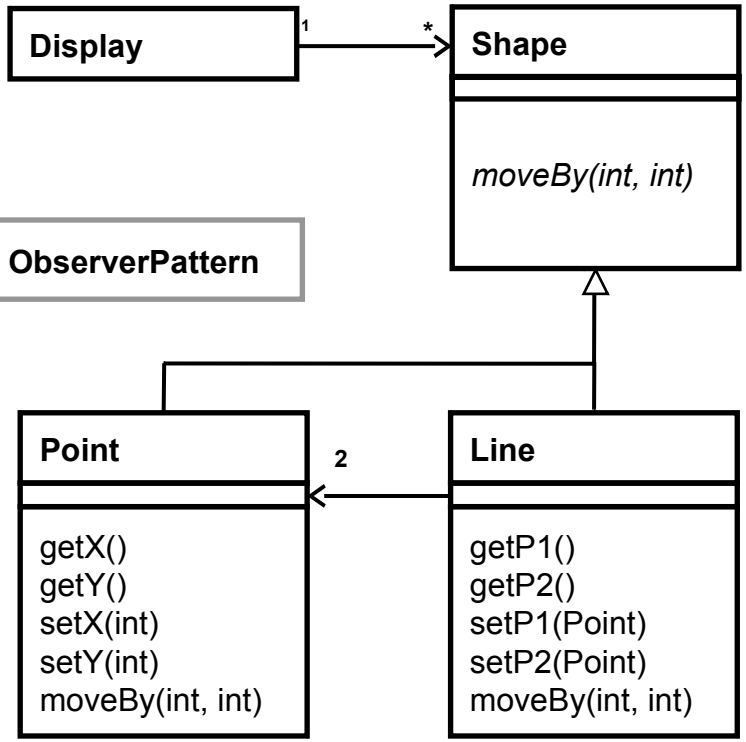
- It is impossible to write a class to encapsulate the ObserverPattern

```
aspect ObserverPattern {

  private Display Shape.display;

  pointcut change():
    call(void figures.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();
  }
}
```

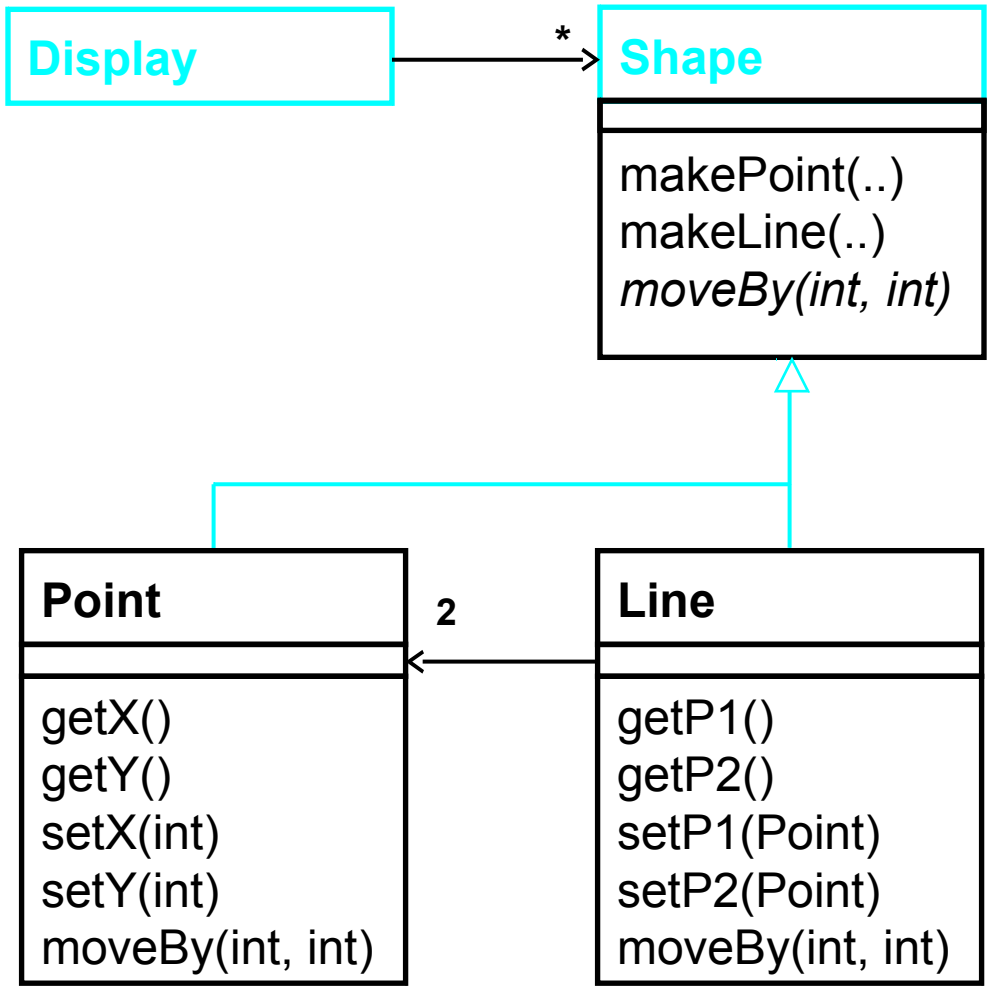


The ObserverPattern can be simplified

```
aspect ObserverPattern {

  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();
  }
}
```



What OOP develop can see



Without AspectJ

"display updating" is not modular

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

```
class Line extends Shape{
  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 extends Shape{
  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

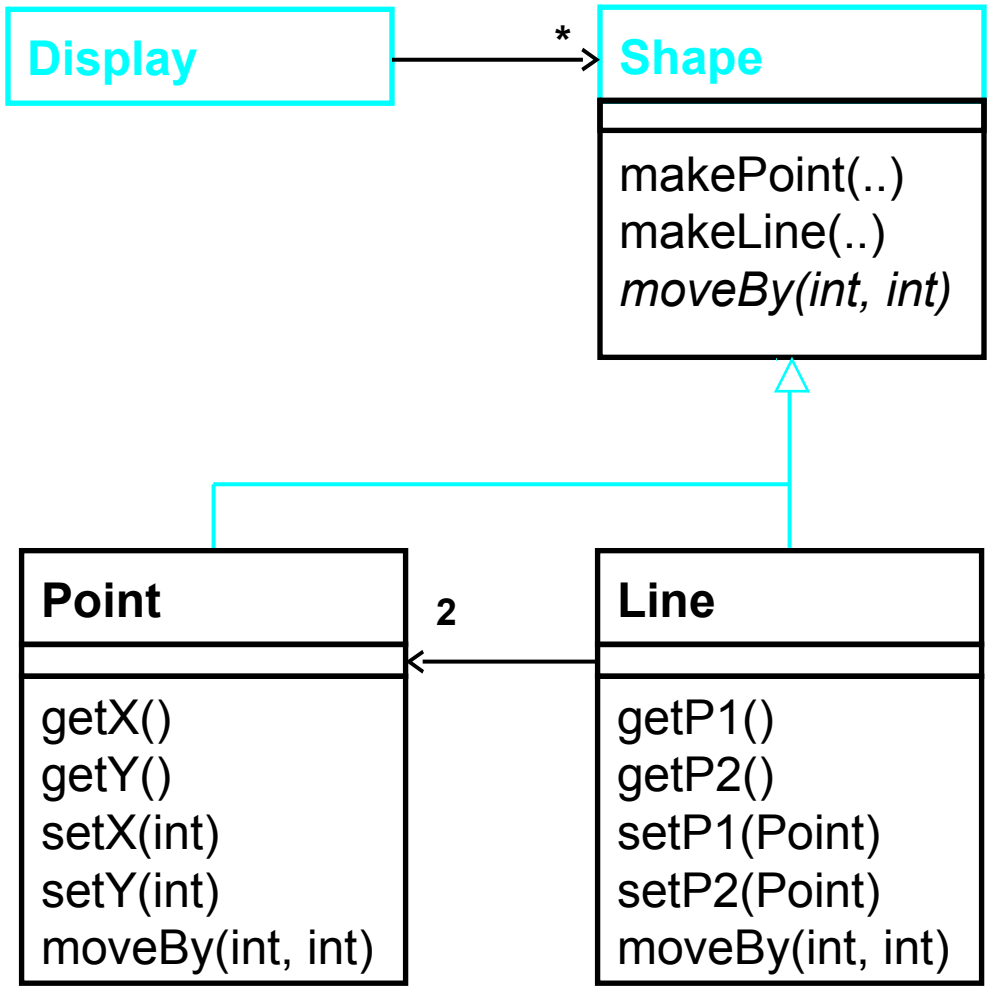
```
class Line extends Shape {
  private Point p1, p2;

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

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}
class Point extends Shape {
  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():
    execution(void Line.setP1(Point)) ||
    execution(void Line.setP2(Point));

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

Top level changes

```
aspect ObserverPattern {

  pointcut change(Shape shape):
    this(shape) &&
    (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)));

  pointcut topLevelchange(Shape s):
    change(s) && !cflowbelow(change(Shape));

  after(Shape shape) returning: topLevelchange(shape) {
    Display.update(shape);
  }
}
```

What AOP developers see



ObserverPattern

BoundsChecking

FactoryEnforcement

Boundary check

```
aspect BoundsPreConditionChecking {

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

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