Aspect Oriented Programming & AspectJ

Mariano Ceccato
ceccato@itc.it

Outline

- The problem: Crosscutting concerns
- The Solution: Aspect Oriented Programming
- The Language: AspectJ
- Examples

The role of OOP

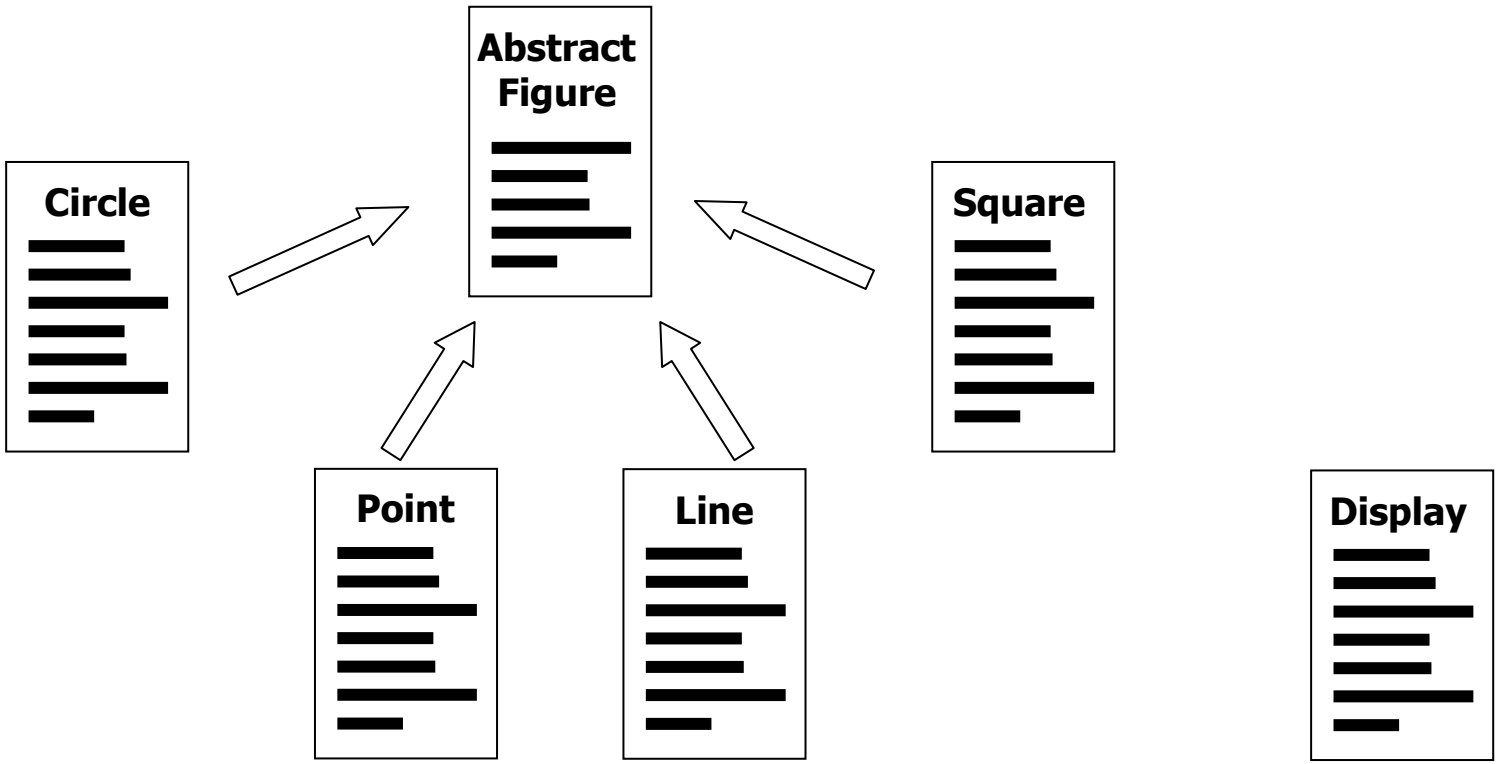
- Object Oriented Programming (OOP) has been proposed to make software development/understanding easier.

- Object are quite near to how humans see, understand and think of the world.

Drawing appl. example

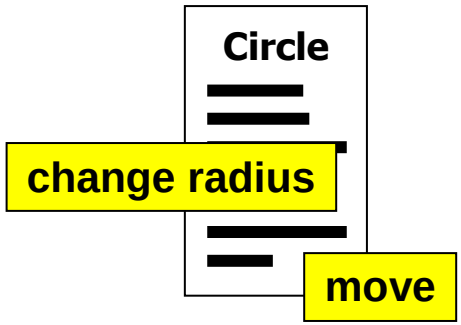
Requirements:

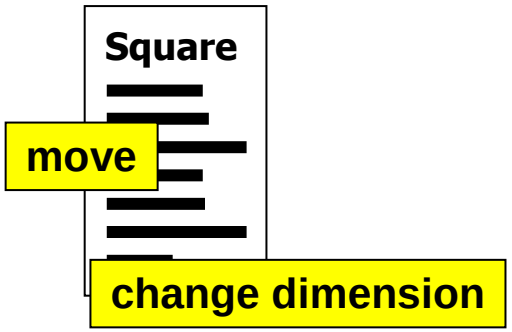
- Geometrical shapes (circles, points, lines and squares)
- Draw, modify and delete geometrical shapes
- Render the drawing

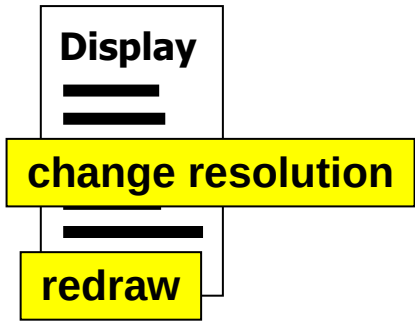


Functionalities mapping

- The whole application is modularized according to the entities (objects) composing the problem domain.
- It should be easier for developers to:
  - Assign functionalities to objects.
  - Think of the application in terms of a set on objects, instead of a set of functionalities.

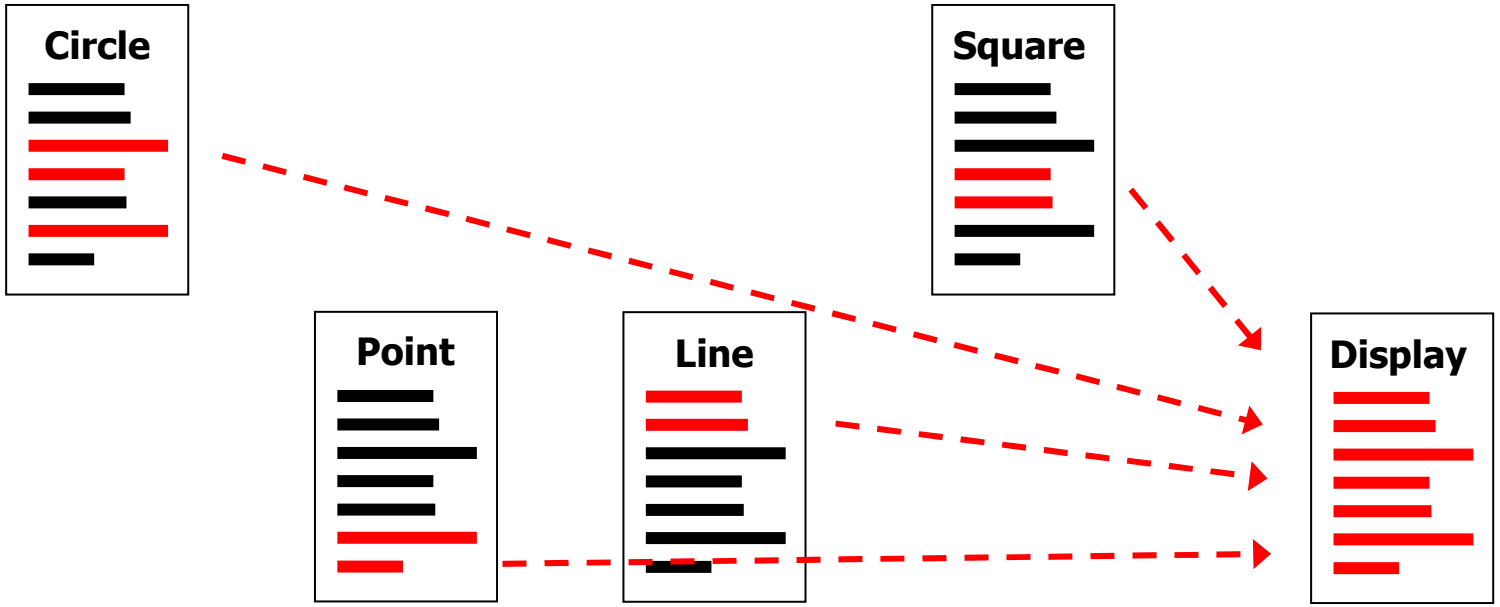






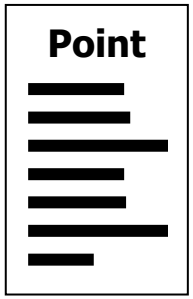
Drawing appl. example

- Display management functionality is isolated info the display class.
- Display change notification is scattered through all the other classes.
- Notification responsibility belongs to the display but it is implemented in the geometrical shape classes.



Drawing appl. example

```
class Point {

  private int x, y;

  public int getX() {
    return x; }

  public int getY() {
    return y; }

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

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

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

  public void update() {
    Display d =
      Display.getDefault();
    d.update();
    d.commit();
  }
}
```



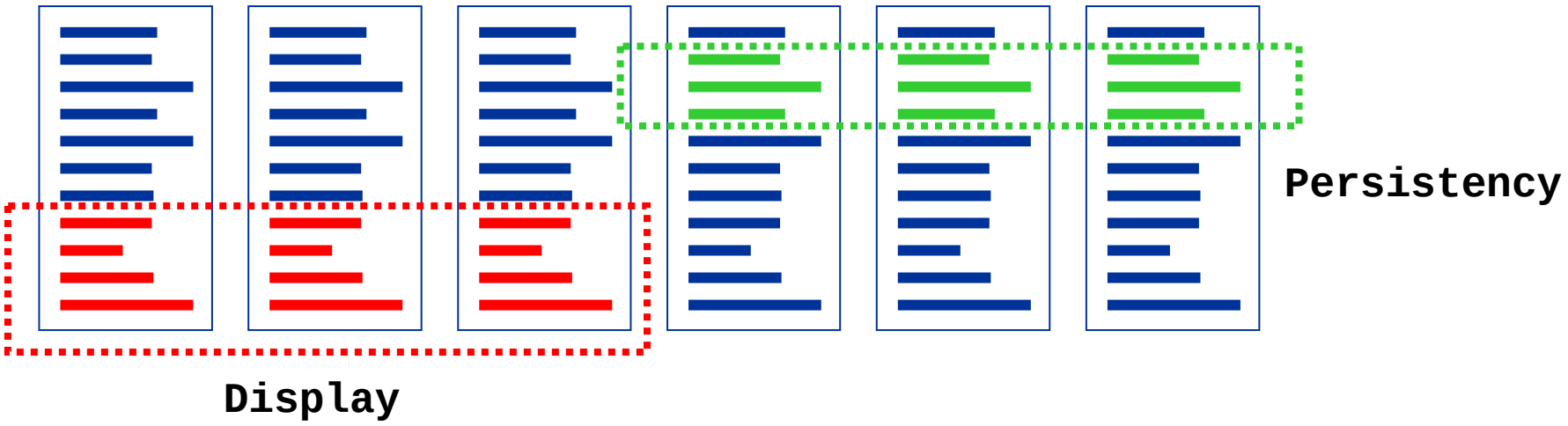
Limitations of OOP

- There are functionalities that do not fit on the principal decomposition:
  - They traverse the principal decomposition of the application.
  - They can not be assigned to (separated in) a single modular unit.

- Existing software often contains several Crosscutting Concerns such as:
  - Logging, tracing, persistency, exception handling.

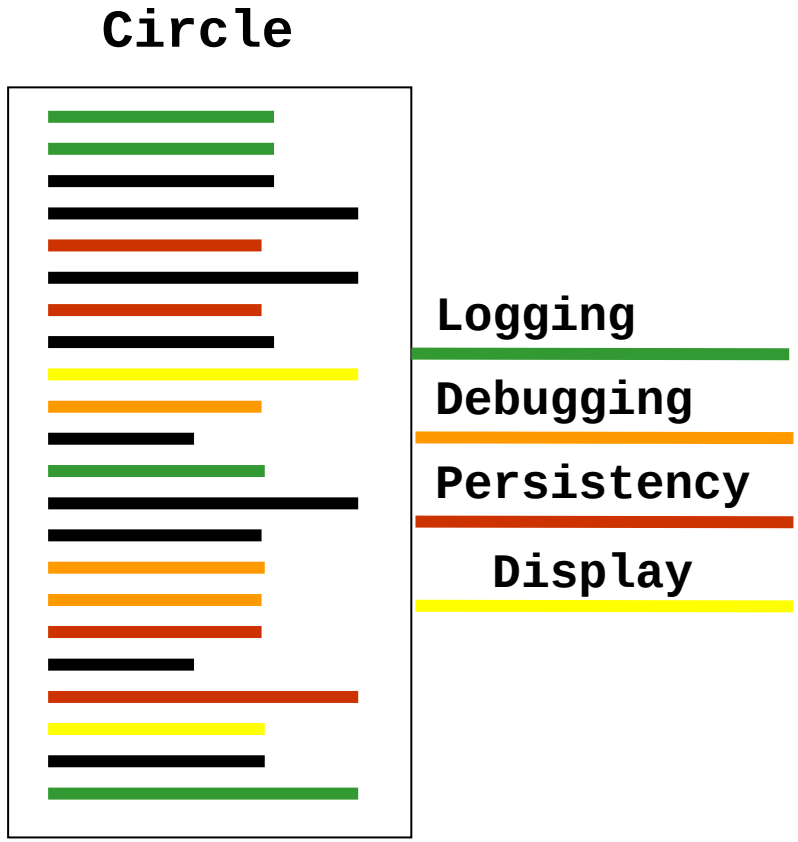
Scattering

- The code implementing the same CC is spread in many different modules.
- In general there is no language support to understand which modules have a role in the same concern.



Tangling

- The code pertaining to a crosscutting concern is intermixed with the rest of the module code.
- The same module can be affected by several crosscutting concern.
- Concerns can not be separated into different modules using standard modularization mechanisms.



Drawbacks

Understandability:

- Crosscutting concerns break OOP guidelines.
- It can be not so easy to see the presence of a crosscutting concern (programming languages don’t support it).
- It can be hard to distinguish the principal responsibility of a module from the crosscutting concerns.

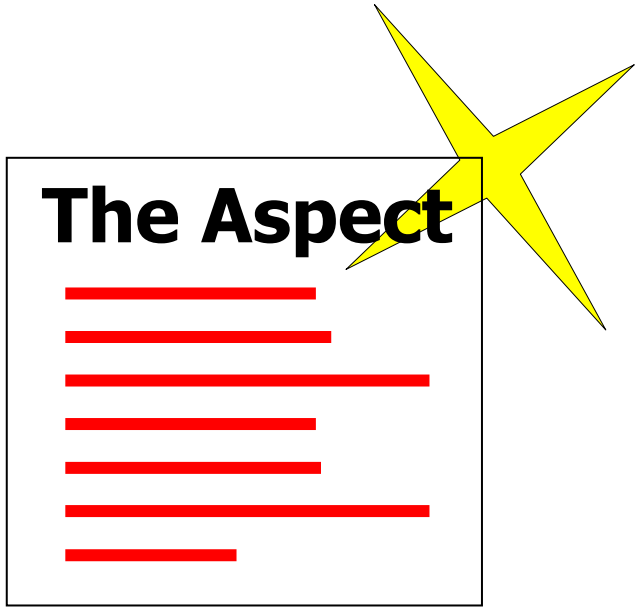
Drawbacks

Maintenance:

- Scattered functionalities could be hard to find.
- Modifying a crosscutting concern requires to change a lot of modules in the same time (loss of modularization benefits)
- When a change is required on a crosscutting concern
  - Mentally untangle the concern from the base code.
  - Perform the change.
  - Re-tangle concerns together with base code.

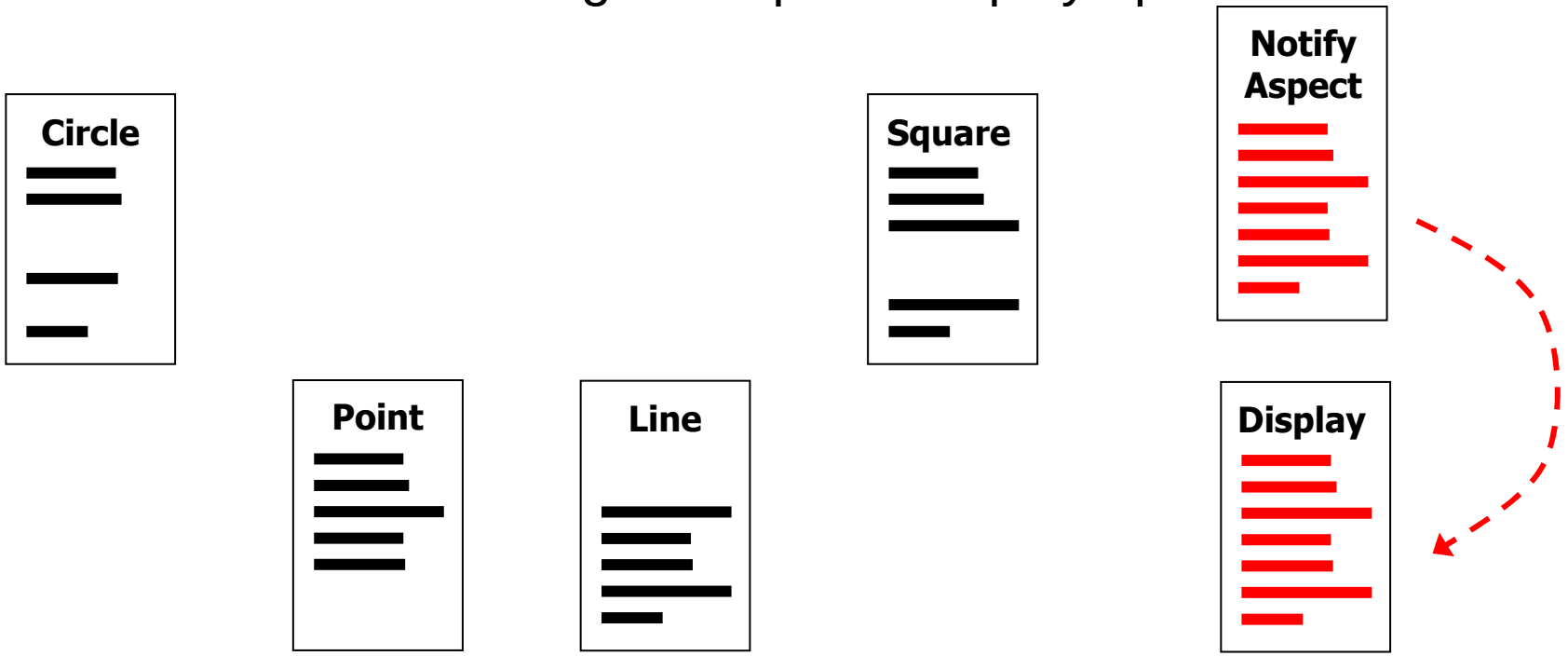
Solution AOP

- AOP defines a new kind of module: the aspect
- An aspect is able to isolate a crosscutting concern
- A class should implement only its defining responsibility
- A class should be oblivious of the presence of the aspect.



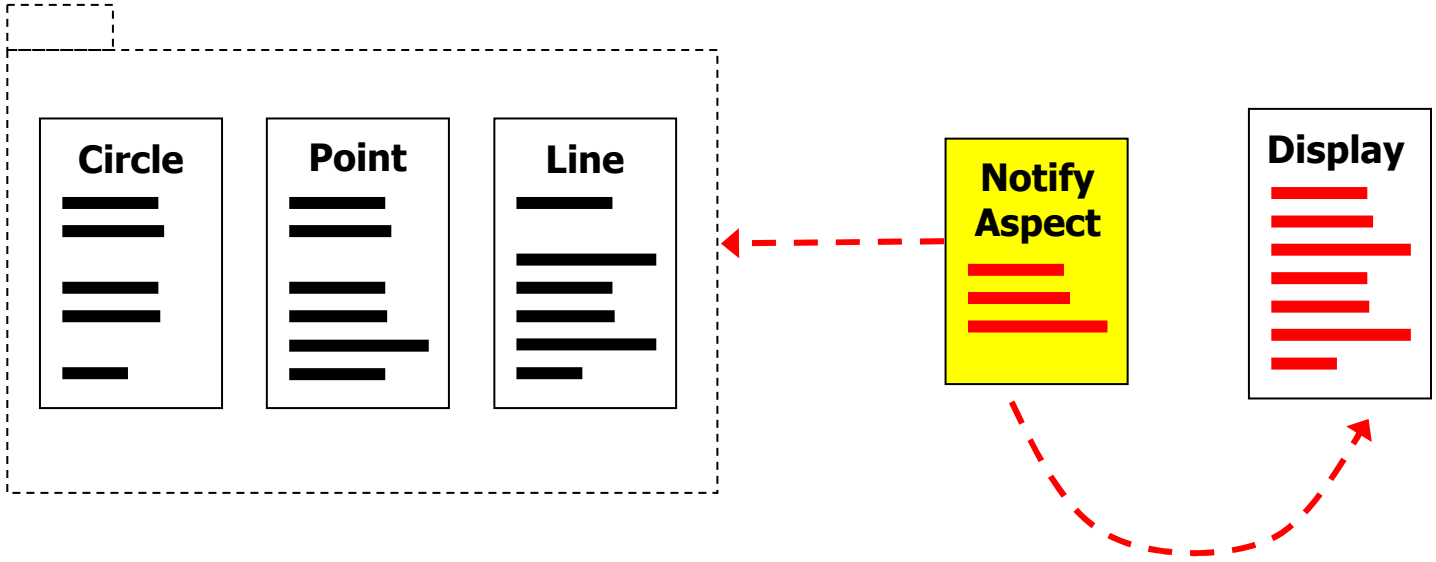
Drawing appl. example

- Drawing figure classes
  - contain only their own defining functionalities
  - are oblivious of the notification mechanism
- The notification aspect
  - knows when each figure requires display update



Aspect Oriented Programming

- The aspect looks at the system execution and when/whether the service is required
  - The aspect stops the main execution
  - The service is started and executed on the current object
  - The execution is resumed



Example - OOP

```
class Point {

  private int x, y;

  public int getX() {
    return x; }

  public int getY() {
    return y; }

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

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

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

  public void update() {
    Display d =
      Display.getDefault();
    d.update();
    d.commit();
  }
}
```

Example - AOP

```
class Point {

  private int x, y;

  public int getX() {
    return x; }

  public int getY() {
    return y; }

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

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

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




}
```

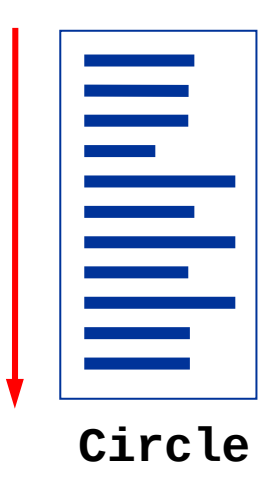
Example - AOP

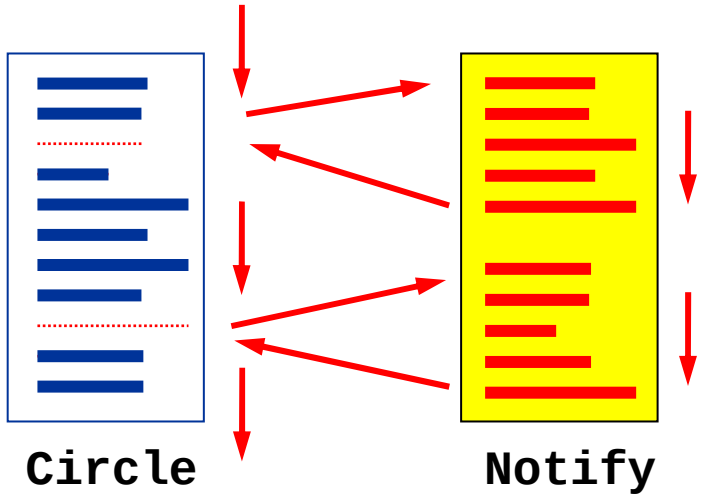
```
class Point {

  private int x, y;

  public int getX() {
    return x; }

  public int getY() {
    return y; }

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

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

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



}
```

```
aspect Change{

public void Point.update() {
  Display d = Display.getDefault();
  d.update();
  d.commit(); }

pointcut change(Point p): this(p) && (
 execution( void Point.setX(int) ) ||
 execution( void Point.setY(int) ) ||
 execution( void Point.moveBy(int, int))
 );

after(Point p): change(p) {
   p.update();}

}
```

Dynamic crosscutting

- The crosscutting concern is implemented by modifying the program control flow.
- The points in the control flow that require the concern are said *join points*
- An aspect use *pointcuts* to locate join points in the base code
- When the control flow reach a join point, the aspect takes control and it execute the matching *advice*





Quantification (pointcut)


```
class Point {

  ...

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

   ...
}
```

```
pointcut change():
   execution( void Point.setX(int) );
```

AspectJ Pointcut Model

- Native pointcuts intercept:
  - Method/constructor invocations
  - Method/constructor executions
  - Field accesses
  - Exception handling
  - Class initialization

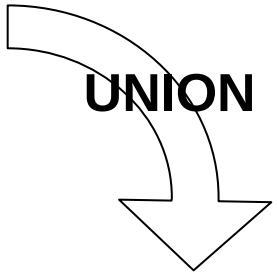
Pointcut composition

- Composition of simple pointcuts using:
  - Union
  - Intersection
  - Negation

```
execution( void Point.moveBy(int, int))
```

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

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




```
pointcut change():
  execution( void Point.setX(int) ) ||
  execution( void Point.setY(int) ) ||
  execution( void Point.moveBy(int, int));
```

Context exposition

- The join point context can be exposed by the pointcut:
  - Method parameters
  - Caller object
  - Called object

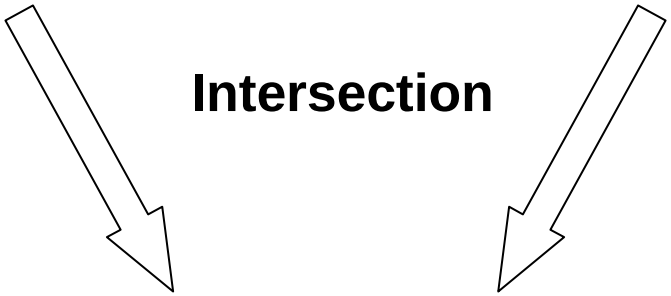
```
class Point {

  ...

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

   ...
}
```

```
pointcut p1(Point p):
  && this(p) ;
```

Pointcut composition

```
pointcut change():
  execution( void Point.setX(int) ) ||
  execution( void Point.setY(int) ) ||
  execution( void Point.moveBy(int, int));
```

```
pointcut p1(Point p):
  && this(p) ;
```



Intersection

```
pointcut change(Point p):
  this(p) &&
    (
    execution( void Point.setX(int) ) ||
    execution( void Point.setY(int) ) ||
    execution( void Point.moveBy(int, int))
    );
```

Advices

- Advices are method-like constructs attached to join points:
  - **Before** advice is executed before the corresponding join point is reached.
  - **After** advice is executed after the corresponding join point.
  - **Around** advice is executed instead of the corresponding join point. The portion of code in the join point can be run in the advice body.

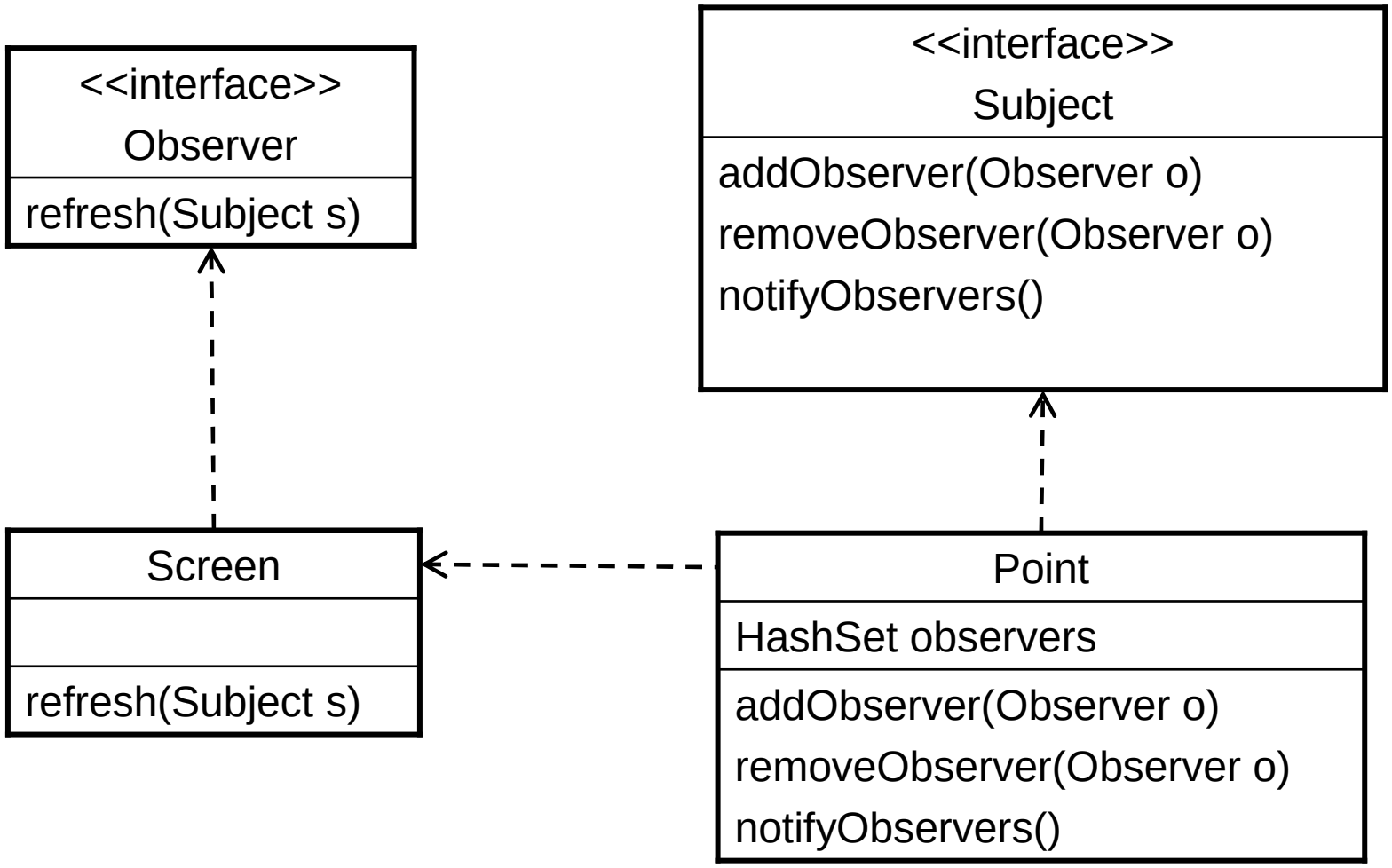
- An advice can use the context exposed by the pointcut:
  - Actual parameters
  - Caller/called object

```
after(Point p):
  p1(p)
    {
    System.out.println("Point notifies update");
    p.update();
    }
```

Weaving aspect code

- Aspects and classes are weaved together:
  - Fresh fields and methods are generated for fields and method introductions.
  - For each pointcut, the join points are computed.
  - Advice code is inserted in the corresponding join points
- The weaver output is OO-compliant and it is compiled using the standard compiler.

Example: Observer Design Pattern



References

- http://www.eclipse.org/aspectj/
- http://www.eclipse.org/ajdt/
- http://aosd.net/2006/index.php
- http://star.itc.it/ceccato/