Software Design & Programming Techniques

Aspect-Oriented Programming with AspectJ

Prof. Dr-Ing. Klaus Ostermann

Based on slides by Prof. Dr. Mira Mezini

Aspect-Oriented Programming with AspectJ

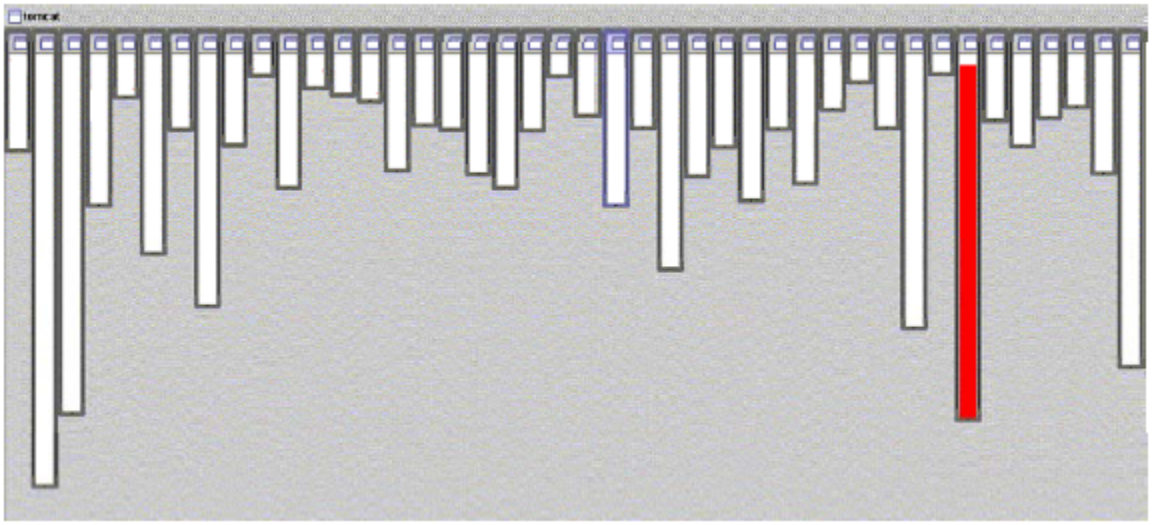
7.1 Goal of Aspect-Oriented Programming (AOP)

Increase code modularity by enabling improved separation of concerns.

- Concerns are cohesive areas of functionality.
  E.g. visualization, exception handling, persistence …
- The goal is to modularize the code that implements concerns.
- Make software easier to evolve and customize.

Code Modularity

XML parsing in `apache.tomcat` is modular:

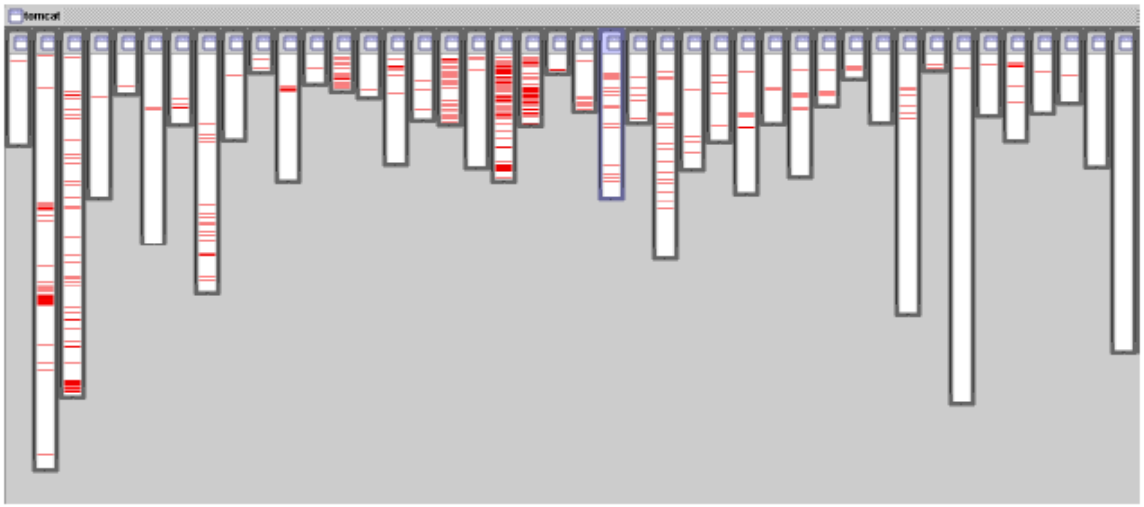


Code implementing a concern is modular if*:*

- It is textually local and not tangled with other concerns.
- There is a well-defined interface.
- The interface is an abstraction of the implementation.
- The interface is enforced.

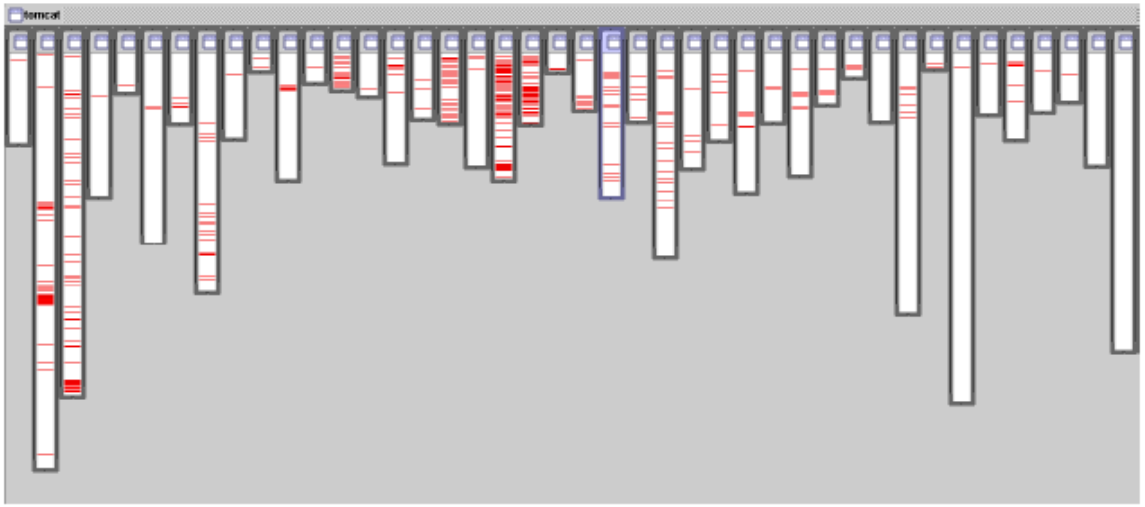
Crosscutting Concerns in OO Software

Session expiration in `apache.tomcat`



- Implementation of session expiration in `apache.tomcat` is not modular.
- Session expiration is a crosscutting concern:
  - Its implementation is cluttered over many classes and packages.
  - Its implementation is tangled with other concerns.

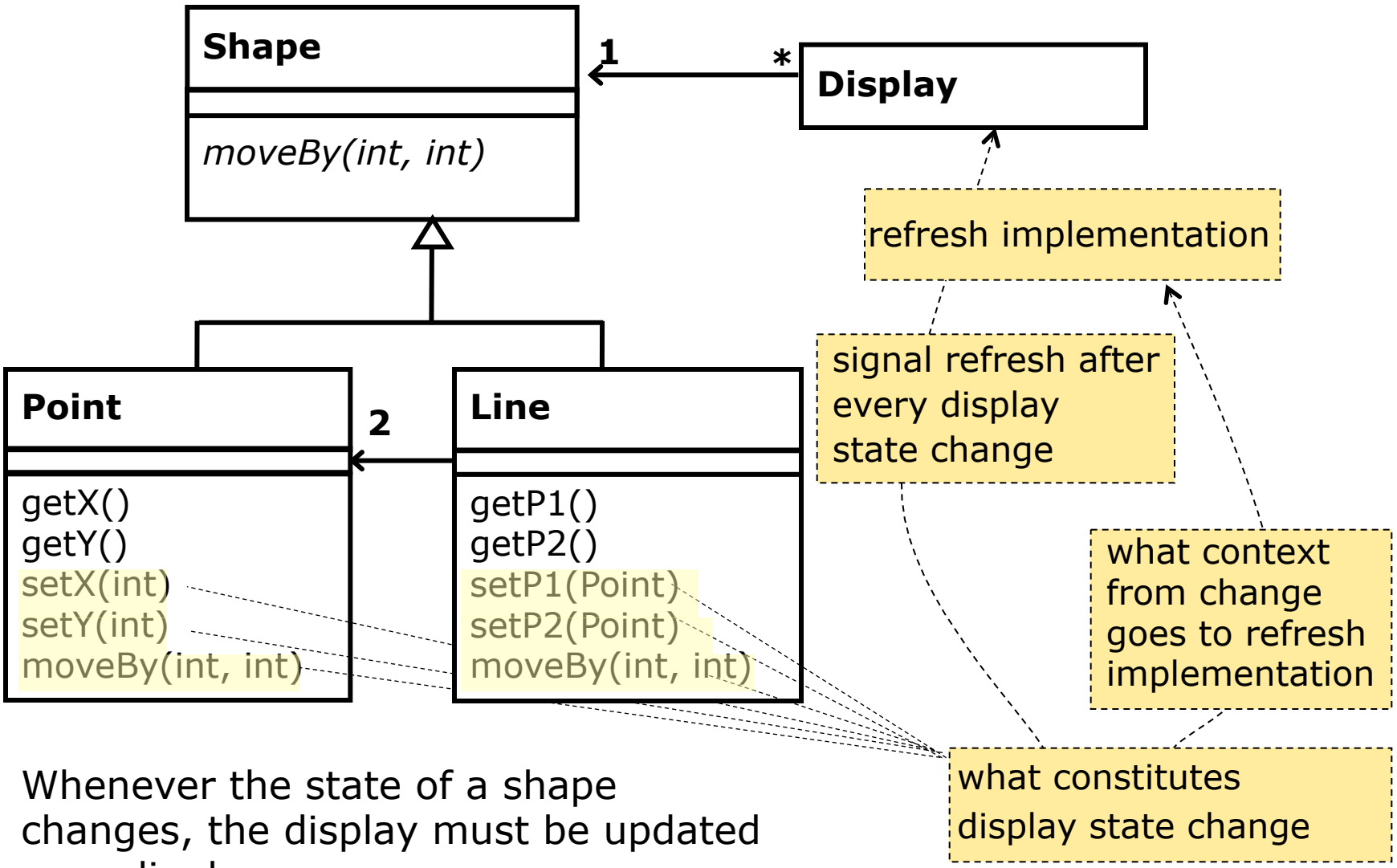
Crosscutting Concerns in OO Software



There are other concerns that "behave" like this:

- exception handling
- performance monitoring and optimizations
- synchronization
- authentication, access control
- transaction & persistence management
- enforcing/checking adherence to architecture / design styles and rules
- co-ordination between objects, e.g., grouping semantics
- …

Illustrative Example



Whenever the state of a shape changes, the display must be updated accordingly.

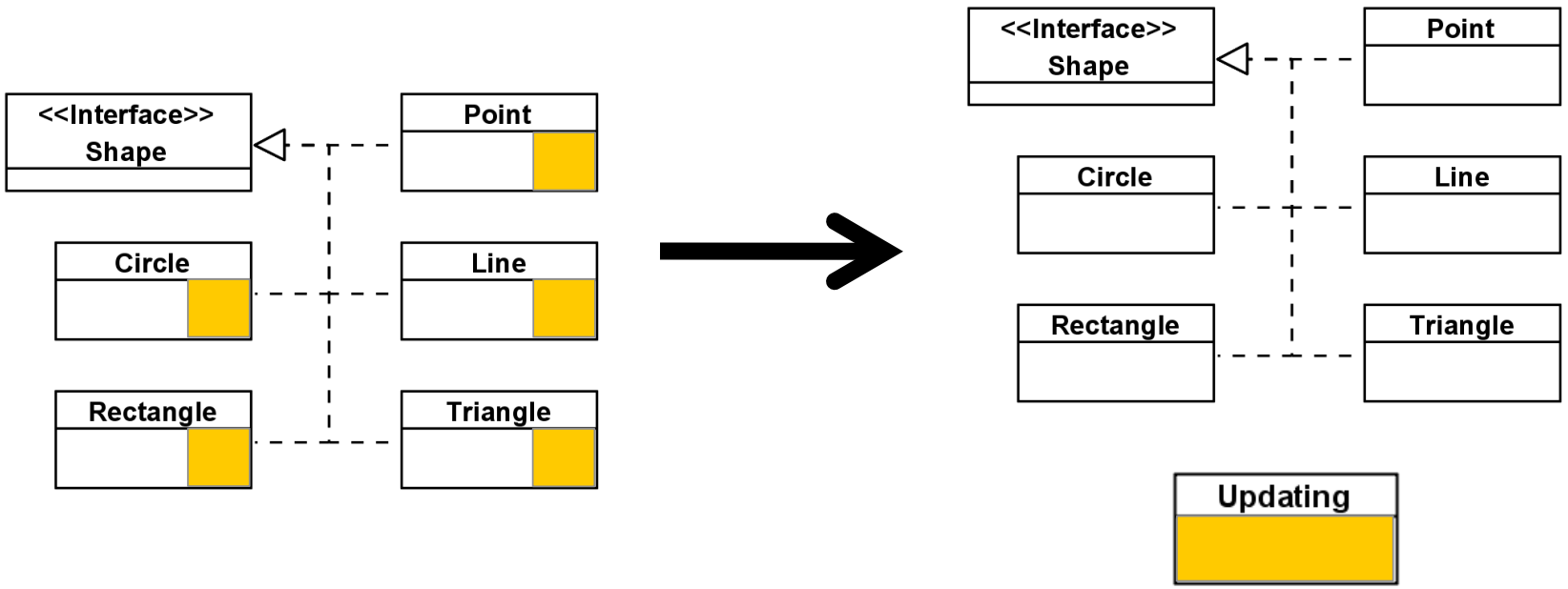
Code for Display Updating Crosscuts the Classes

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

        public void setP1(Point p1) {
                this.x = x;
                Display.update();
        }

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

        public void move(int dx, int dy) {
                p1.move(dx, dy);
                p2.move(dx, dy);
                Display.update();
        }
}
```

- **Updating is scattered over the code of every subclass of `Shape`.**
- Maintenance nightmare: Any change of the updating means adjusting every shape class.

- Do you know a pattern that might be relevant here?
- Does it solve the scattering and tangling problem?

The Goal of AOP Revisited



- Encapsulate the code of crosscutting concerns into separate modules. **"Separation of concerns"**
- More concise, declarative, first-class expression of crosscutting concerns (use of quantification to avoid repetition) in a special kind of modules called aspects.
- Making crosscutting functionality reusable and optional.
- Beware, though, that modularity is more than code localization

7.2 AspectJ in a Nutshell

7.2 AspectJ in a Nutshell

- 7.2.1 Introduction to AspectJ
- 7.2.2 Quick Introduction to Join Points
- 7.2.3 Quick Introduction to Pointcuts
- 7.2.4 Advice
- 7.2.5 A First Example Aspect
- 7.2.6 More on Primitive Pointcuts
- 7.2.7 The Special Value thisJoinPoint

7.2.1 Introduction to AspectJ

- AspectJ is an aspect-oriented extension of Java.
- Originally developed at Xerox Parc and later IBM
- Open source (http://eclipse.org/aspectj)
- Programming Guide: http://eclipse.org/aspectj/doc/released/progguide/

- The next slides will introduce the basic concepts of AspectJ.
- The used terminology is partly specific to AspectJ.

AspectJ Language Elements in a Nutshell

- AspectJ adds the construct Aspect to Java.
- Aspects are on the same level as Interfaces and Classes and have similar elements (methods, attributes, etc.).

- Additionally, an Aspect typically defines AOP-specific elements:
  - **Pointcuts** for querying over **Join Points**
  - **Advice**
  - **Inter-type declarations**

7.2.2 Quick Introduction to Join Points

- **During the execution** of object-oriented programs,
  - there are several kinds of **"things that happen"**
  - what these things are is determined by the language.

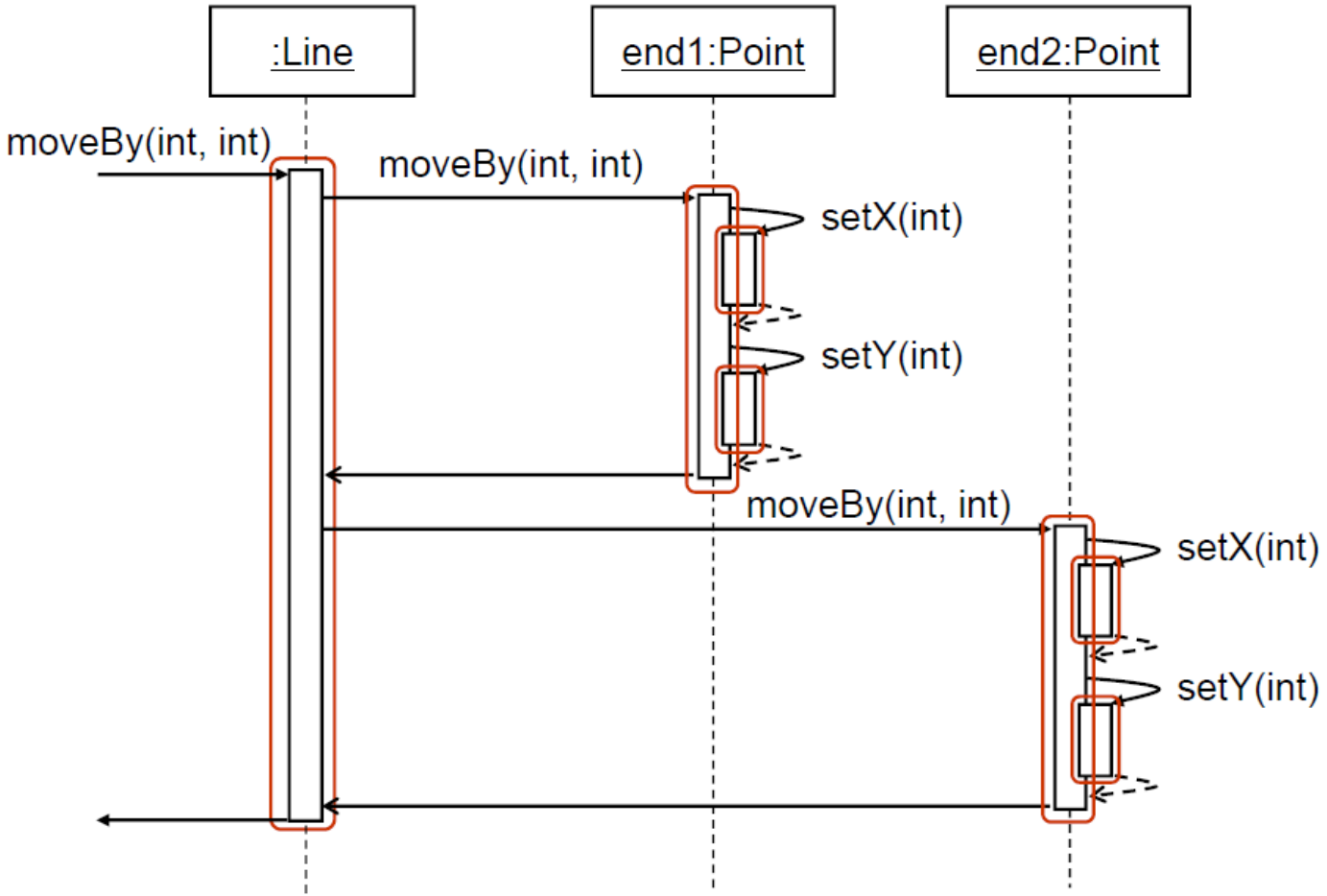
- Things that happen during the execution of a Java program are e.g.,:
  - method calls, method executions,
  - object instantiations, constructor executions,
  - field references
  - handler executions
  - etc.

- In AOP these things are called **join points**.
- Some of them are AO language specific.

Join Points in AspectJ

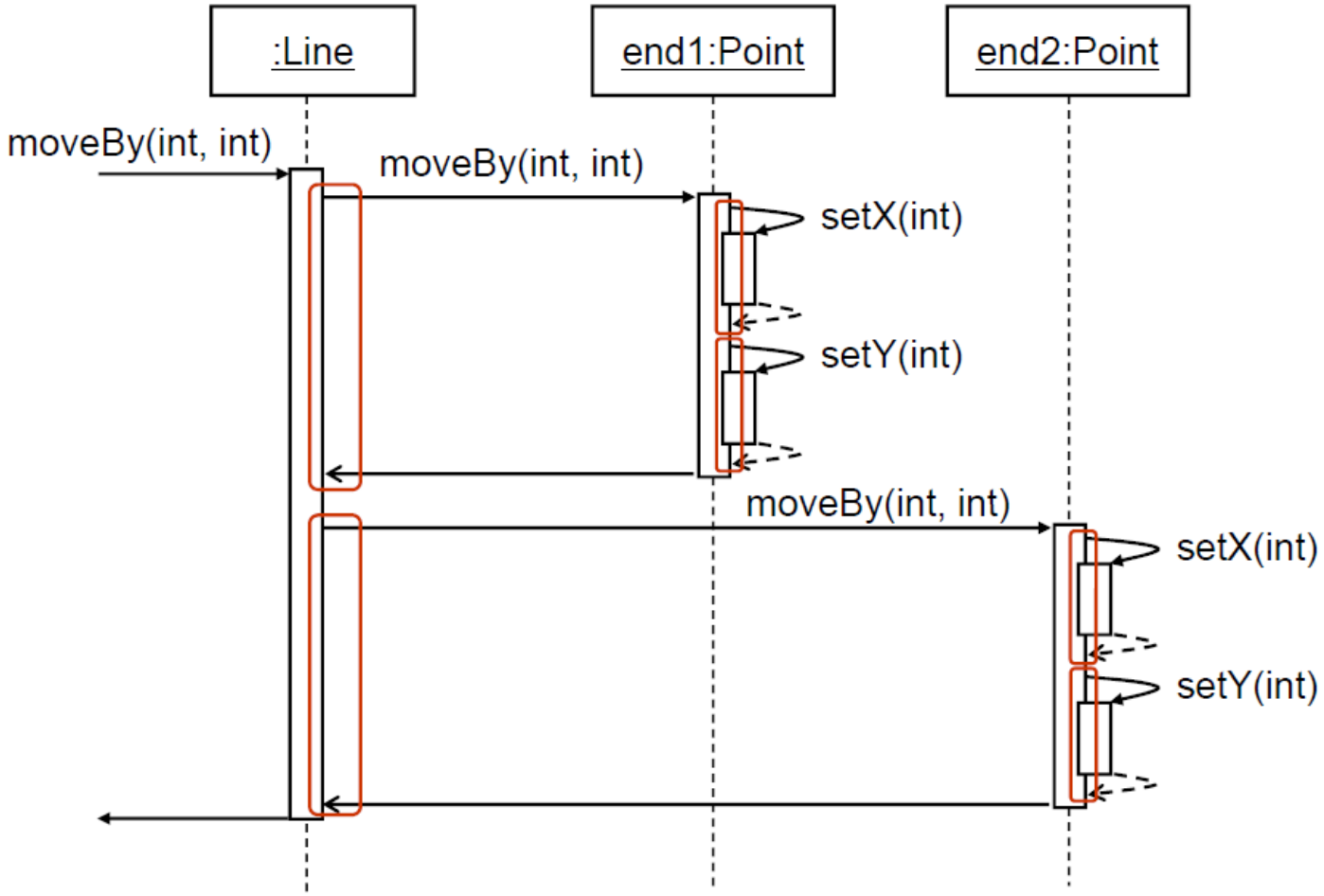
- In AspectJ a **join point is a region in the dynamic control flow of a Java application**.
  - It is in general **not** a place in the source code
- AspectJ supports different kinds of join points:
  - Method call
  - Method execution
  - Constructor call
  - Constructor execution
  - Field get
  - Field set
  - Exception handler execution
  - Pre-initialization
  - Initialization
  - Static initialization

Method Execution Join Points



A method execution join point is the region of the control flow during which a method body executes.

Method Call Join Points



Method call join points exists where a method is called.

7.2.3 Quick Introduction to Pointcuts

- A **pointcut** is an expression that
  - **selects** (matches) **join points**
  - retrieves and **exposes values from the context of selected join points.**

- Exposed values are bound to variables that are declared as parameters of pointcuts.

- AspectJ provides two kinds of pointcuts:
  - **Primitive pointcuts** are built-in the language
  - **User-defined pointcuts** can be defined by composing primitive pointcuts.

Overview of AspectJ's Primitive Pointcuts

- **Kinded primitive pointcuts** select joint points by their kind:
  - `call`, `execution`,
  - `get`, `set`,
  - `handler`,
  - `initialization`, `staticinitialization`
- A kinded pointcut selects join points using **patterns**, some of which ***match based on signature***, and some of which ***match based on modifiers***

```
execution(void Shape.set*(..))
```

- Other pointcuts select join points based on
  - **lexical scope**: `within`, `withincode`
  - **dynamic properties**: `this`, `target`, `args`, `cflow`, `cflowbelow`

User-Defined Pointcuts

Primitive pointcuts can be composed to user-defined pointcuts by means of logical operators, `||`, `&&`, `!`.

```
// Matches every join point, at which the state of
// Shape, Line and Point is changed
pointcut shapeChange():
        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));
```

Example Pointcuts

- The following user-defined pointcut picks out each call to `setX(int)` or `setY(int)` when the target of the call is an instance of `Point`.

```
pointcut setter(): target(Point) &&
                   (call(void setX(int)) ||
                    call(void setY(int)));
```

- The following pointcut picks out each point in the execution of the program, where exceptions of type `IOException` are handled inside the code defined by class `MyClass`

```
pointcut ioHandler():
                within(MyClass) &&
                handler(IOException);
```

7.2.4 Advice

- An advice is a piece of code similar to a method body.
- An advice is associated with a pointcut and executes at any point selected by that pointcut.
- Values exposed by the pointcut are accessible in the advice body.

```
pointcut shapeMove():
        execution(void Shape.moveBy(int, int));

// Advice executing after moveBy was executed on a shape
after returning: shapeMove() {
        ...
}
```

Different Kinds of Advice

- Advice can be defined to execute:
  - `before` proceeding at join point.
  - `after returning` normally (i.e., with a value) from a join point.
  - `after throwing` a throwable from a join point.
  - `after` returning from a join point either way.
  - `around` a join point, which means instead of the join point.
    (One may call `proceed(...)` in advice to run the join point after all.)

7.2.5 A First Example Aspect

- One aspect to manage updating of `Display` for every instance of `Shape`, `Line` and `Point`.
- The entire display updating logic is localized in the `DisplayUpdating` module.
- The interface is defined by the `shapeChange` pointcut.

```
aspect DisplayUpdating {

        pointcut shapeChange():
                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: shapeChange() {
                Display.update();
        }
}
```

7.2.6 More on Primitive Pointcuts

More Kinded Pointcuts

- In addition to method/constructor execution/call pointcuts, there are other primitive kinded pointcuts.

- `get(FieldPattern)`, e.g., `get(int Line.p2)`
  `set(FieldPattern)`, e.g., `set(int Line.p1)`
  Match field reference respectively assignment join points.
  E.g. `Point p = p2` or `p1 = new Point()`.

- `initialization(ConstructorPattern)`,
  e.g., `initialization(Line.new(..))`
  Matches object initialization join points.

- `staticinitialization(TypePattern)`
  e.g., `staticinitialization(Line)`
  Matches class initialization join points (as the class is loaded).

- `handler(TypePattern)`
  Matches exception handler execution by an exception type pattern.

Advice Execution Pointcuts

- AspectJ provides one primitive pointcut designator to capture execution of advice: **`adviceexecution()`**

- This can be used, e.g., to filter out any join point in the control flow of advice from a particular aspect.

```
aspect TracingAdviceExec {

      before(): adviceexecution() {
         // do some tracing
      }
  }
```

Pointcuts Selecting by Lexical Scope

- **within**(<class/aspect name>)
  Matches any join point that occurs during the execution of code contained within class or aspect name.

- **withincode**(<method/constructor signature>)
  Any join point that occurs during the execution of code contained within specified method or constructor.

```
aspect TracingAdviceExec {
      pointcut adviceToTrace():
            adviceexecution() && !within(TracingAdviceExec);

      before(): adviceToTrace() {
         // do something
      }
  }
```

Pointcuts for Object-Based Selection

```
this(<TYPE>)
target(<TYPE>)
args(<TYPE>, …)
```

- These pointcuts match any join point at which the executing object (`this`), the target object (`target`), respectively the argument objects (`arg`/`args`) is/are instances of type `TYPE`.

```
this(<typed var>)
target(<typed var>)
args(<typed var>, …)
```

- These pointcuts match any join point at which `this`, `target`, respectively `arg`/`args` is/are instances of the declared type(s).
- Further, the pointcuts bind the respective values at the matched join point to **`var`**.

```
// Matches every join point corresponding to any method
// call on an instance of Shape
pointcut shapeCall(): target(Shape);

// Same, but also binds the target shape instance to s
pointcut shapeCall(Shape s): target(s);
```

Object-Based Selection Illustrated

```
pointcut shapeMove(Shape s):
        this(s) &&
        execution(void Shape.moveBy(int, int));

after(Shape s) returning: shapeMove(s) {
        System.out.println(s.getClass() + " was moved");
}


pointcut shapeMove(Shape s, int dx, int dy):
        this(s) &&
        args(dx, dy) &&
        execution(void Shape.moveBy(int, int));

after(Shape s, int dx, int dy) returning: shapeMove(s, dx, dy) {
        System.out.println(shape.getClass() + " was moved by "
                + dx + " and " + dy);
}
```

Annotation-Based Pointcuts

```
class Point extends Shape {
    ...

    @ShapeChange
    void setX(int x){...}

    @ShapeChange
    void setY(int y){...}

}
```

```
class Line extends Shape {
    ...

    @ShapeChange
    void setP1(Point p1){...}

    @ShapeChange
    void setP2(Point p2){...}

}
```

```
after returning: execution(@ShapeChange *.*(..) {
        Display.update();
}
```

7.2.7 The Special Value thisJoinPoint

- `thisJoinPoint` can be used in advice to access different properties of the matched join point.

- Provides information about the join point, e.g.:
  - `Signature getSignature()`
  - `Object[] getArgs()`

```
before(Point p, int newVal):
        set(int Point.*)
        && target(p)
        && args(newVal)
{
        System.out.println("At " + thisJoinPoint.getSignature()
                        + " field is set to " + newVal);
}
```

7.3 Abstraction in AspectJ's Pointcuts

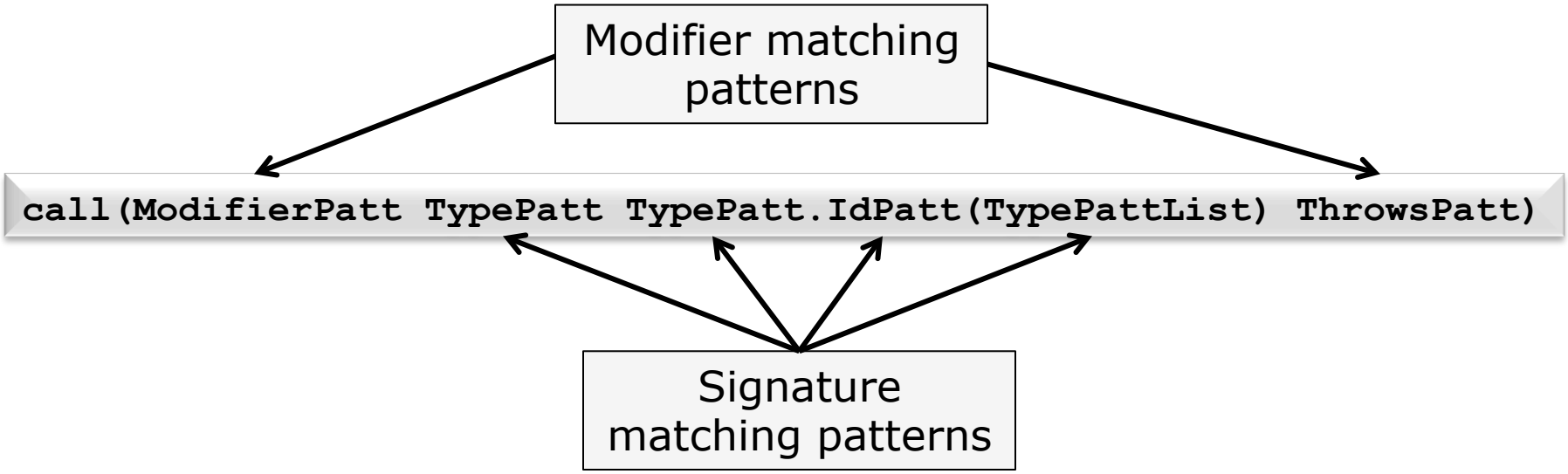
7.3 Abstraction in AspectJ's Pointcuts

- 7.3.1 Patterns in Kinded Pointcuts
- 7.3.2 Signatures of Call Join Points
- 7.3.3 Signatures of Execution Join Points
- 7.3.4 Signatures of Field Access Join Points
- 7.3.5 Join Point Modifier Patterns
- 7.3.6 Complete Rule for Join Point Matching
- 7.3.7 Wildcards in Pointcuts Patterns
- 7.3.8 Join Points in Control Flow

7.3.1 Patterns in Kinded Pointcuts

- A kinded pointcut uses patterns to abstract over specific properties of join points that participate in the implementation of a a crosscutting concern.
- Some patterns match join points based on signature.
- Some patterns match join points based on their modifiers.

Example: call pointcut



Patterns in Kinded Pointcuts

The question is what do we consider to be the signature of a join point?

A join point has potentially multiple signatures, but only one set of modifiers.

7.3.2 Signatures of Call Join Points

- Given this piece of code:

```
T t = new T();
...
t.m(parameters);  <= call join point occurs at runtime
```

- The signature of the call join point is defined as follows:

For each super-type `A` of `T`,
if `m(parameters)` is defined for that super-type, then

`R(A) A.m(parameters)` is a signature of the call join point,

where `R(A)` is the return type of `m(parameters)`
as defined in `A`,
or as inherited by `A` if `A` itself does not provide a definition of `m(parameters)`.

Signatures of Call Join Points Illustrated

- What is the signature of the call join point below

```
T t = new T();
...
t.m("hello");  <= call join point occurs when this is executed
```

- … given the following hierarchy?

```
interface Q { R m(String s); }
class P implements Q { R m(String s) {...} }
class S extends P { R' m(String s) {...} }
class T extends S {}
```

Signatures of Call Join Points Illustrated

- In the example from previous slide, the signatures for the call join point arising from the call `t.m("hello")` are:
  - **`R′ T.m(String)`**
  - **`R' S.m(String)`**
  - **`R  P.m(String)`**
  - **`R  Q.m(String)`**
- Every signature in the above list has the same id and the same parameter types, but different declaring and return types.

Rationale for Call JP Signatures

What do you think is the rationale for the definition of the call join point signatures?

- To understand the rationale underlying the call JP signatures, one needs to think in terms of pointcut matching…
- Roughly speaking:
  - given, the call join point `t.m(…)`, where `t` is of static type `T`,
  - one would like that a pointcut defined in terms of types in the definition of the method `m` in a super-type of `T` matches.
- Ultimately , `t.m(…)` is substitutable for any `a.m(…)`, where `a` is an object of type `A`, and `A` is a supertype of `T`.
- Things become more clear when we consider the rule for pointcut matching in the following slide…

Rule for Join Point Matching

A kinded primitive pointcut matches a particular join point if and only if:

- The pointcut and the join point are of the same kind
- The signature pattern of the pointcut (exactly) matches at least one signature of the join point
- The modifier patterns of the pointcut matches the modifiers of the subject of the join point

Quiz

Given the hierarchy

```
interface Q {
   R m(String s);
}
class P implements Q {
   public R m(String s) {...}
}
class S extends P {
   public R' m(String s) {...}
}
class T extends S {}
```

and the program fragment:

```
P p = new P();
S s = new S();
T t = new T();
...
p.m("hello");
s.m("hello");
t.m("hello");
```

Which calls are matched by:

(1) the pointcut `call(R P.m(String))`?

(2) the pointcut `call(R' m(String))`?

Quiz: Answer 1

```
interface Q {
   R m(String s);
}
class P implements Q {
   public R m(String s) {...}
}
class S extends P {
   public R' m(String s) {...}
}
class T extends S {}
```

```
P p = new P();
S s = new S();
T t = new T();
...
p.m("hello");
s.m("hello");
t.m("hello");
```

`call(R P.m(String))`
matches all calls.

The call to `p.m("hello")` has a single signature: `R P.m(String)`. This signature matches the pointcut's pattern.

The other calls also have a signature that matches the signature pattern of the pointcut.

Quiz: Answer 2

```
interface Q {
   R m(String s);
}
class P implements Q {
   public R m(String s) {...}
}
class S extends P {
   public R' m(String s) {...}
}
class T extends S {}
```

```
P p = new P();
S s = new S();
T t = new T();
...
p.m("hello");
s.m("hello");
t.m("hello");
```

`call(R' m(String))` matches `s.m("hello")` and `t.m("hello")`

The call to `p.m("hello")` does not have a signature that matches the return type pattern.

7.3.3 Signatures of Execution Join Points

- Signatures of method execution join points are defined in a similar way as method call join points.

- Except that only types that provide their own declarations of a method contribute signatures.

Signatures of Execution Join Points Illustrated

In the following example:

- the execution join point does not have the signature **`R' T.m(String)`**
- `T` does not provide its own declaration **`m(String)`**

```
interface Q { R m(String s); }
class P implements Q { R m(String s) {...}  }
class S extends P { R' m(String s) {...} }
class T extends S { }
class U extends T { R' m(String s) {...} }
```

```
U u = new U();
u.m("hello");  <= execution join point occurs
```

Rationale for Execution JP Signatures

What is the rationale for the execution join point signatures?

Intuitively:

A pointcut **`execution(R' T.m(String))`** should not match, when **`U.m`** is executed as the result of the call **`u.m("hello")`** below.

```
interface Q { R m(String s); }
class P implements Q { R m(String s) {...}  }
class S extends P { R' m(String s) {...} }
class T extends S { }
class U extends T { R' m(String s) {...} }
```

```
U u = new U();
u.m("hello");  <= execution join point occurs
```

7.3.4 Signatures of Field Access Join Points

- For a field get/set join point, where an access is made to a field `f` of type `F` on an object of static type `T`:
  - `F T.f` is a signature.
  - If `T` does not directly declare a field `f`:
    - for each super-type `S` of `T`, up to and including the most specific super-type of `T` that does declare the member `f`,
    - `F S.f` is a signature of the join point.

Signatures of Field Access Join Points Illustrated

Given the hierarchy and the program extract below, there are two signatures for the field get join point:

```
F T.f
F S.f
```

```
class P
{
  F f;
}

class S extends P { F f; }
class T extends S { }
```

```
T o;
...
... o.f ...
```

7.3.5 Join Point Modifier Patterns

- Every join point has a single set of modifiers which includes:
  - standard Java modifiers: public, private, static, etc.,
  - annotations,
  - throws clauses of methods and constructors.

- These modifiers are the modifiers of the **subject of the join point**.

Join Point Kind	**Subject**
Method call	The static target of the call.
Method execution	The method that is executing.
Constructor call	The constructor being called.
Constructor execution	The constructor executing.
Field get	The field being accessed.
Field set	The field being set.
Pre-initialization	The first constructor executing in this constructor chain.
Initialization	The first constructor executing in this constructor chain.
Static initialization	The type being initialized.
Handler	The declared type of the exception being handled.
Advice execution	The advice being executed.

Join Point Modifier Patterns Illustrated

- Given the following type definitions on the left-hand site:

```
public class X {
        @Foo
         protected void doIt() {...}
}
public class Y extends X {
         public void doIt() {...}
}
```

```
Y y;
X x;
...
y.doIt()
x.doIt();
```

- What are the modifiers for the calls on the right-hand site?

Answer

- The modifiers for the call to `y.doIt()` are {`public`}.
- The modifiers for the call to `x.doIt()` are {`@Foo, protected`}.

7.3.6 Complete Rule for Join Point Matching

A kinded primitive pointcut matches a particular join point if and only if:

- The pointcut and the join point are of the same kind
- The signature pattern of the pointcut (exactly) matches at least one signature of the join point
- The modifier patterns of the pointcut matches the modifiers of the subject of the join point

Quiz

Given the hierarchy

```
interface Q {
   R m(String s);
}
class P implements Q {
   @Foo
   public R m(String s) {...}
}
class S extends P {
   @Bar
   public R' m(String s) {...}
}
class T extends S {}
```

and the program fragment:

```
P p = new P();
S s = new S();
T t = new T();
...
p.m("hello");
s.m("hello");
t.m("hello");
```

Which calls are matched by `call(@Foo R P.m(String))`?

Quiz: Answer

```
interface Q {
   R m(String s);
}
class P implements Q {
   @Foo
   public R m(String s) {...}
}
class S extends P {
   @Bar
   public R' m(String s) {...}
}
class T extends S {}
```

```
P p = new P();
S s = new S();
T t = new T();
...
p.m("hello");
s.m("hello");
t.m("hello");
```

`call(@Foo R P.m(String))`
matches only `p.m("hello")`

The call to `p.m("hello")` has a single signature: `R P.m(String)`.
This signature matches the pointcut's pattern.
The modifier of its subject (the declaration of `m` in `P`) also matches the modifier pattern (both are `@Foo`).

The other calls have a signature that matches the signature pattern of the pointcut but the modifiers of their subjects do not match.

7.3.7 Wildcards in Pointcuts Patterns

- AspectJ supports abstracting from the concrete appearance of join points in pointcut specifications by means of wildcards*)
- Allows property-based selection of join points.

  - "`*`" is wildcard (matches one arbitrary element).
  - "`..`" is multipart wildcard (matches any number of arbitrary elements).
  - "`+`" all subclasses (only usable with types).
  - "!" is negation.

Pointcut	Matches
`execution(* *(..))`	Execution of any method
`execution(public * Point.*(..))`	Execution of any public method of `Point`
`execution(private * Shape+.*(..))`	Execution of every private method of `Shape` and its subclasses
`execution(public !static * *(..))`	Execution of any public, non static method

*) Note: Not equal to regular expressions! E.g. "Sh*p*" not possible.

An Example Aspect Using Wildcards

```
aspect DisplayUpdating {

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

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

The Need to Filter by Control Flow

```
aspect DisplayUpdating {

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

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

Do you see any problem with the aspect?

```
...
class Line implements Shape {
        ...
        public void moveBy(int dx, int dy) {
                p1.moveBy(dx, dy);
                p2.moveBy(dx, dy);
        }
}
```

The Need to Filter by Control Flow

```
aspect DisplayUpdating {

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

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

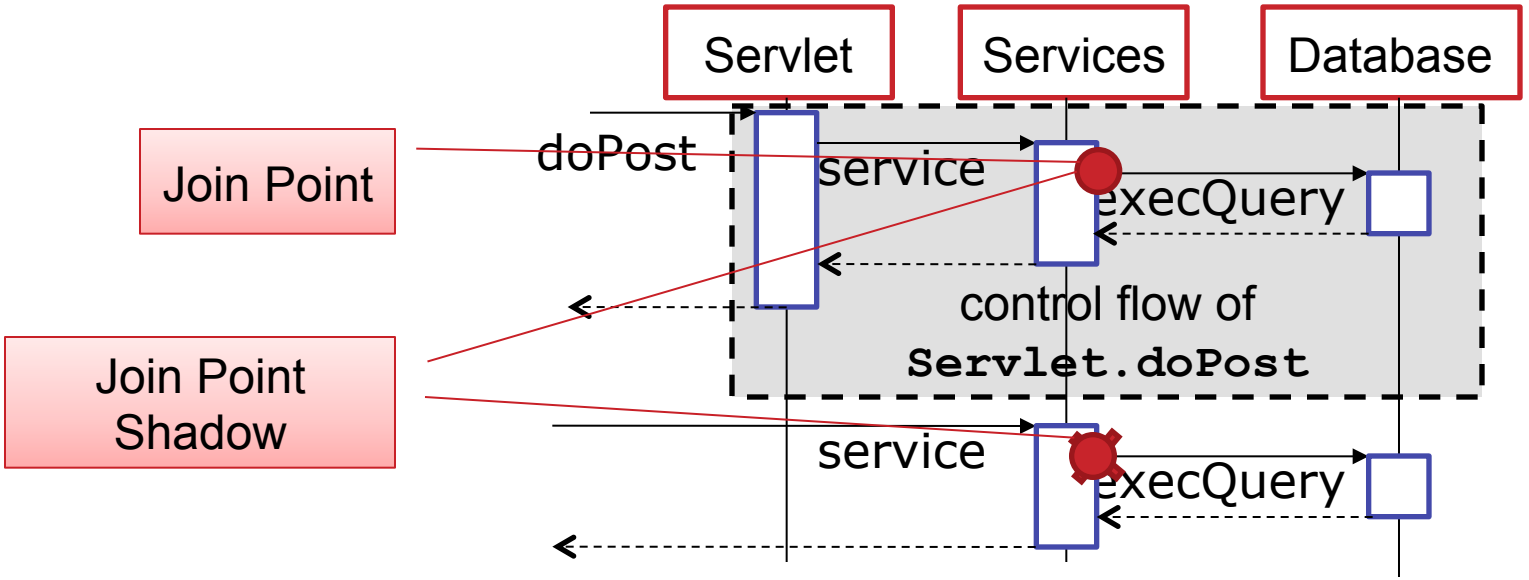
- In this example, the advice `after(Shape shape) returning` will be executed three times, even though one time would suffice.
- Even worse, scenarios with endless loops are possible!

- To solve such problems, we need to use the control flow as a filter when selecting join points.

The Need to Filter by Control Flow

When: about to send query to database
in control flow of `Servlet.doPost`
What: sanitize query

Description of **Join Points**
Advice **Action**
Advice Unit

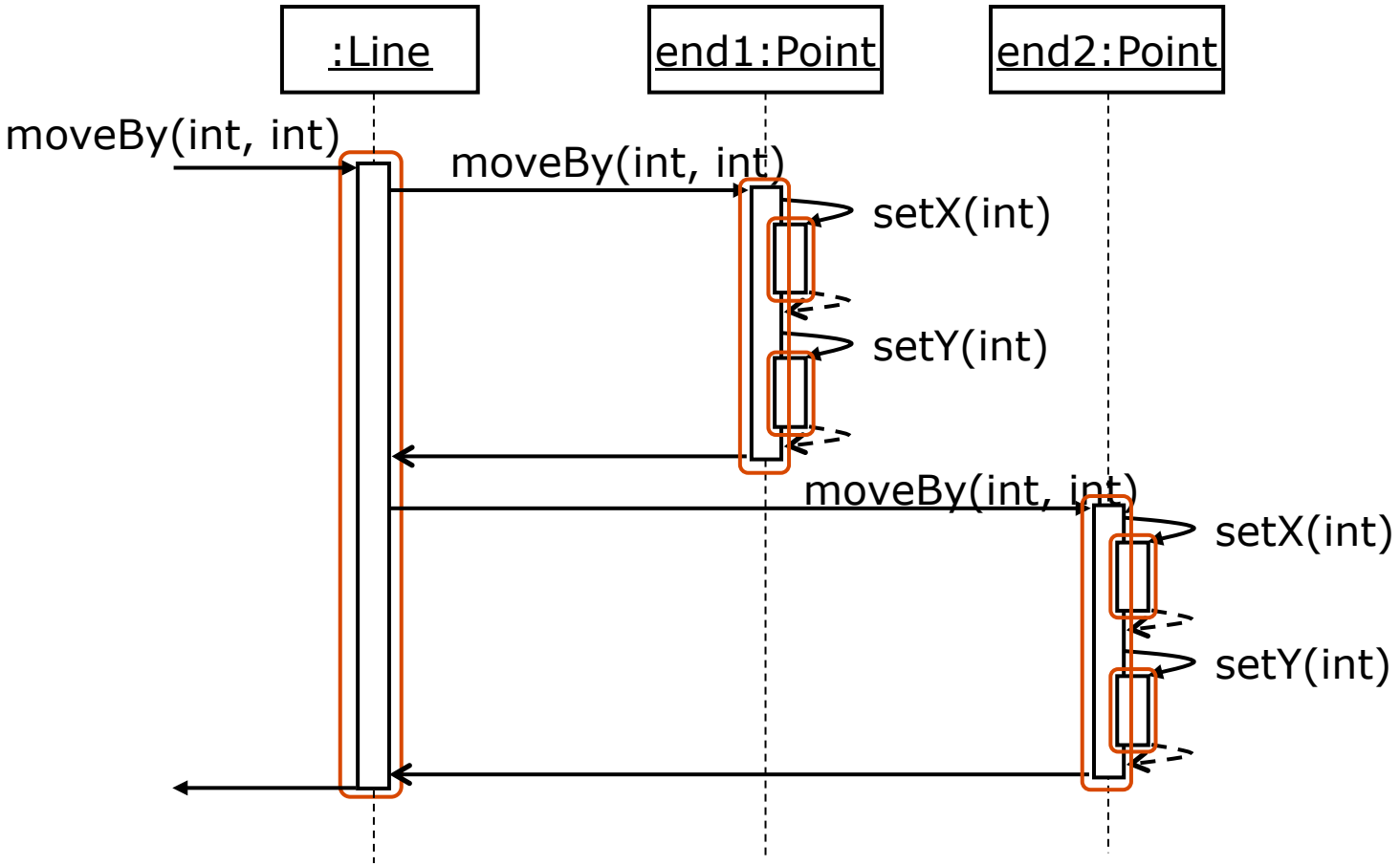


Pointcuts for Selecting Based on Control Flow

- **`cflow`**`(<pointcut designator>)`
  Matches all join points within the dynamic control flow of any join point in `<pointcut designator>`.

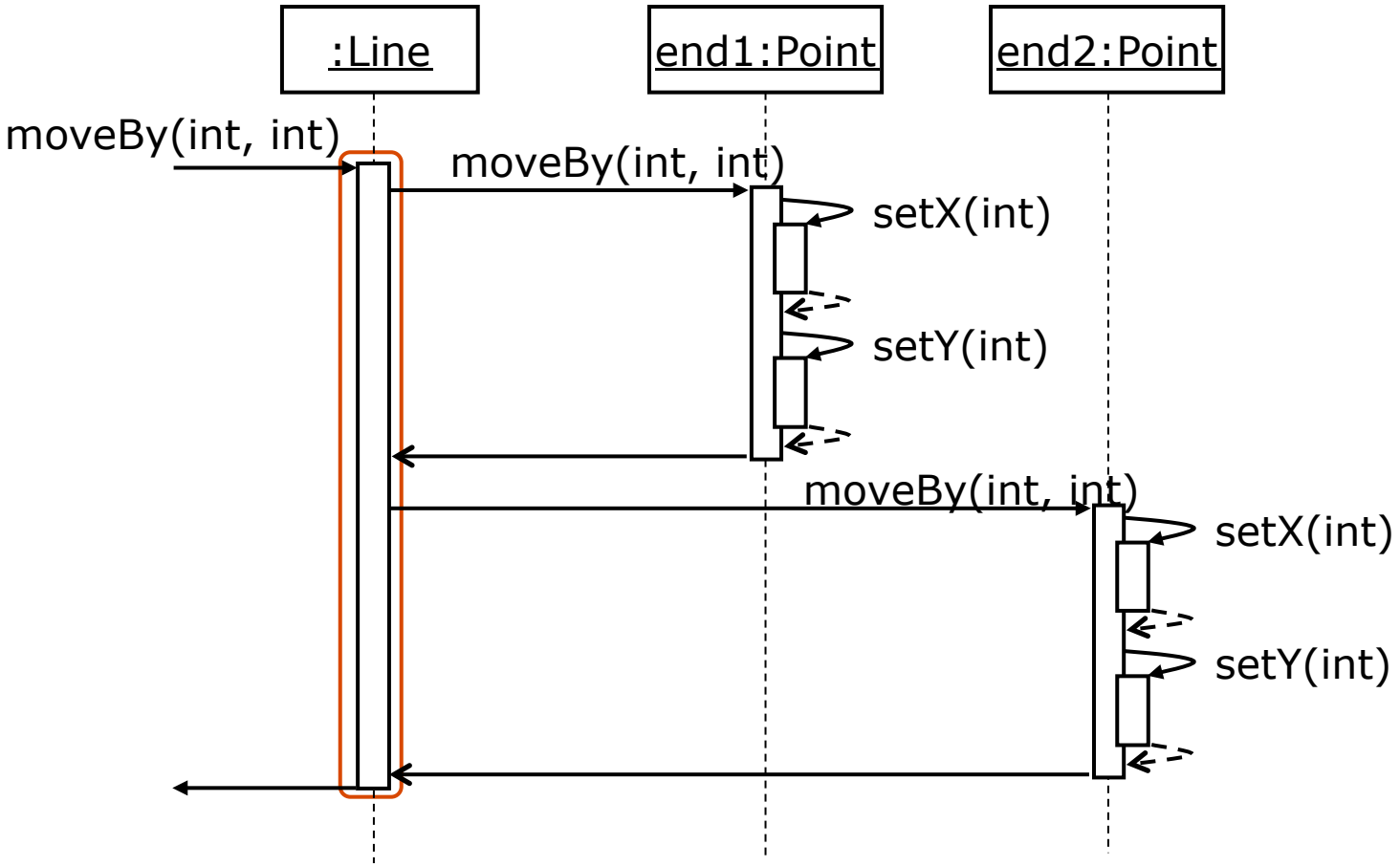
- **`cflowbelow`**`(<pointcut designator>)`
  Matches all join points within the dynamic control flow below any join point in `<pointcut designator>`.

Method Execution Join Points



Join points in **`shapeChange`**

Method Execution Join Points

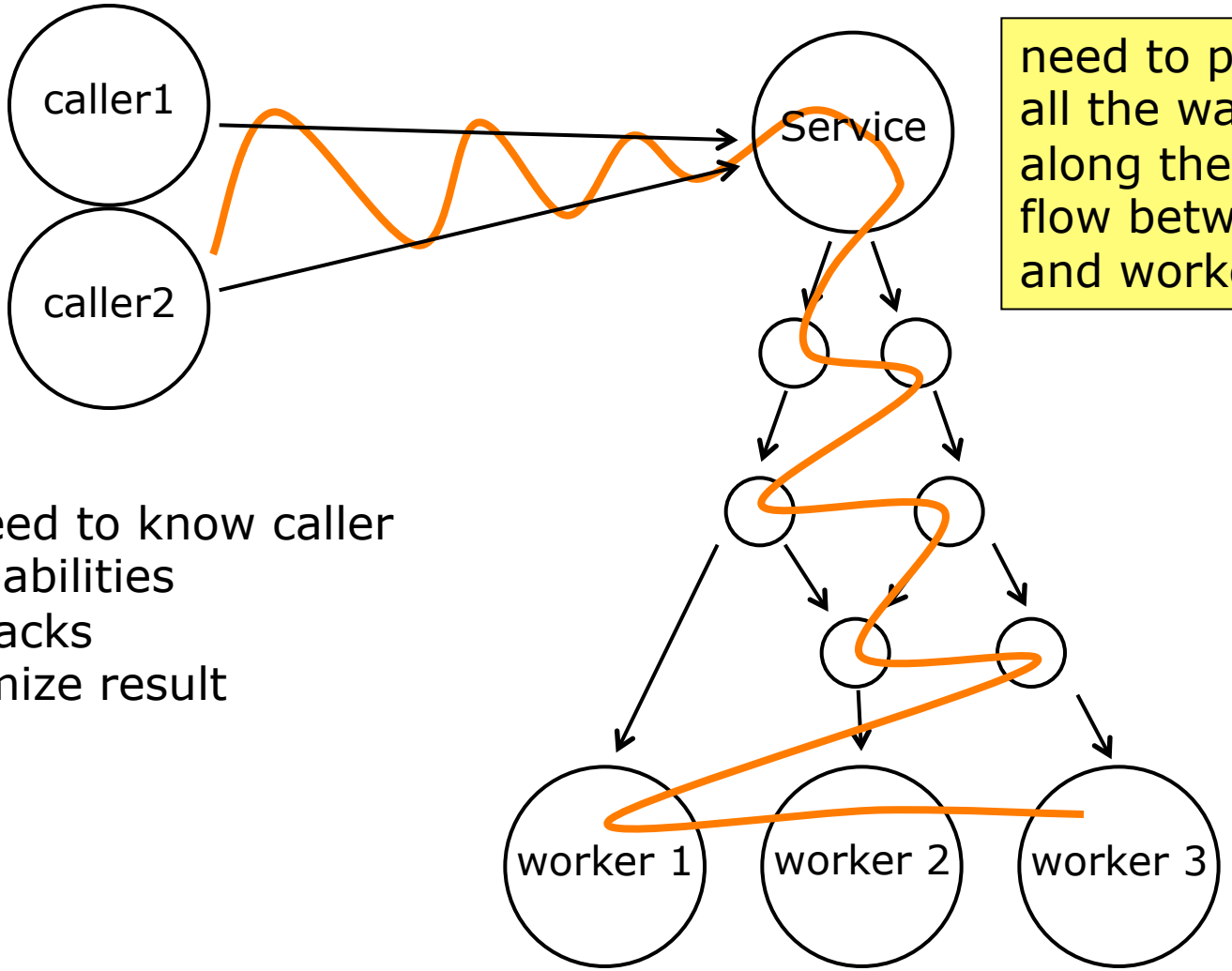


Join points in **`shapeChange`**
and not in `cflowbelow(shapeChange)`.

Avoiding Repetitive Display Updates

```
aspect DisplayUpdating {

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

        after(Shape shape) returning: shapeChange(shape)
                                      && !cflowbelow(shapeChange()) {
                Display.update(shape);
        }
}
```

Context Passing Aspects

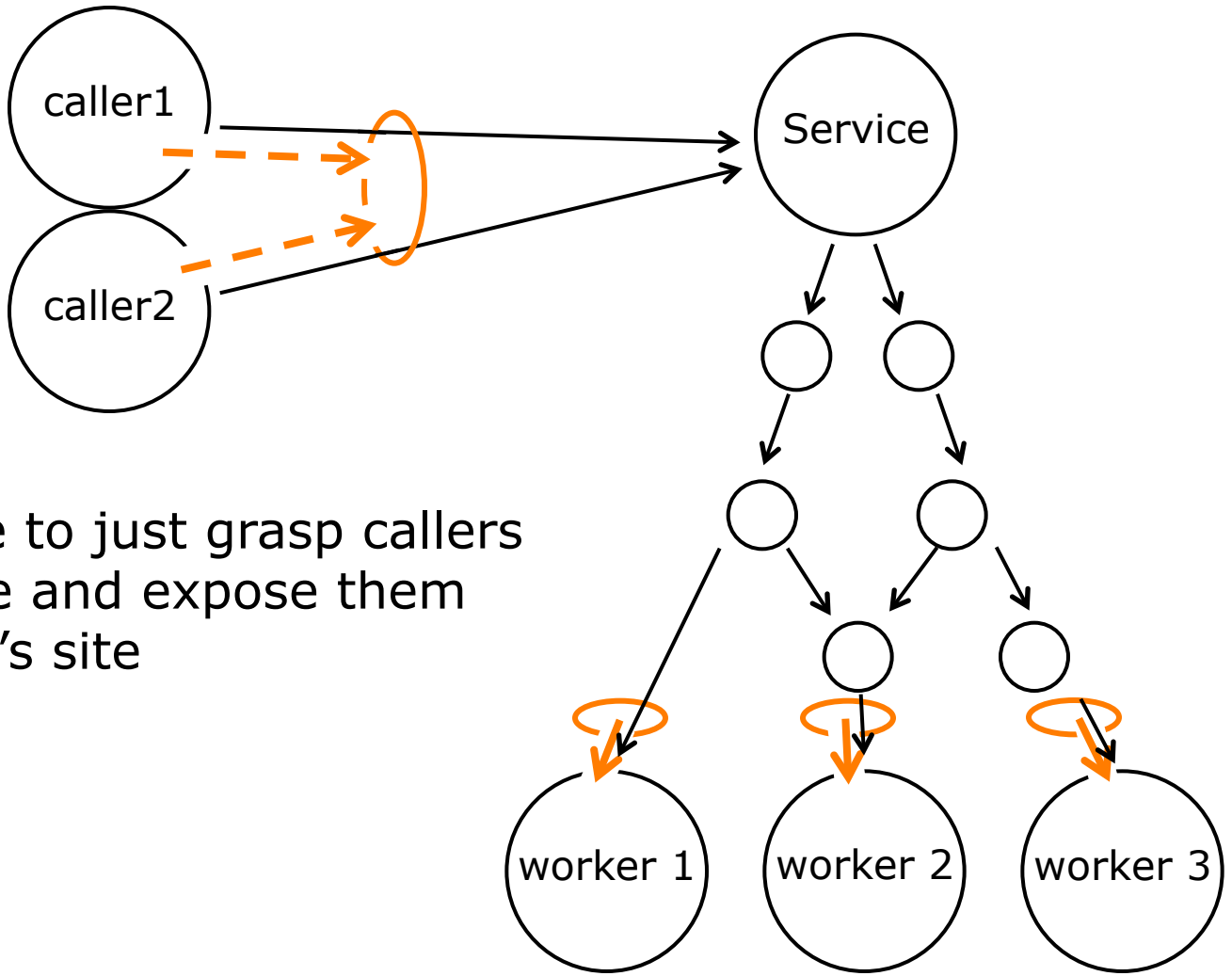


need to pass caller all the way down along the control flow between caller and workers ☹

workers need to know caller and its capabilities

- charge backs
- to customize result

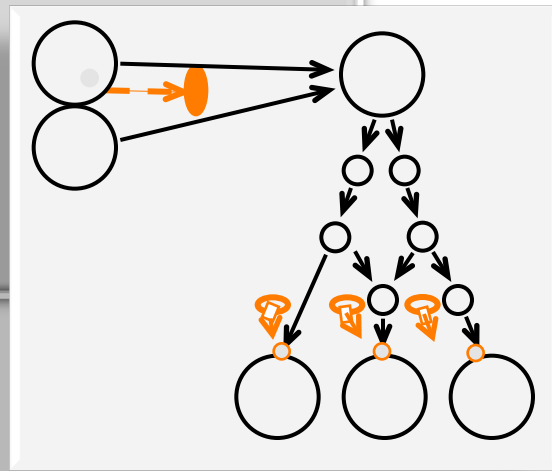
Context Passing Aspects



would like to just grasp callers at call site and expose them at worker's site

Context Passing Aspects

```
aspect CapabilityChecking {

  pointcut invocation(Caller c):
    this(c) && call (void Service.doService(String));

  pointcut workPoint( Worker w):
    call(void doTask(Task)) && target(w);

  pointcut perCallerWork(Caller c, Worker w):
    cflow (invocation(c)) && workPoint(w);


  before (Caller c, Worker w):
                    perCallerWork(c, w) {
        w.checkCapabilities(c);
  }
}
```



7.4 Reflections on Modularity

7.4.1 Case-Study: Display Updating

```
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;

  }
}
```

OO Display Updating V1

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

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;
  }
}
```

OO Display Updating V2

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

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

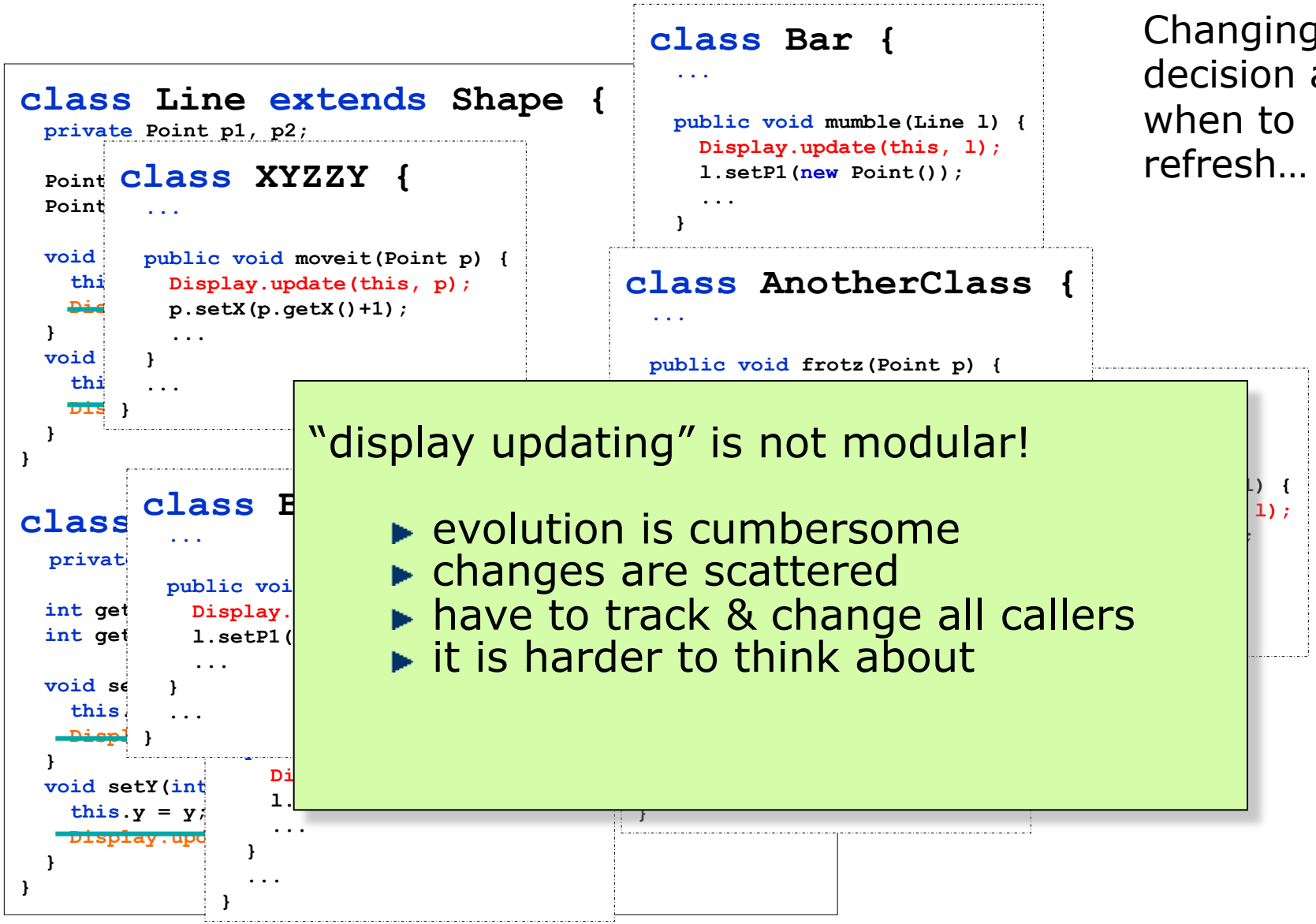
Changing the decision about what constitutes change…

OO Display Updating V3

```
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);
  }
}
```

Changing the decision about the context to pass from change to refreshing…

OO Display Updating V4



"display updating" is not modular!

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

Changing the decision about when to refresh...

AO Display Updating

```
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;
  }
}
```

AO Display Updating V2

```
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 DisplayUpdating {

  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();
  }
}
```

AO Display Updating V2.5

```
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 DisplayUpdating {

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

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

Abstracting over the execution… not possible in OO. The **interface** between both concerns becomes **thinner**!

AO Display Updating V3

```
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 DisplayUpdating {

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

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

AO Display Updating V4

```
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 DisplayUpdating {

  pointcut change(Object mover, Shape movee):
    this(mover)       &&
    target(movee)     &&
    (call(void Shape.moveBy(int, int)) ||
     call(void Shape+.set*(..)));

  after(Object mover,Shape movee)
                          returning: change(mover, movee) {
    Display.update(mover, movee);
  }
}
```

Summary of the Case Study

- The aspect is localized and has a clear interface
- The classes are better localized (no invasion of updating)

		localized	interface	abstraction	enforced
non AOP	display updating	no	n/a	n/a	n/a
	Point, Line	medium	medium	medium	yes
AOP	Display Updating	high	high	medium	yes
	Point, Line	high	high	high	yes

Seeing the Forest versus Seeing the Trees

- Global invariant is explicit, clear, modular
  - can reason about it in a modular way
  - can change it locally
  - can plug it in and out
  - the compiler can implement it more efficiently

- Yet, local effects can be made clear by IDE

AJDT

7.4.2 Some Usage Guidelines

Pointcut-Advice Instead of Method Calls

- If some functionality is needed at a scattered set of points in the execution of a program, then a pointcut/advice makes:
  - the set of points local and explicit.
  - the binding from that set to the functionality to execute local and explicit.
- Improves comprehensibility and evolution.
- Consider using pointcut-advice instead of multiple calls to an operation if:
  - More than a small number of points must invoke the operation.
  - The binding between the points and the operation may be disabled or may be context-sensitive.
  - The calling protocol to the operation may change.

Enumeration Versus Pattern-Based Pointcuts

- Enumeration: Explicitly declare every point per name in the pointcut.
- Prefer when:
  - It is difficult to write a stable property-based pointcut.
  - The set of points is relatively small.

- Pattern-based: Use wildcards in pointcuts.
- Prefer when:
  - It is possible to write one that is stable.
  - The set of points is relatively large (more than ten).

To Annotate or not to Annotate

- Use annotations if:
  - It is difficult to write a stable property-based pointcut.
  - Annotation expresses an inherent domain property rather than a context-dependent aspect of the join points that is only true in some configurations.

7.5 Inter-Type Declarations

7.5 Inter-Type Declarations

- 7.5.1 Introduction to Inter-Type Declarations
- 7.5.2 ITD Illustrated by Example
- 7.5.3 Problems with AspectJ‘s ITDs

7.5.1 Introduction to Inter-Type Declarations

- There are two kinds of crosscutting structures: dynamic and static.

- Dynamic crosscutting is expressed via pointcut/advice.
  - Adds functionality to the execution of an application.
  - Adds per-aspect state.
  - Acts at runtime.

- **Static crosscutting** via Inter-type declarations (ITDs).
  - Adds new methods to existing classes.
  - Adds new attributes to existing classes.
  - Changes inheritance relations.
    E.g. make an existing class implement a certain interface.
  - **Acts at compile time**.

Overview of ITDs in AspectJ

- ITDs are declared inside aspects.
- Some of ITDs supported by AspectJ are shown in the following.

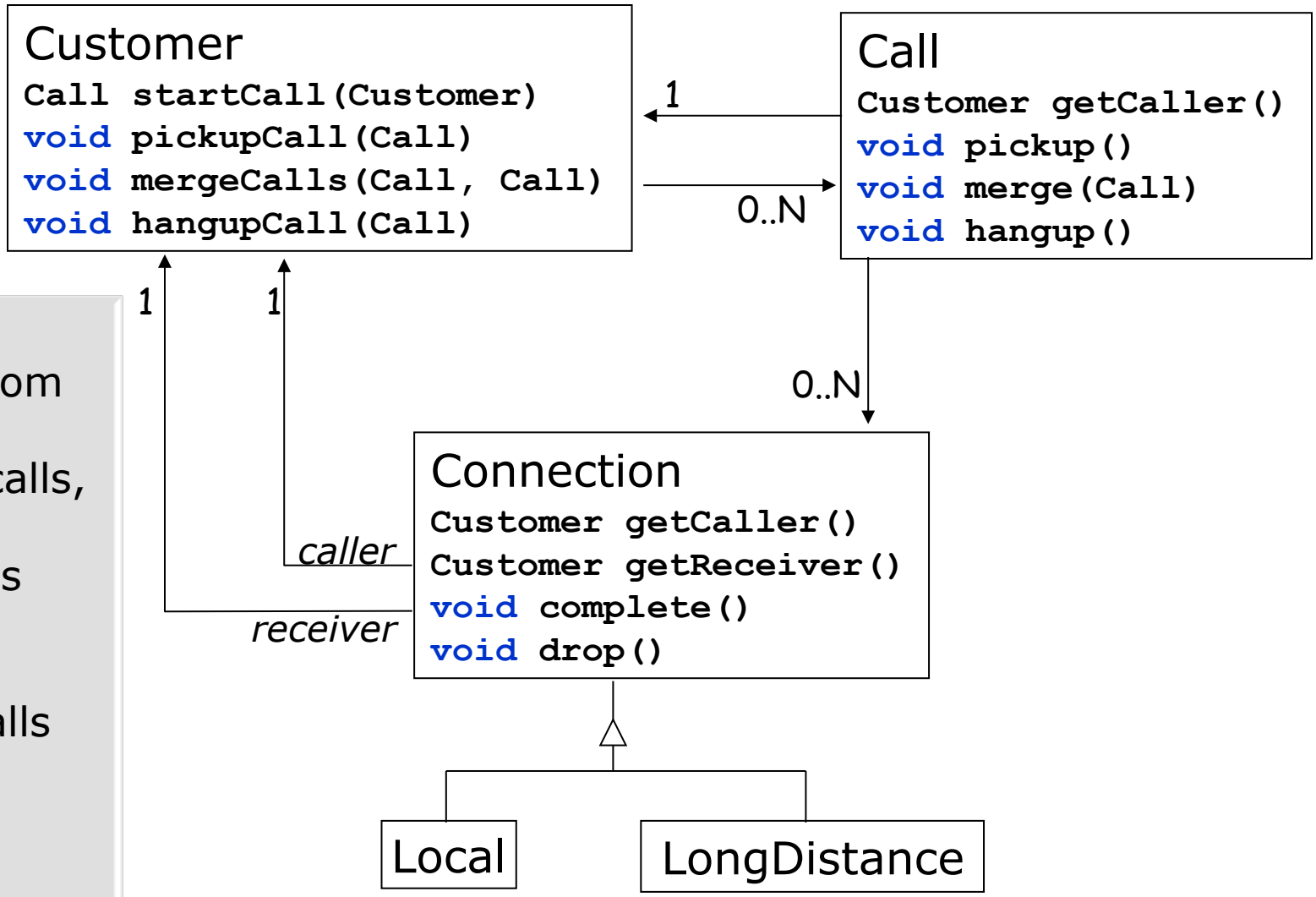
`public <type> <class>.<name>;`
Adds the public attribute `<name>` of type `<type>` to the existing class `<class>`.

`private <type> <class>.<name>;`
Adds the private (means only accessible inside the aspect) attribute `<name>` of type `<type>` to the existing class `<class>`.

`public <returntype> <class>.<name>(<args>);`
Adds the public method `<name>` with return type `<returntype>` and arguments `<args>` to the existing class `<class>`.

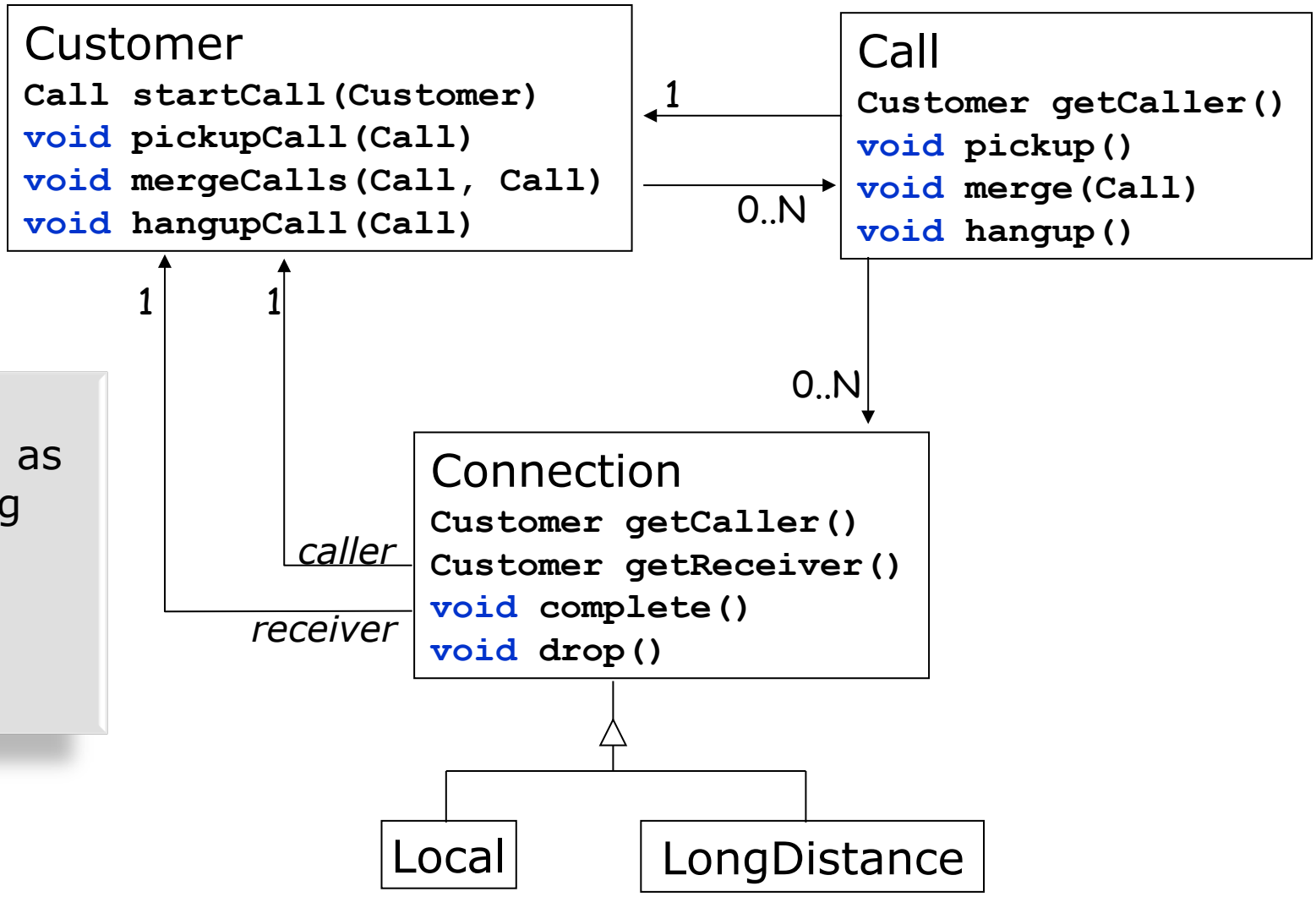
`declare parents: <class> implements <interface>;`
Makes the existing class `<class>` implement the interface `<interface>`.
If you use this, you must ensure that the methods declared by `<interface>` are present in `<class>` or you have to add them yourself via ITDs.

7.5.2 ITD Illustrated by Example



Given:
A basic telecom design, with customers, calls, connections. These classes define the protocol for setting up calls (includes conference calling) and establishing connections

ITD Illustrated by Example

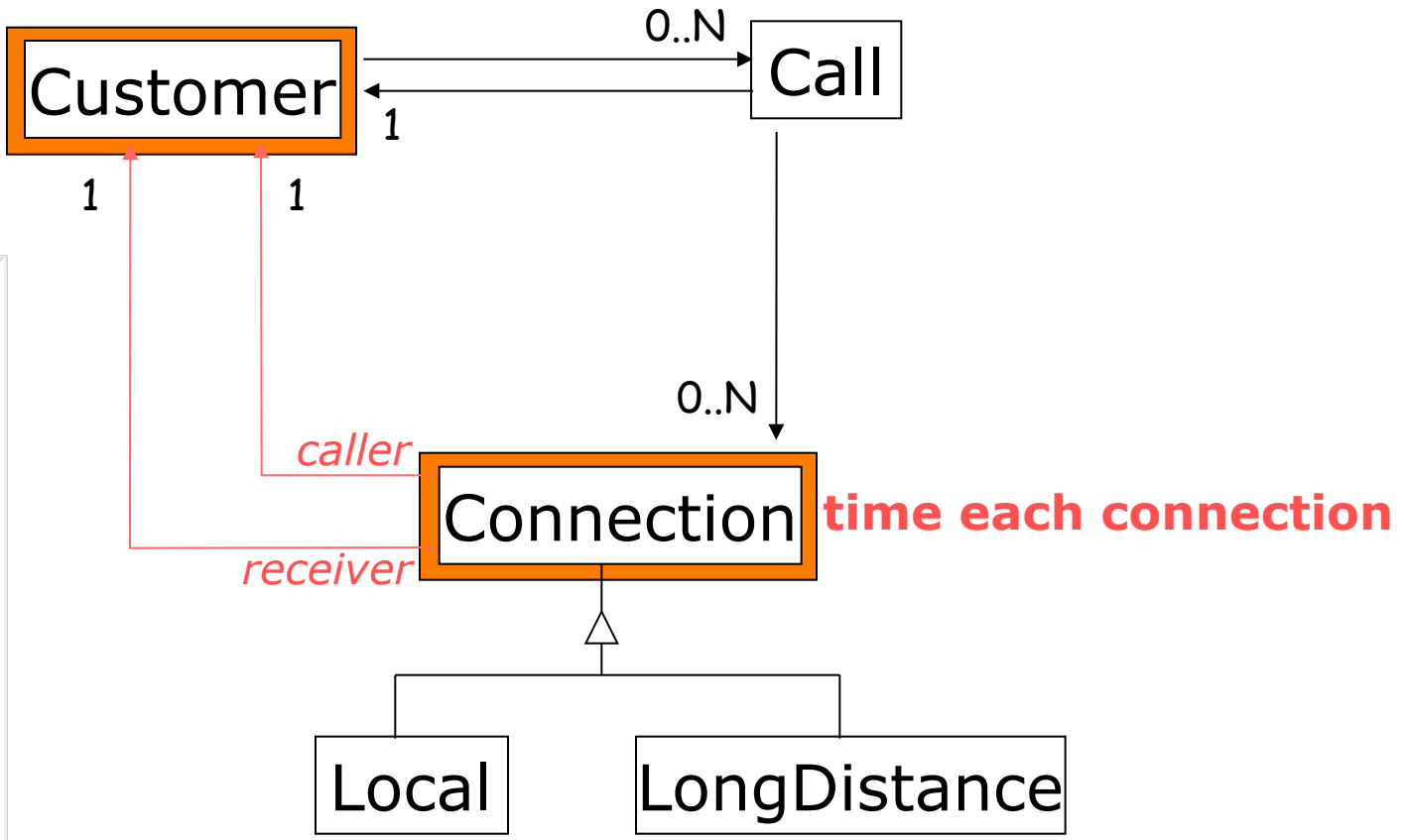


Needed:
Utilities such as timing, billing (depends on timing), consistency checks etc.

Adding Timing Functionality

Adding timing to the telecom design has non-local effects.

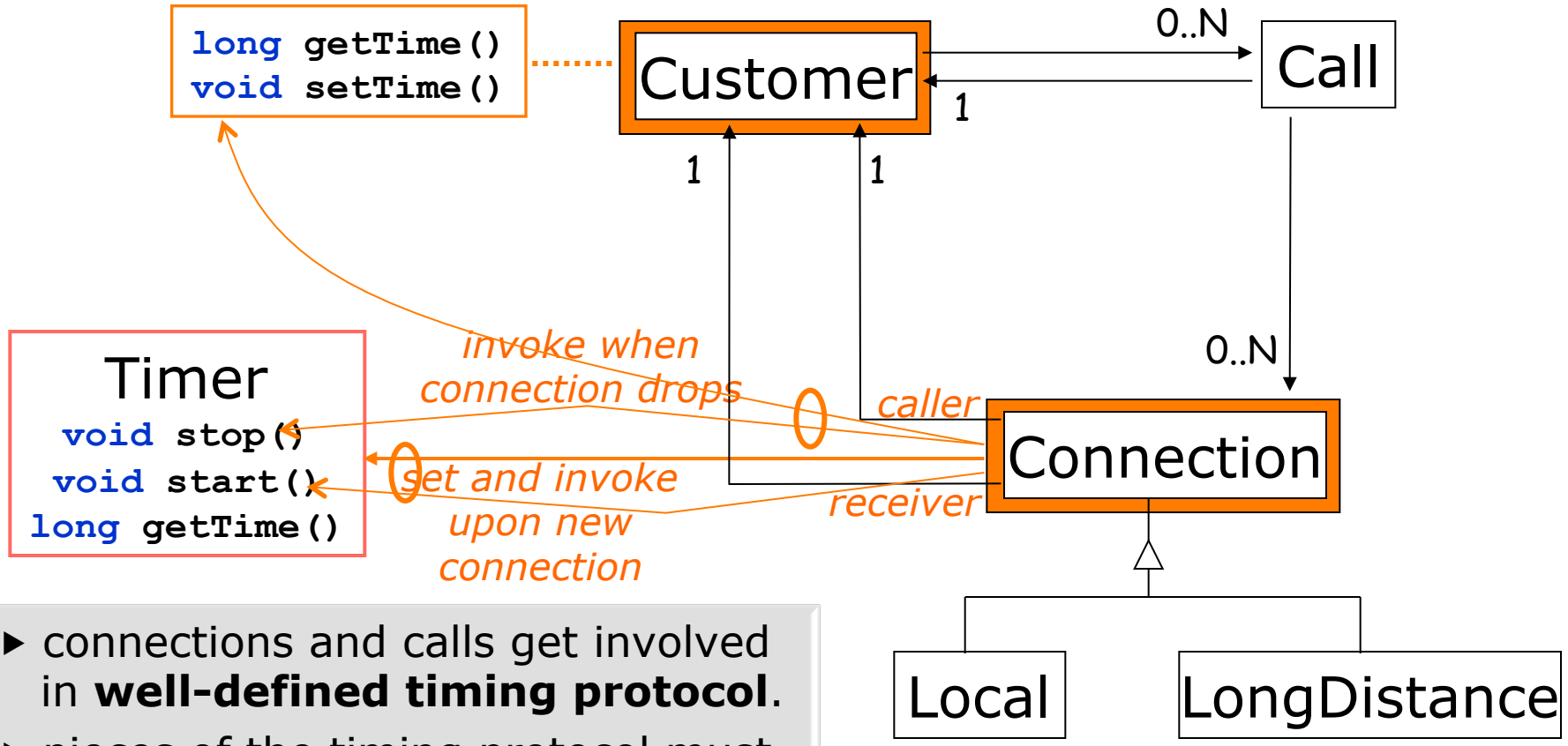
store total connection time



- **Timing-specific state**
- `Connection` should record the duration of calls.
- `Customer` should store the total connection time.

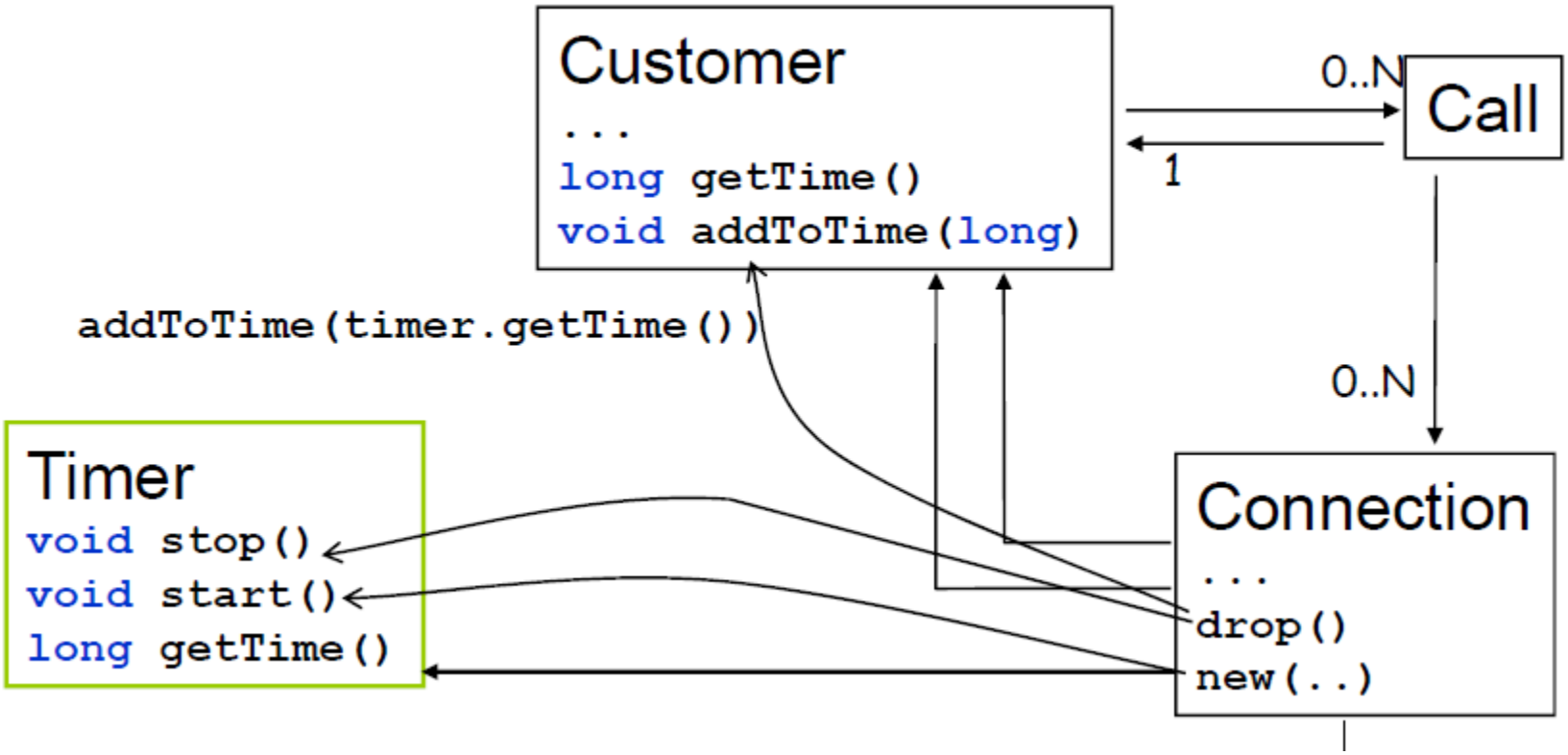
Adding Timing Functionality

Adding timing to the telecom design has non-local effects.



- connections and calls get involved in **well-defined timing protocol**.
- pieces of the timing protocol must be triggered by the execution of certain basic operations.

Timing is Hard to Modularize in OO



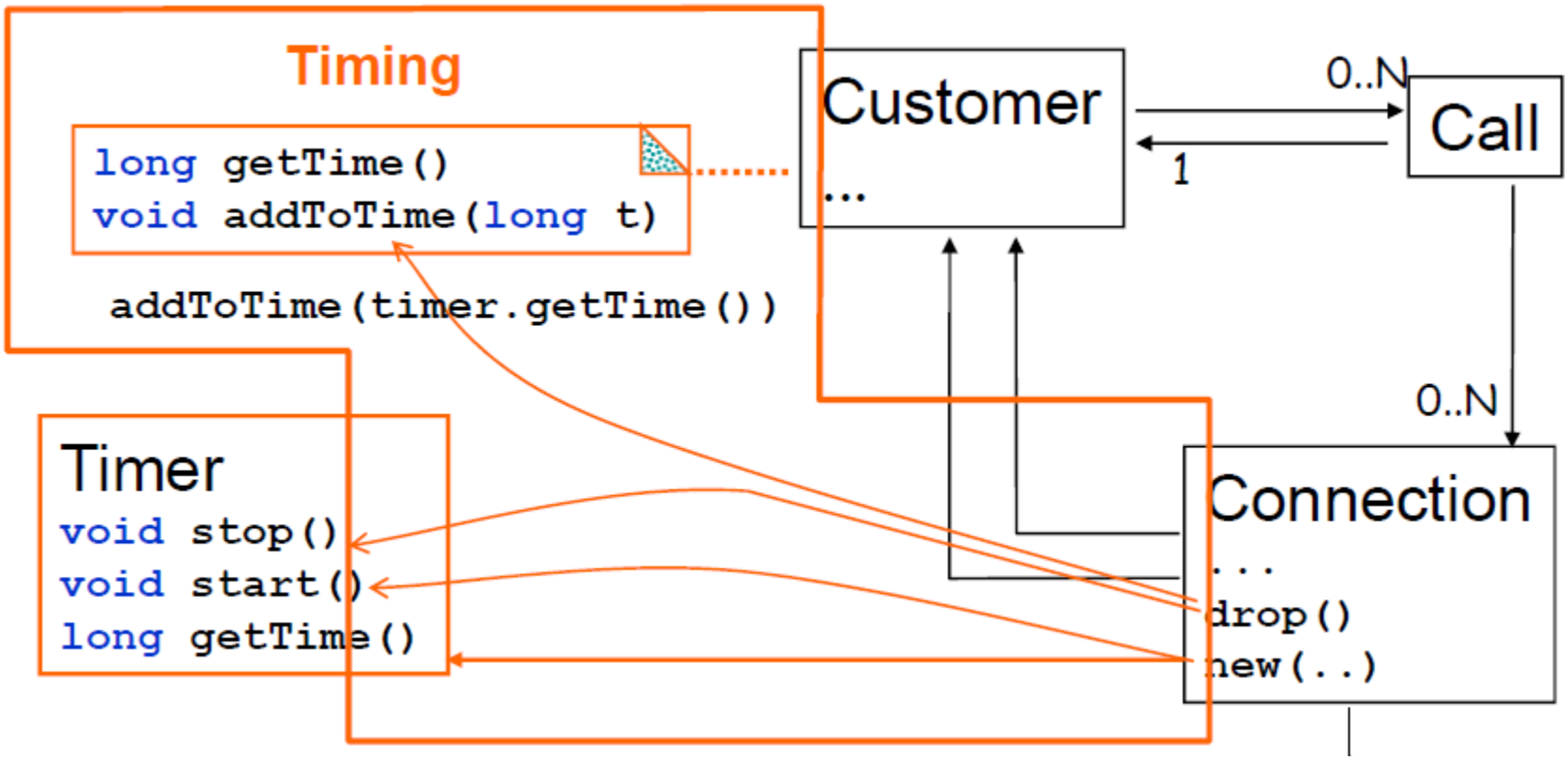
- One can encapsulate issues of measuring time in a (`Timer`) object.
- But, such an object **does not capture the protocols involved in implementing the timing feature.**
- Encoding the timing protocol needs changes in several classes in current design: `Customer` and `Connection`.

Motivation for Using Aspects for Modeling Timing

- There are good reasons for not changing `Customer` and `Connection`.
- They might be members of closed modules.
- Even if they can be changed,
  - Making timing built in part of `Customer` and `Connection` makes it hard to make timing optional (switch on/off in different products for different customers)
  - Makes it harder to test and evolve independently of core functionality.
  - Violation of SRP, OCP.

- We have seen that incremental programming via inheritance is not effective either, when several related classes are affected.

Timing as an Aspect



- Timing as an aspect captures both the state and the protocols involved in implementing the timing feature.
- It also "adds" links to the timing state to existing classes in a non-invasive way.

Timing as an Aspect

```
aspect Timing {
    // Add attribute timer and respective accessors to Connection
    private Timer Connection.timer = new Timer();
    public Timer Connection.getTimer() { return timer;}

    // Add total connection time and respective accessors to Customer
    private long Customer.totalConnectTime = 0;
    public void Customer.addToTime(long time) { totalConnectTime += time; }
    public long Customer.getTotalConnectTime() { return totalConnectTime; }

    pointcut startTiming(Connection c): target(c)
                                    && call(void Connection.complete());

    pointcut endTiming(Connection c): target (c)
                                    && call(void Connection.drop());

    after (Connection c): startTiming(c) {c.getTimer().start();}

    after(Connection c): endTiming(c) {
        Timer timer = c.getTimer();
        timer.stop();
        c.getCaller().addToTime(timer.getTime(););
    }
}
```

Benefits of Timing as an Aspect

- No need to change existing classes.
- Basic objects are not responsible for using the timing facility.
  Timing aspect encapsulates that responsibility for appropriate objects.
- If requirements for timing facility change, that change is shielded from the base functionality; only the timing aspect is affected.
- Adding/removing timing from the design is trivial.
  Add/remove the timing aspect from configuration files.

7.5.3 Problems with AspectJ's ITDs

- Intertype declarations are **globally visible**
  - The introduced members can be accessed without importing the aspect
  - Create **implicit dependencies** between modules
    - Difficult to analyze dependencies in a program
    - Removing or changing an aspect may destabilize other modules
    - Incompatible with independent compilation

- Independently developed aspects may contain introductions with **clashing names**
  - Prohibits independent evolution

Problems with Global Visibility of ITDs Illustrated

```
aspect Timing {
  ...
  // Visible to all modules importing Customer
  public void Customer.addToTime(long time) { totalConnectTime += time; }
  public long Customer.getTotalConnectTime() { return totalConnectTime; }
  ...
}
```

```
import telecom.Customer;

class Billing {
   public static float bill(Customer c) {
     float toCharge = c.getTotalConnectTime() *
pricePerMin;
     ...
   }
   ...
}
```

- Difficult to detect that **`Billing`** depends on **`Timing`** and how
- Change to **`Timing`** may break **`Billing`**
- Compilation dependencies of **`Billing`** are not known – only global compilation possible

Private Introductions

```
aspect Timing {
  // Visible only within Timing
  ...
  private Timer Connection.timer = new Timer();
  ...
  private long Customer.totalConnectTime = 0;
  ...
}
```

- Private introductions are visible to the introducing aspect only
- Pros:
  - avoid implicit dependencies
  - avoid name clashes
- Cons:
  - cannot be used by other modules, even not in inheriting aspects
  - private methods cannot be overridden

Workaround: Avoid Non-Private Introductions

```
aspect Timing {
  ...
  private long Customer.totalConnectTime = 0;
  public static void addToTime(Customer c,long time) {c.totalConnectTime += time;}
  public static long getTotalConnectTime(Customer c) {return c.totalConnectTime;}
  ...
}
```

```
import telecom.Customer;
import Timing;

class Billing {
   public static float bill(Customer c) {
     float toCharge = Timing.getTotalConnectTime(c) * pricePerMin;
     ...
   }
   ...
}
```

Enforces explicit dependency of **Billing** on **Timing**

- But prohibits advantages of object-orientation
- Cannot introduce late-bound methods in this way
- No explicit interface of introduced functionality, e.g., billing cannot abstract from variations in the implementation of timing

7.6 Aspect Inheritance and Instantiation

7.6 Aspect Inheritance and Instantiation

- 7.6.1 Default Instantiation Semantics
- 7.6.2 Non-Default Instantiation Semantics
- 7.6.3 Instantiation Semantics Illustrated
- 7.6.4 Accessing Aspect Instances
- 7.6.5 Limitations of Aspect Instantiation in AspectJ
- 7.6.6 Inheritance & Specialization

7.6.1 Default Instantiation Semantics

- The state of the crosscutting functionality can be kept in variables declared in an aspect.
- This state can be used in advices, but not in pointcuts.

- Aspects are **singletons** by default.
  - One aspect of a certain type per virtual machine.
  - The state is globally shared.

7.6.2 Non-Default Instantiation Semantics

- Other aspect-instance relations are possible by explicit declaration.
  - One aspect **per object**
  - One aspect **per control-flow**

- Declaration

```
aspect AspectName per...(<pointcut>)
```

Non-Default Instantiation Semantics

Per control-flow

```
aspect AspectName percflow(<pointcut>)
aspect AspectName percflowbelow(<pointcut>)
```

- One aspect instance for each jointpoint matched by <pointcut>
- The aspect is active only within the control flow of the **<pointcut>**
- The aspect's pointcuts are filtered by **cflow(<pointcut>)** (or **cflowbelow(<pointcut>)**)

Per object

```
aspect AspectName pertarget(<pointcut>)
aspect AspectName perthis(<pointcut>)
```

one aspect instance for each different "this"/"target" of a joinpoint matched by <pointcut>

7.6.3 Instantiation Semantics Illustrated

Consider account management in a banking software…

```
class Account {
  int accountNo;
  void debit(float amount) {
     float balance = getBalance();
     if (balance > amount) {
        setBalance(balance – amount);
     }
  }
  void setBalance(float balance) {
    Statement stm = getConnection().createStatement();
    stm.executeUpdate("update accounts set balance = " + balance
                  + "where accountNumber = " + _accountNumber);
    ...
  }
  float getBalance();
  ...
}
```

Instantiation Semantics Illustrated

- New functionality needs to be added to the account management software.
  - Need to count the number of database accesses performed by every account operation for profiling purposes.
  - Need to count the number of financial transactions performed on each account in order to charge after exceeding free operation limit.

- Will use aspects for modularizing the new functionality:
  - **`AccountOpProfiling`**
  - **`AccountCharging`**

Which instantiation semantics would you propose for each of these aspects?

Per-Control-Flow Instantiation Illustrated

An aspect for counting database accesses performed by every account operation…

```
aspect AccountOpProfiling percflow(accountOp()) {

  pointcut accountOp(): execution(* Account.*(..));
  pointcut databaseOp(): call(* java.sql..*.*(..));
                         // && cflow(accountOp())

  // state local to a profiled operation
  int accessCount = 0;

  // using local state in advice
  before(): databaseOp() { accessCount ++; }
  after(): accountOp() {
     System.out.println(thisJoinPoint.getSignature()
               + " accessed DB " + accessCount + " times");
  }
}
```

Per-Object Instantiation Illustrated

An aspect for counting operations on every account object, charge after exceeding free operation limit

```
aspect AccountCharging perthis(accountOp()) {

  pointcut accountOp(Account acc): execution(* Account.*(..));

  // additional state associated to an Account
  int accessCount = 0;
  int freeLimit = 10;

  // using local state in advices
 before(): accountOp(acc) {
     accessCount++;
     if (accessCount > freeLimit) { acc.chargeFee(); }
  }
}
```

7.6.4 Accessing Aspect Instances

- Singleton aspect instance can be retrieved by **`AspectName.aspectOf()`**

- Aspect instance associated to an object can be retrieved by **`AspectName.aspectOf(obj)`**

```
void changeFreeLimit(Account acc) {
   AccountCharging.aspectOf(acc).freeLimit = 0;
}
```

- For aspects associated with a control flow, during each such flow of control, the static method A.aspectOf() will return an object of type A.
- An instance of the aspect is created upon entry into each such control flow. "

7.6.5 Limitations of Aspect Instantiation in AspectJ

- One-to-one relation between an aspect instance and an object
  - Cannot have multiple instances of an aspect for an object
  - Cannot associate an aspect to a combination of objects, e.g. a pair of an account and a session
- The aspect scope is limited to the control flow of an object
  - The aspect associated to an object cannot intercept joinpoints outside the execution on the object
- Cannot control dynamic aspect instantiation depending on runtime conditions e.g., dynamically turn on and off fee charging on selected accounts.

7.6.6 Inheritance & Specialization

- Just as Interfaces and Classes, Aspects may:
  - Be defined abstract.
  - Be extended.

- Pointcuts may be defined abstract too.
  - Abstract from the joinpoints affected by of an aspect
- Abstract pointcuts can be specialized by extending aspects.

- Advice on abstract pointcuts:
  - Is inherited by extending aspects.
  - Runs on join points matched by concrete pointcut.

- Facilitates reuse of aspect functionality
  - Reuse the state and operations of an aspect
  - Reuse pointcut definitions
  - Reuse pieces of advice on different pointcuts

Example: Reusable Database Access Profiling

```
aspect AccountOpProfiling percflow(accountOp()) {
  pointcut accountOp(): execution(* Account.*(..));
  pointcut databaseOp(): call(* java.sql..*.*(..));
  int accessCount = 0;

  before(): databaseOp() { accessCount ++; }
  after(): accountOp() {
     System.out.println(thisJoinPoint.getSignature()
                        + " accessed DB " + accessCount +
  " times");
   }
}
```

We may move the specific part (the operations whose database access needs to be profiled) to a subaspect

Example: Reusable Database Access Profiling

The reusable aspect abstracts from the affected joinpoints: the operations to be profiled

```
abstract aspect DatabaseAccessProfiling percflow(profiledOp()) {
  abstract pointcut profiledOp();
  pointcut databaseOp(): call(* java.sql..*.*(..));
 int accessCount = 0;
  before(): databaseOp() { accessCount++; }
  after(): profiledOp () {
     System.out.println(thisJoinPoint.getSignature()
       + " accessed DB " + accessCount + " times");
  }
}
```

The subaspect specifies where the aspect is to be applied

```
aspect AccountOpProfiling extends DatabaseAccessProfiling {
   pointcut profiledOp(): execution(* Account.*(..));
}
```

7.7 Static Advice and Property Enforcement

7.7 Static Advice and Property Enforcement

- 7.7.1 Declare Error / Warning
- 7.7.2 Enforcing Coding Conventions
- 7.7.3 Enforcing Architectural Conventions
- 7.7.4 Expressing and Imposing Contracts

7.7.1 Declare Error / Warning

- As we have seen, aspects can be used to solve design problems.
- But aspects can also be used to enforce coding and architectural convention at compile time.

- Special kind of advice:
  - `declare error`: generates compile error.
  - `declare warning`: generates compile warning.
- Only works with static pointcuts!

7.7.2 Enforcing Coding Conventions

- Enforces programmers to use setter/getter.
  Pointcut `set(* Line.*) && !withincode(void Line set*(*))`
  matches attribute manipulation outside of setter methods.
- Generates an error at compilation if convention is broken.

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

        static aspect SetterEnforcement {

                declare error: set(* Line.*) &&
                                !withincode(void Line set*(*))
                "Use setter method, even inside Line class.";
        }
}
```

7.7.3 Enforcing Architectural Conventions

- Enforces programmers to use factory methods.
- Generates error at compilation time if convention is broken.

```
aspect FactoryEnforcement {

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

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

        declare error: newShape() && !inFactory():
                "Call factory to create shapes.";

}
```

7.7.4 Expressing and Imposing Contracts

- Aspects cannot only be used to express and enforce contracts and architectural styles on some OO code.

- They can also be used
  - to express and enforce contracts regulating the interface between aspects and the code it applies to
  - to express and enforce contracts on the behavior of aspects.

- For illustration consider the following implementation of Display Updating functionality.
- The implementation uses a “coding style” that separates
  - The definition of the interface between the base and the display updating functionality
  - The encoding of the display updating functionality

An Interface Aspect

```
public aspect ShapeChange {

  public pointcut change():
            call(void Shape+.set*(..))
            || call(void Shape+.moveBy(..))
            || call(Shape+.new(..)));

  public pointcut topLevelChange():
           change() && !cflowbelow(change());

  ...
```

First the “functional” interface defining execution points where the two concerns interact.

An Interface Aspect

```
// public aspect ShapeChange {
 ...
 protected pointcut staticmethodscope():
           withincode (void Shape+.set*(..))
        || withincode(void Shape+.moveBy(..))
        || withincode(Shape+.new(..));

 // REQUIRES: state is changed only by Shape mutators
  declare error:
    (!staticmethodscope() && set(int Shape+.*)):
    "Contract violation: must set Shape"
    + " field inside setter methods only!";

...
```

Than the "non-functional" interface stating the assumptions of the aspect about on the base code to ensure proper functioning of the aspect.

An Interface Aspect

```
// public aspect ShapeChange {
   ...
   // PROMISES: advisers will not change state
   before():
     cflow(adviceexecution()) && change() {
        ErrorHandling.signalFatal(
          "Contract violation:"
        + " advisor of ShapeChange cannot"
        + " change Shape instances");
     }
}
```

Than the "non-functional" interface putting constraints on the aspect behavior to avoid that the aspect breaks assumptions of clients of the base code.

The Operational Aspect

```
public aspect DisplayUpdate {

  after():
    ShapeChange.topLevelChange() {
         updateDisplay();
    }

   public void updateDisplay() {
      Display.update();
   }
}
```

7.8 Takeaway

- Crosscutting functionality involves several classes/objects and is for this reason hard to modularize with OO.
- Goal of AOP extend the programming toolbox with language mechanisms to express crosscutting structure and to attach functionality to it in a modular way.
- As a consequence
  - the potential for better separation of concerns and better support for design principles is increased
  - with respective advantages for evolvability and customizability

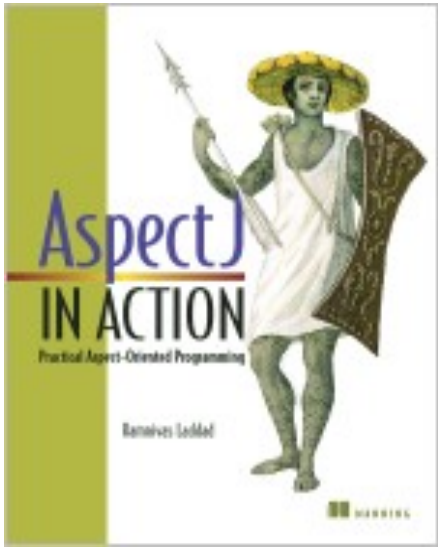
Takeaway: AspectJ

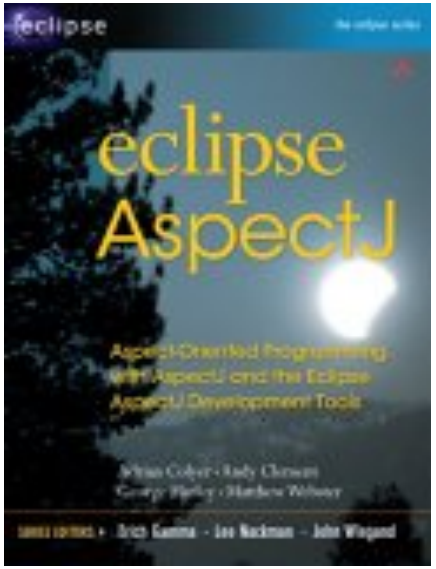
- AspectJ is the most mature and used AOP language.
- AspectJ adds to Java:
  - Pointcuts to
    - identify points in the execution where aspect functionality joins
    - declare execution state at these points to be exposed to aspects
  - Advice to define the functionality to be added at execution points identified by pointcuts.
  - ITDs add new state, methods, and inheritance relations to classes.
  - A new kind of module, called aspect, which in addition to attributes and methods can contain pointcut, advice, and ITDs.

Takeaway: Further Material on AspectJ

- http://www.eclipse.org/aspectj/
- AspectJ in Action, by Ramnivas Laddad
- Eclipse AspectJ by Adrian Colyer, Andy Clement, George Harley

- Aspect implementations of authentication, authorization, transactions, caching … show the power of AspectJ in a more convincing way.





Takeaway: Other AOP Languages

- There are many other AOP languages.
- All support some notion of pointcut and advice but differ on the power and abstraction capabilities of the pointcuts as well as specific characteristics of advice.
- There are significant differences with regard to the aspect modules:
  - Concerning their instantiation and scoping mechanisms.
  - Concerning the way aspect-specific state and behavior is defined.