

***Software
Technology
Group***

TU Darmstadt | FB Informatik

Software Engineering Design

10. Aspect-Oriented Programming

Prof. Dr. Mira Mezini

Dipl.-Inform. Christoph Bockisch

Dipl.-Ing. Michael Haupt

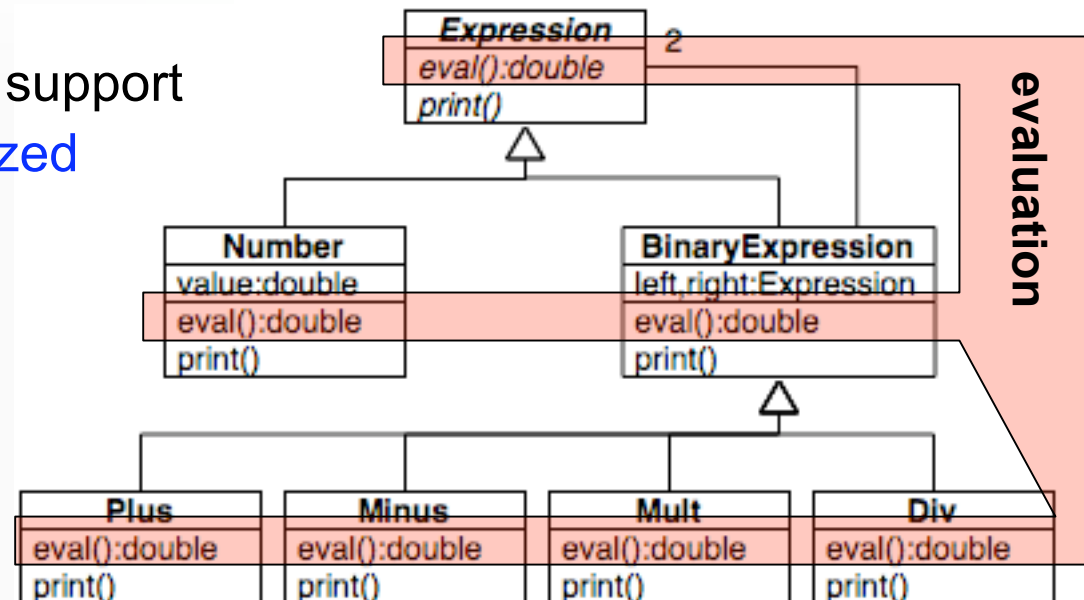
Outline

- Introduction to the AOP idea
- Introduction to AspectJ
- Examples

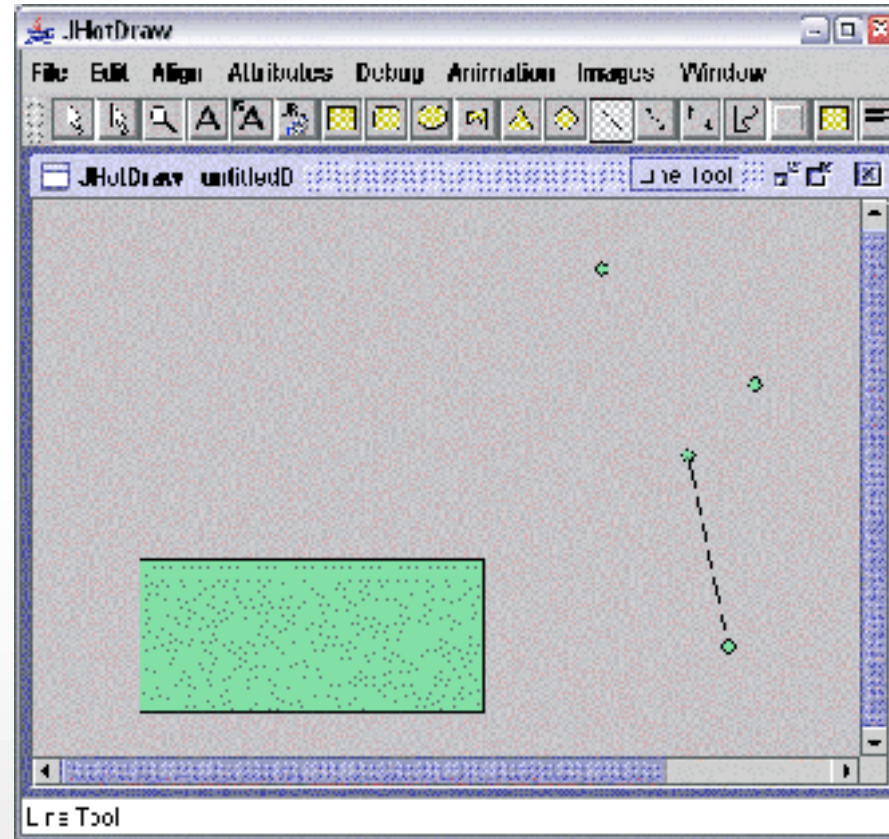
The AOP Idea

aspect-oriented programming

- crosscutting is inherent in complex systems
- crosscutting concerns
 - have a clear purpose
 - have a natural structure
 - defined as a set of methods, module boundary crossings, points of resource utilization, lines of dataflow...
- so, let's capture the structure of crosscutting concerns explicitly...
 - in a modular way
 - with linguistic and tool support
- aspects are **well-modularized** cross-cutting concerns



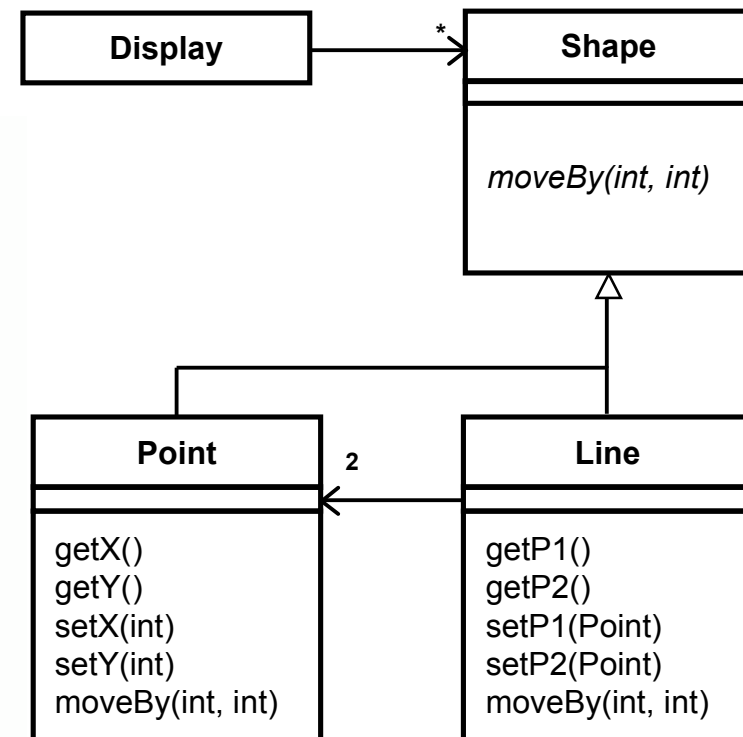
Consider Developing...



a simple drawing application (JHotDraw)

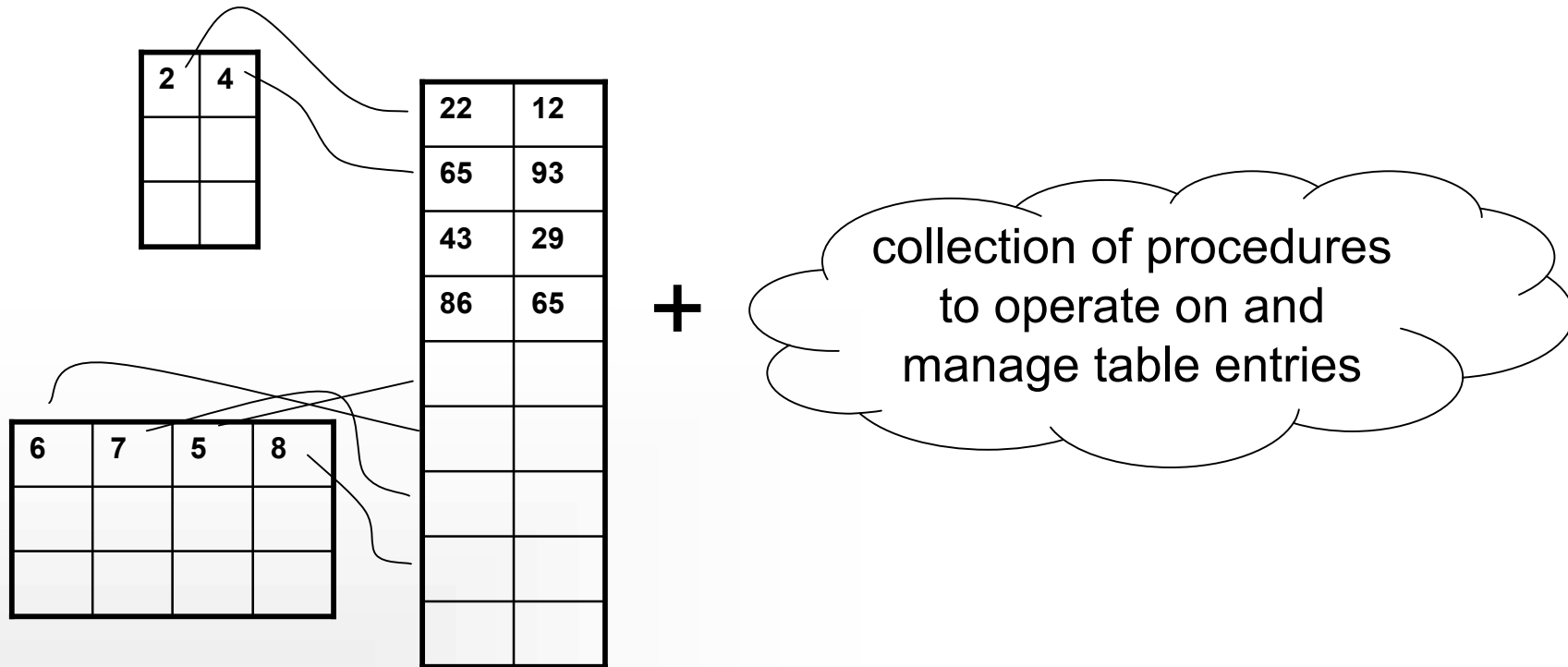
Intuitively Thinking of Objects?

- Points, Lines...
- Drawing surfaces
- GUI Widgets
- ...



In 1969...

most programmers would have used something like this:



this has **poor design and code modularity!**

What is OOP?

- a “way of thinking”
 - objects, classification hierarchies
- supporting mechanisms
 - classes, encapsulation, polymorphism...
- allows us to
 - make code look like the design
 - improves design and code modularity
- many possible implementations
 - style, library, ad-hoc PL extension, integrated in PL

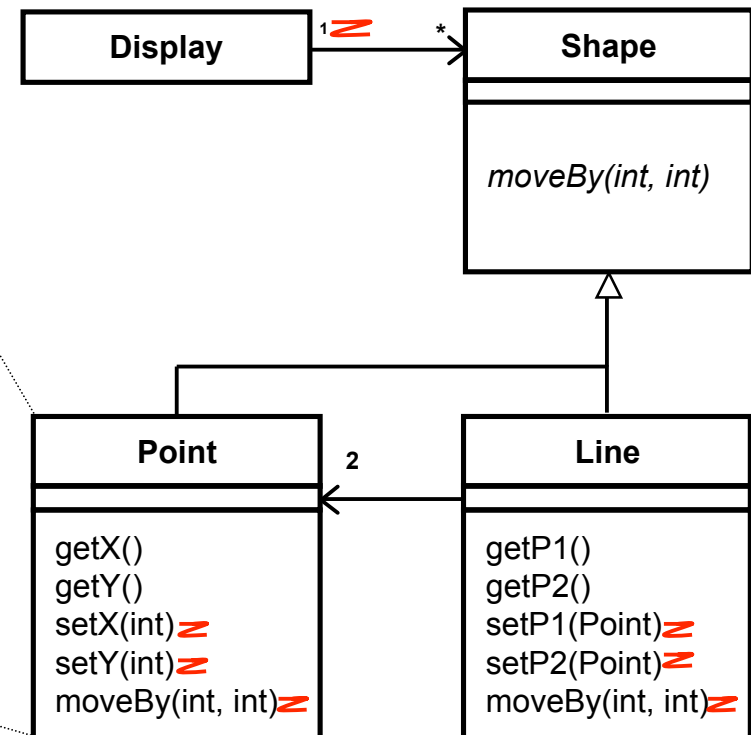
many other
benefits build
on these

But Some Concerns "Don't Fit"

e.g., a simple **Observer** pattern

fair design modularity
but poor code modularity

```
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 AOP, they *Do* Fit

aspect ObserverPattern {

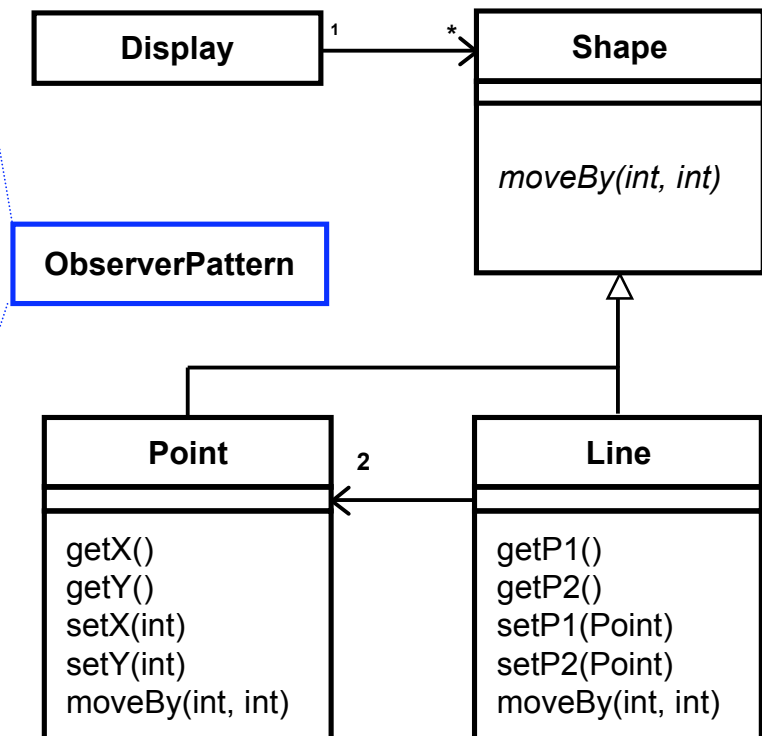
```
private Display Shape.display;
```

```
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();  
}
```

good design modularity
good code modularity



Code Looks Like the Design

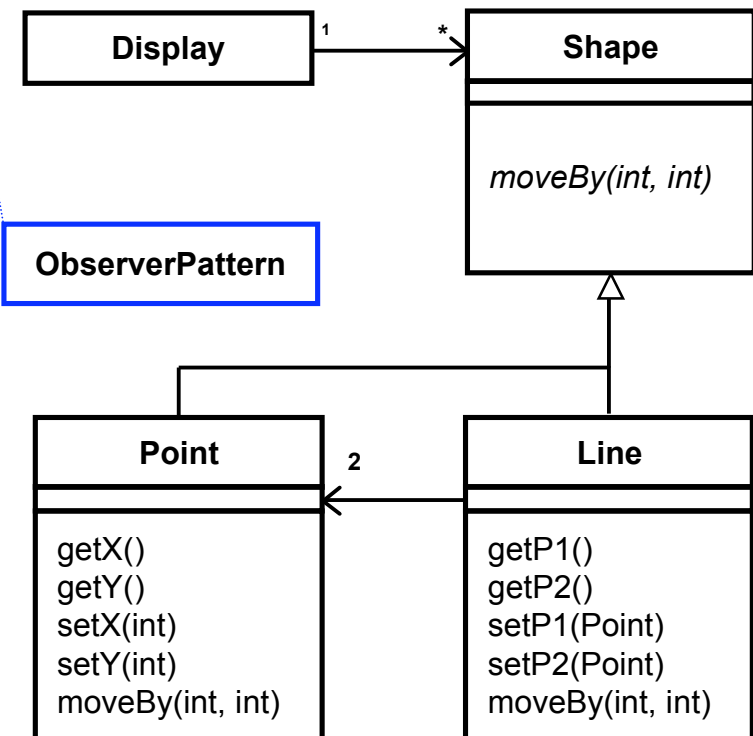
aspect ObserverPattern {

```
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();  
}
```

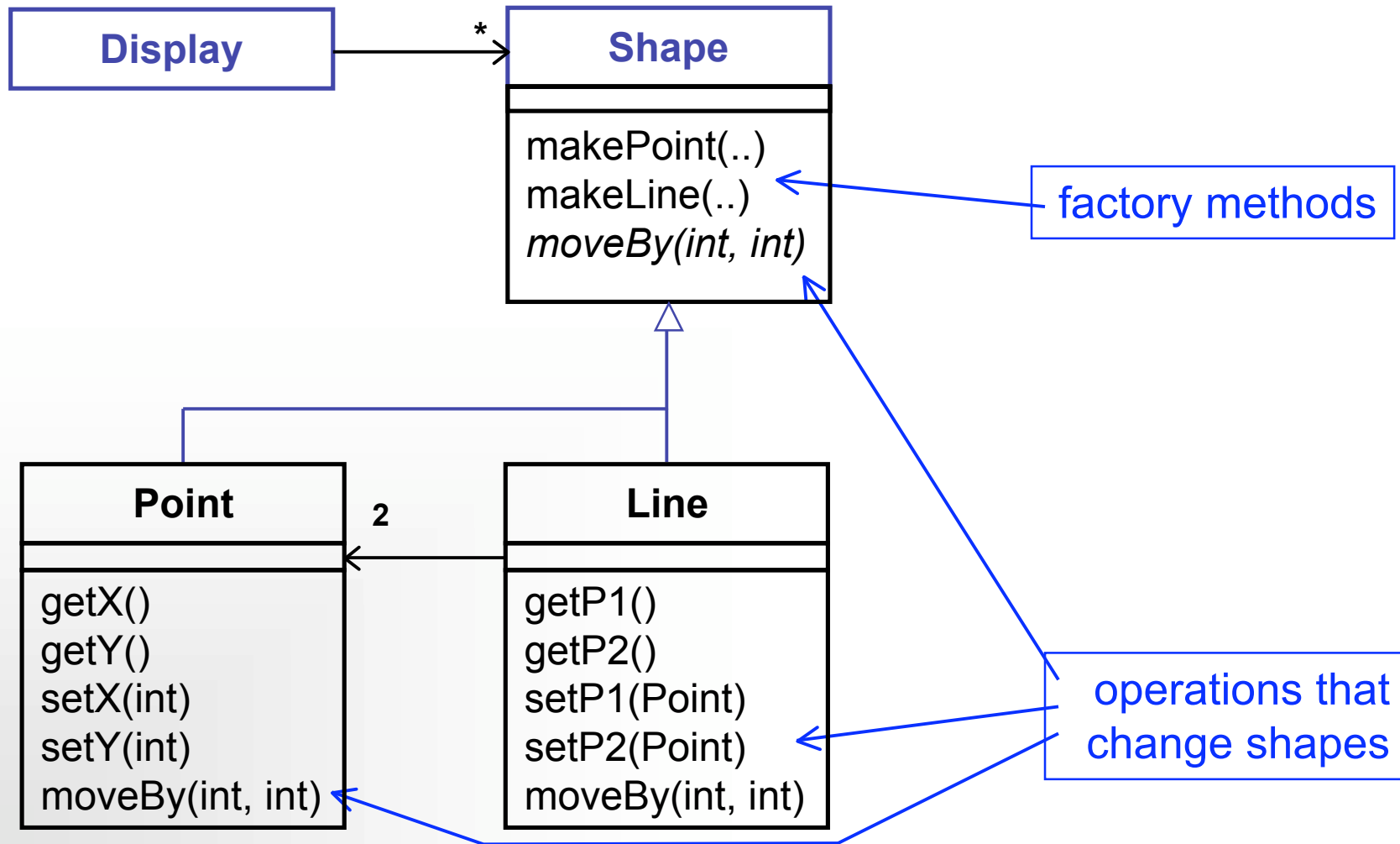
Ask yourself: could you name a single class “ObserverPattern”?



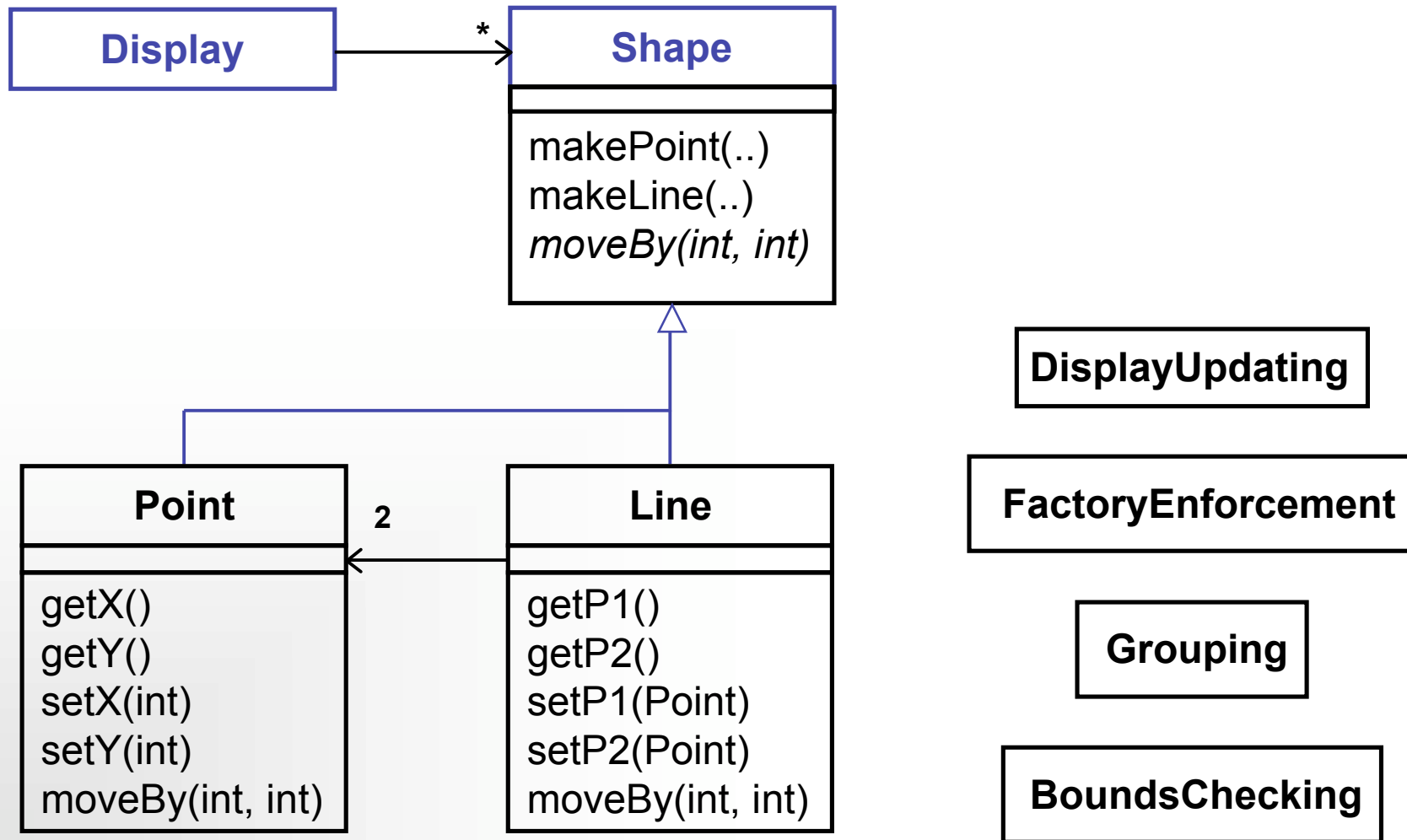
What is AOP?

- “a way of thinking”
 - aspects, crosscutting structure
- supporting mechanisms
 - join points, pointcuts, advice...
- allows us to
 - make code look like the design
 - improve design and code modularity
- many possible implementations
 - style, library, ad-hoc PL extension, integrated in PL

Today, We All *Intuitively* See...



AOP Developers Intuitively See...



Outline

- Introduction to the AOP idea
- Introduction to AspectJ
- Examples

AspectJ

- AspectJ is:
 - an aspect-oriented extension to Java
 - supports general-purpose aspect-oriented programming
- aspects are two things:
 - concerns that crosscut [design level]
 - a programming construct [implementation level]
 - enables crosscutting concerns to be captured in modular units

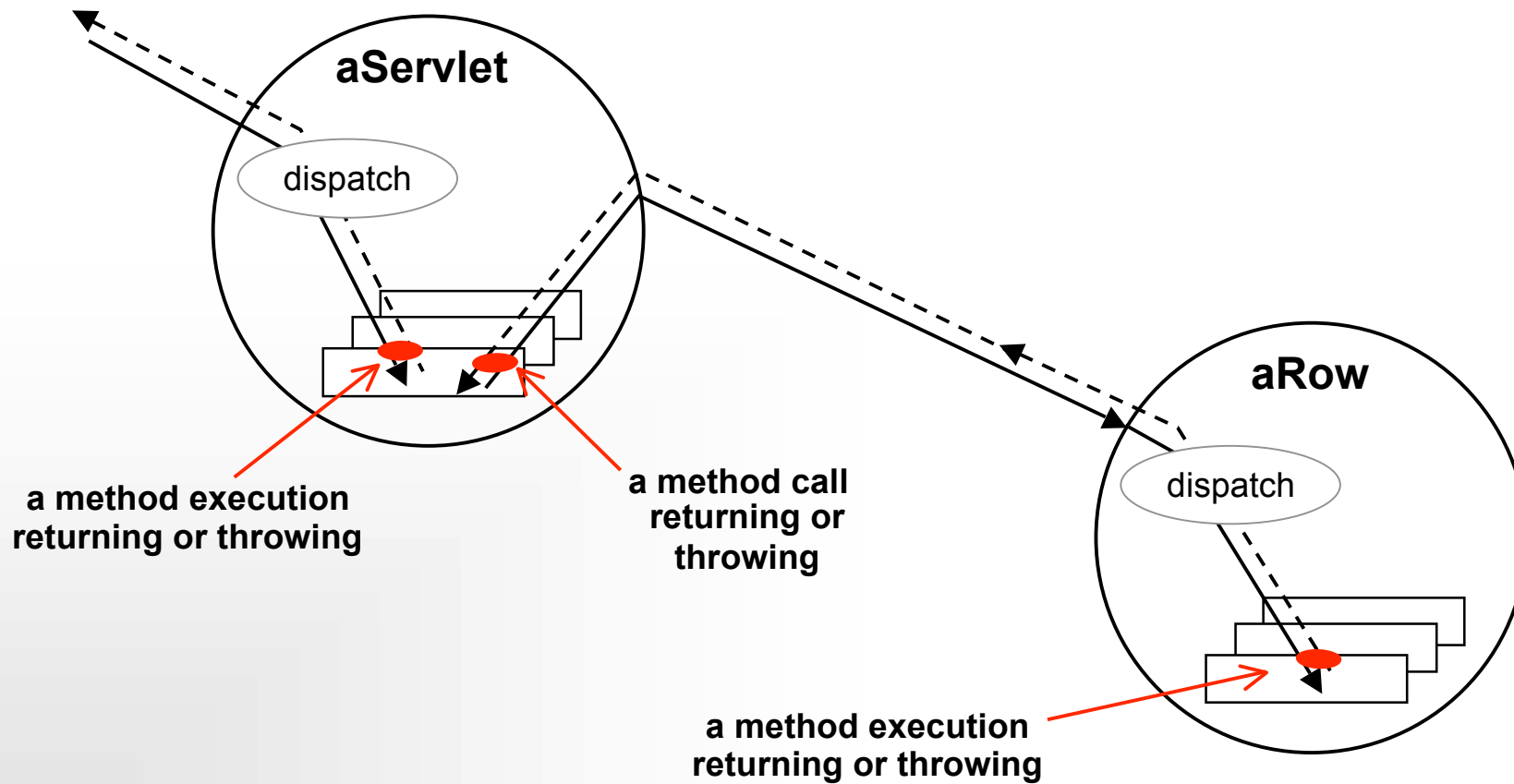
AOP Concepts

- **aspect**
 - unit of modularity for crosscutting concerns
- **join point**
 - well-defined point in the program flow
- **pointcut**
 - join point "query"
 - defines in one place "where to affect existing behaviour"
- **advice**
 - additional code to be executed at specified points in the program execution
 - defines "how to affect the behaviour"

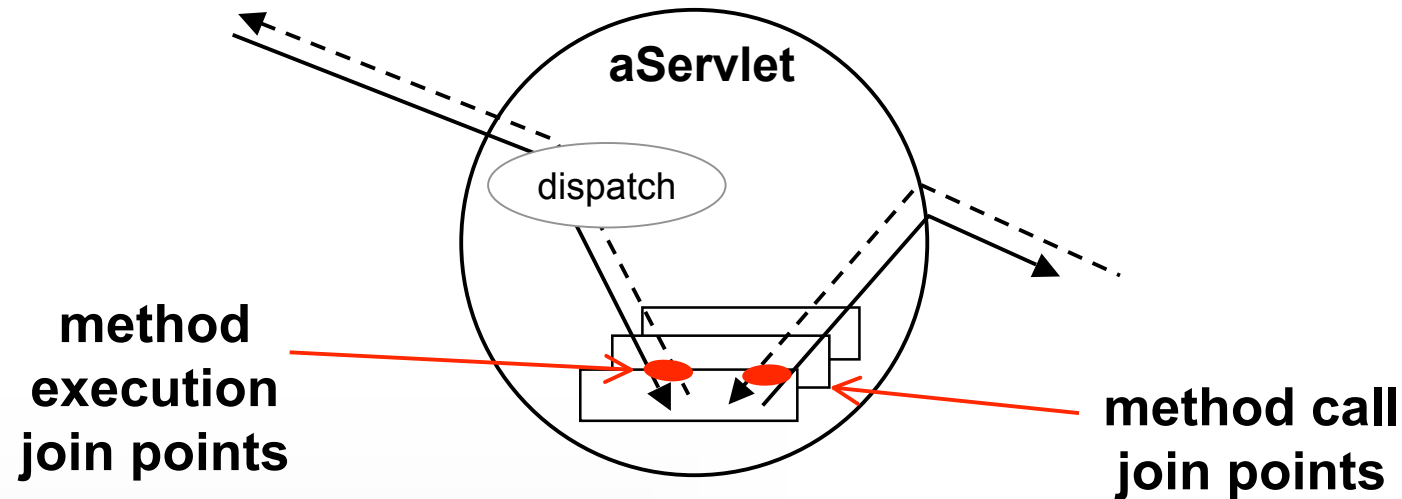
Join Points

key points in dynamic execution flow

Imagine `aServlet.service(request, response)`



Join Point Terminology



- Several kinds of join points
 - method & constructor call
 - method & constructor execution
 - field get & set
 - exception handler execution
 - static & dynamic initialization

Pointcuts

name certain join points

each time there is a call to a method in **banking.Account**
or **banking.InterAccountTransferSystem**

name and parameters

call to any method in banking.Account

```
public pointcut authOperations():  
    call(* banking.Account.*(..))  
|| call(* banking.InterAccountTransferSystem.*(..));
```

or

call to any method in banking.InterAccountTransferSystem

Primitive Pointcuts

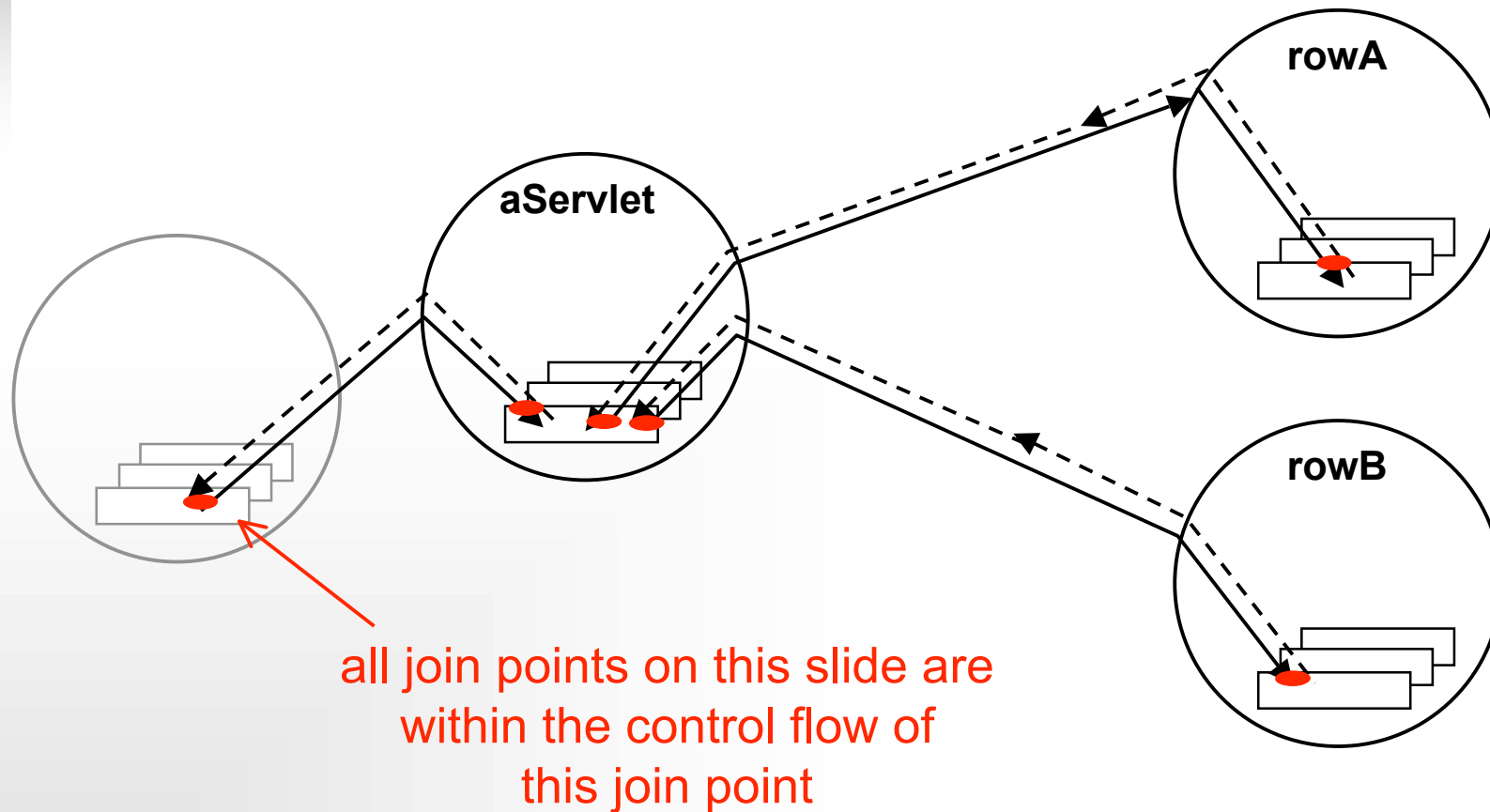
- **Some example pointcuts** picking out
 - when a particular method body executes
 - **execution(void Point.setX(int))**
 - when a method is called
 - **call(void Point.setX(int))**
 - when an exception handler executes
 - **handler(ArrayOutOfBoundsException)**
 - when the object currently executing (i.e., **this**) is of type **SomeType**
 - **this(SomeType)**

Primitive Pointcuts

- **Some example pointcuts** picking out
 - when the target object is of type **SomeType**
 - **target(SomeType)**
 - when the executing code belongs to class **MyClass**
 - **within(MyClass)**
 - when the join point is in the control flow of a call to a **Test**'s no-argument main method
 - **cflow(call(void Test.main()))**

Join Point Terminology

key points in program or in the
dynamic execution flow



Join Point Selection via Wildcards

- **execution(* *(..))**
 - **call(* set(..))**
1. the execution of any method **regardless of return or parameter types**, and
 2. the call to any method named **set()**, **regardless of return or parameter types** -- in case of overloading there may be more than one such **set()** method; this pointcut picks out calls to all of them.

Join Point Selection Based on Types

- `execution(int *())`
 - `call(* setY(long))`
 - `call(* Point.setY(int))`
 - `call(*.new(int, int))`
1. execution of any method with no parameters that returns an **int**,
 2. the call to any **setY()** method that takes a **long** as an argument, regardless of return type or declaring type,
 3. the call to any of **Point**'s **setY()** methods that take an **int** as an argument, regardless of return type, and
 4. the call to any classes' constructor, so long as it takes exactly two **int** as arguments.

Join Point Selection Based on Modifiers

- **call(public * *(..))**
 - **execution(!static * *(..))**
 - **execution(public !static * *(..))**
1. any call to a public method
 2. any execution of a non-static method
 3. any execution of a public, non-static method.

Composing Pointcuts

- **target(Point) && call(int *())**
 - any call to an **int** method with no arguments on an instance of **Point**, regardless of its name
- **call(* *(..)) && (within(Line) || within(Point))**
 - any call to any method where the call is made from the code in **Point**'s or **Line**'s type declaration
- **within(*) && execution(*.new(int))**
 - the execution of any constructor taking exactly one **int** argument, regardless of where the call is made from,
- **!this(Point) && call(int *(..))**
 - any method call to an **int** method when the executing object is any type except **Point**.

Advice

action to take before, after, or around
computation at join points

also **after** & **around**

```
before() { authOperations() {  
    if(!_authenticatedSubject != null) { return; }  
    try {  
        authenticate();  
    }  
    catch(LoginException ex) {  
        throw new AuthenticationException(ex);  
    }  
}
```

Passing Context

...from join points to advice

```
before ( Account account, float amount ):  
  call (void Account.credit(float))  
  && target ( account )  
  && args ( amount )  
{  
  System.out.println("Crediting "  
    + amount + " to " + account );  
}
```

passing **target** object

Passing Context

...from join points to advice

```
before ( Account account, float amount ) :  
  call (void Account.credit(float))  
  && target ( account )  
  && args ( amount )  
{  
  System.out.println("Crediting "  
    + amount + " to " + account );  
}
```

passing **argument** object

Passing Context

...from join points to advice

```
after() returning(Connection conn):  
  call(Connection DriverManager.getConnection(..)) {  
    System.out.println("Obtained database connection " +  
      conn)  
  };  
}
```

passing **return** object

```
after() throwing(RemoteException ex):  
  call(* *.*(..) throws RemoteException) {  
    System.out.println("Exception " + ex +  
      "while executing " +  
      thisJoinPoint)  
  };  
}
```

accessing **special variable**

passing **exception** object

Reflective Access to Join Points

- by means of three special objects available in each advice body:
 - **thisJoinpoint**,
 - **thisJoinpointStaticPart**,
 - **thisEnclosingJoinpointStaticPart**
- dynamic information → changes with each invocation of the same join points
 - receiver, parameters, etc.
- static information → does not change between multiple executions
 - Kind of join point: method call, setter, getter, etc.
 - name and source code location of a method join point

Outline

- Introduction to the AOP idea
- Introduction to AspectJ
- Examples

Modularizing Authentication in AspectJ

```
public interface Account {  
    int getAccountNumber();  
    void credit(float amount);  
    void debit(float amount)  
        throws InsufficientBalanceException;  
    float getBalance();  
}
```

Modularizing Authentication in AspectJ

```
public class AccountSimpleImpl implements Account {
    private int _accountNumber;
    private float _balance;
    public AccountSimpleImpl(int accountNo)
        { _accountNumber = accountNo; }
    public int getAccountNumber()
        { return _accountNumber;}
    public void credit(float amount)
        { _balance += amount;}
    public void debit(float amount)
        throws InsufficientBalanceException {
        if(_balance < amount) {
            throw new InsufficientBalanceException(
                "Total balance not sufficient");
        } else { _balance -= amount; }
        }
    public float getBalance() { return _balance; }
}
```

Modularizing Authentication in AspectJ

```
public class InterAccountTransferSystem {  
    public static void transfer(  
        Account from, Account to, float amount  
    ) throws InsufficientBalanceException  
    {  
        to.credit(amount);  
        from.debit(amount);  
    }  
}
```

Modularizing Authentication in AspectJ

```
public class Test {  
    public static void main(String[] args) throws Exception  
    {  
        Account account1 = new AccountSimpleImpl(1);  
        Account account2 = new AccountSimpleImpl(2);  
        account1.credit(150);  
        account1.credit(200);  
  
        InterAccountTransferSystem.transfer(account1,  
                                             account2, 100);  
        InterAccountTransferSystem.transfer(account1,  
                                             account2, 100);  
    }  
}
```

Modularizing Authentication in AspectJ

- Next: a **reusable authentication service** modularized in an aspect...
 - leaves abstract the definition of the crosscut → abstract pointcut definition
 - the set of join points to be affected by the authorization logic is application specific
 - to be defined specifically for each application of the aspect
- Defines the semantic effect of the authentication in an advice associated to the abstract pointcut

Modularizing Authentication in AspectJ

```
public abstract aspect AbstractAuthAspect {  
    private Subject _authenticatedSubject;  
    public abstract pointcut authOperations();  
  
    before(): authOperations() {  
        if (_authenticatedSubject != null) { return; }  
        try { authenticate(); }  
        catch (LoginException ex) {  
            throw new AuthenticationException(ex); }  
    }  
  
    private void authenticate() throws LoginException {  
        ... do authentication ... }  
  
    public static class AuthenticationException  
        extends RuntimeException  
    { ... }  
}
```

Modularizing Authentication in AspectJ

```
import auth.AbstractAuthAspect;  
  
public aspect BankingAuthAspect  
extends AbstractAuthAspect {  
  
    public pointcut authOperations():  
        call(* banking.Account.*(..))  
        || call(* banking.InterAccountTransferSystem.*(..));  
  
}
```

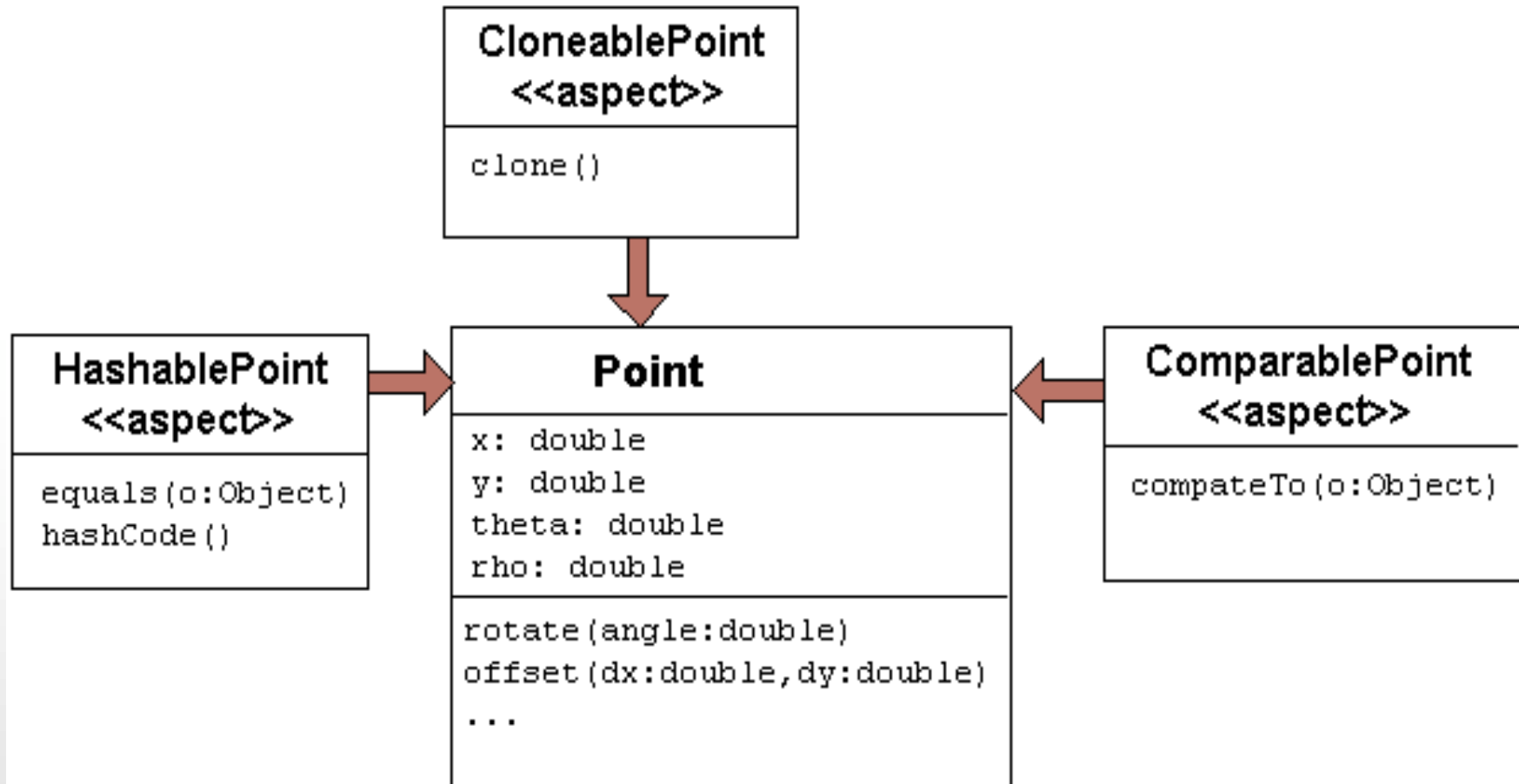
Modularizing Authentication in AspectJ

- A sample implementation of the [authorization service](#) shows the power of AspectJ in an even more convincing way...
 - See: [AspectJ in Action](#), by Ramnivas Laddad, Chap. 10
- Similarly, one could modularize other concerns such as transaction management, resource management, etc.

Two Kinds of Crosscutting

- aspect
 - a modular unit of crosscutting implementation
 - advice
 - adds functionality to existing operations
 - adds per-aspect state
 - inter-class declarations (ICDs)
 - add new operations
 - add per-object state
 - change inheritance relations
 - act at compile time

ITDs Illustrated: Adding Facets to **Point**



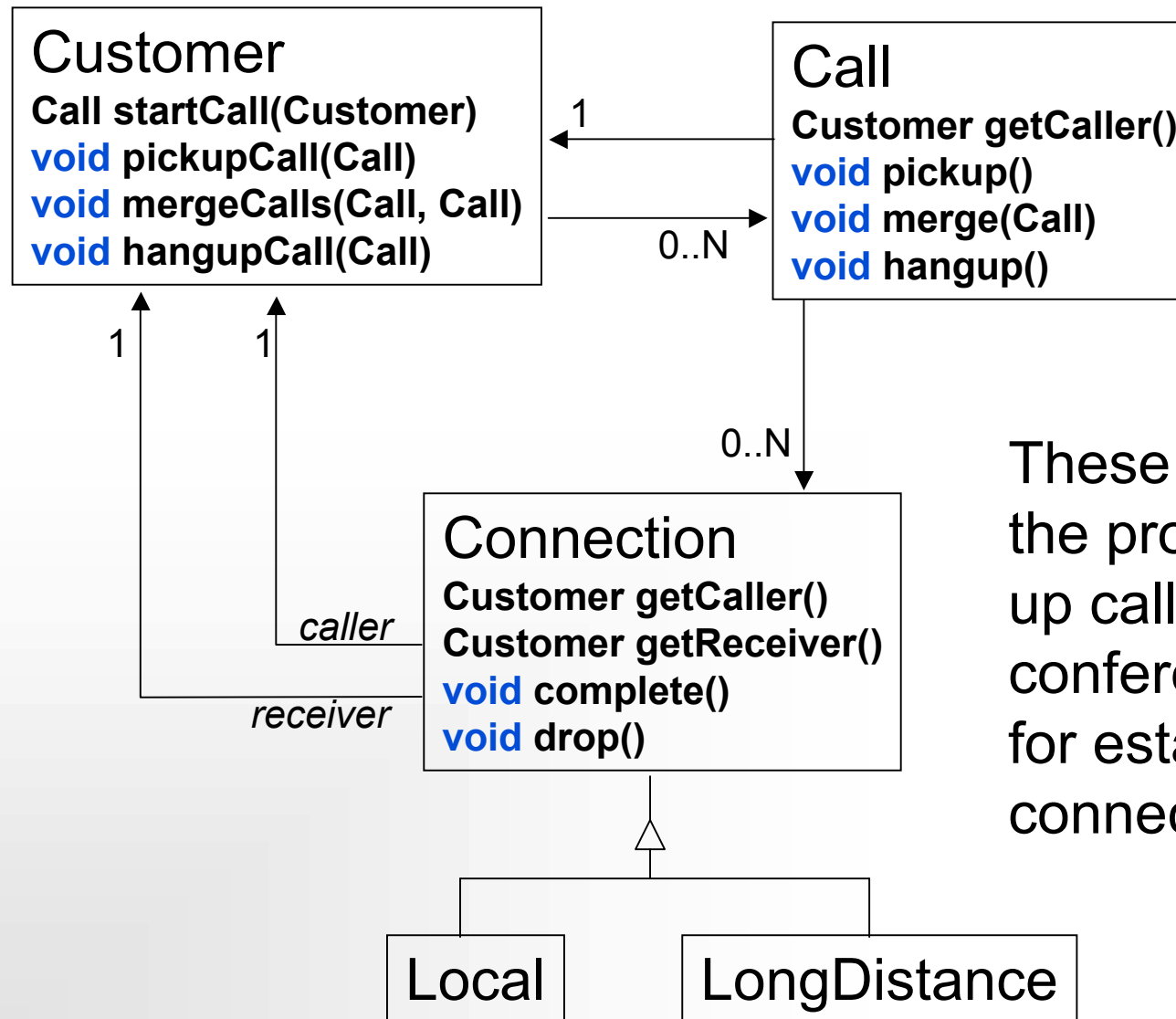
Adding Facets to **Point**

```
public aspect CloneablePoint {  
    declare parents: Point implements Cloneable;  
    public Object Point.clone()  
    throws CloneNotSupportedException {  
        // bring fields up to date before cloning.  
        makeRectangular(); makePolar();  
        return super.clone();  
    }  
    public static void main(String[] args){  
        Point p1 = new Point(); Point p2 = null;  
        p1.setPolar(Math.PI, 1.0);  
        try { p2 = (Point)p1.clone(); }  
        catch (CloneNotSupportedException e) {}  
        System.out.println("p1 =" + p1 );  
        System.out.println("p2 =" + p2 );  
        ...  
    }  
}
```

ITDs: A More Sophisticated Example

- given a basic telecom operation, with customers, calls, connections
- model/design/implement utilities such as
 - timing
 - billing (depends on timing)
 - consistency checks
 - ...

Telecom Basic Design



These classes define the protocols for setting up calls (incl. conference calls) and for establishing connections

AbstractSimulation.java (1)

```
public abstract class AbstractSimulation {
    public static AbstractSimulation simulation;

    public void run() {
        Customer jim = new Customer("Jim", 650);
        Customer mik = new Customer("Mik", 650);
        Customer crista = new Customer("Crista", 415);

        say("jim calls mik..."); Call c1 = jim.call(mik);
        wait(1.0);
        say("mik accepts..."); mik.pickup(c1);
        wait(2.0);
        say("jim hangs up..."); jim.hangup(c1);
        report(jim); report(mik); report(crista);

        say("mik calls crista..."); Call c2 = mik.call(crista);
        say("crista accepts..."); crista.pickup(c2);
        wait(1.5);
        say("crista hangs up..."); crista.hangup(c2);
        report(jim); report(mik); report(crista);
    }
}
```

AbstractSimulation.java (2)

```
// Print a report of connection time for customer  
abstract protected void report(Customer c);
```

```
protected static void wait(double seconds) {  
    /**  
    * Wait 0.1 seconds per "second" for simulation  
    */  
}
```

```
protected static void say(String s){  
    /**  
    * Put a message on standard output  
    */  
}  
}
```

BasicSimulation.java

```
package telecom;
```

```
/**
```

```
 * This simulation subclass implements  
 * AbstractSimulation.run(..) with a test script  
 * for the telecom system with only the basic objects.
```

```
*/
```

```
public class BasicSimulation extends AbstractSimulation {
```

```
    public static void main(String[] args){  
        simulation = new BasicSimulation();  
        simulation.run();  
    }
```

```
        protected void report(Customer c) { }
```

```
    }
```

Running the Basic Version

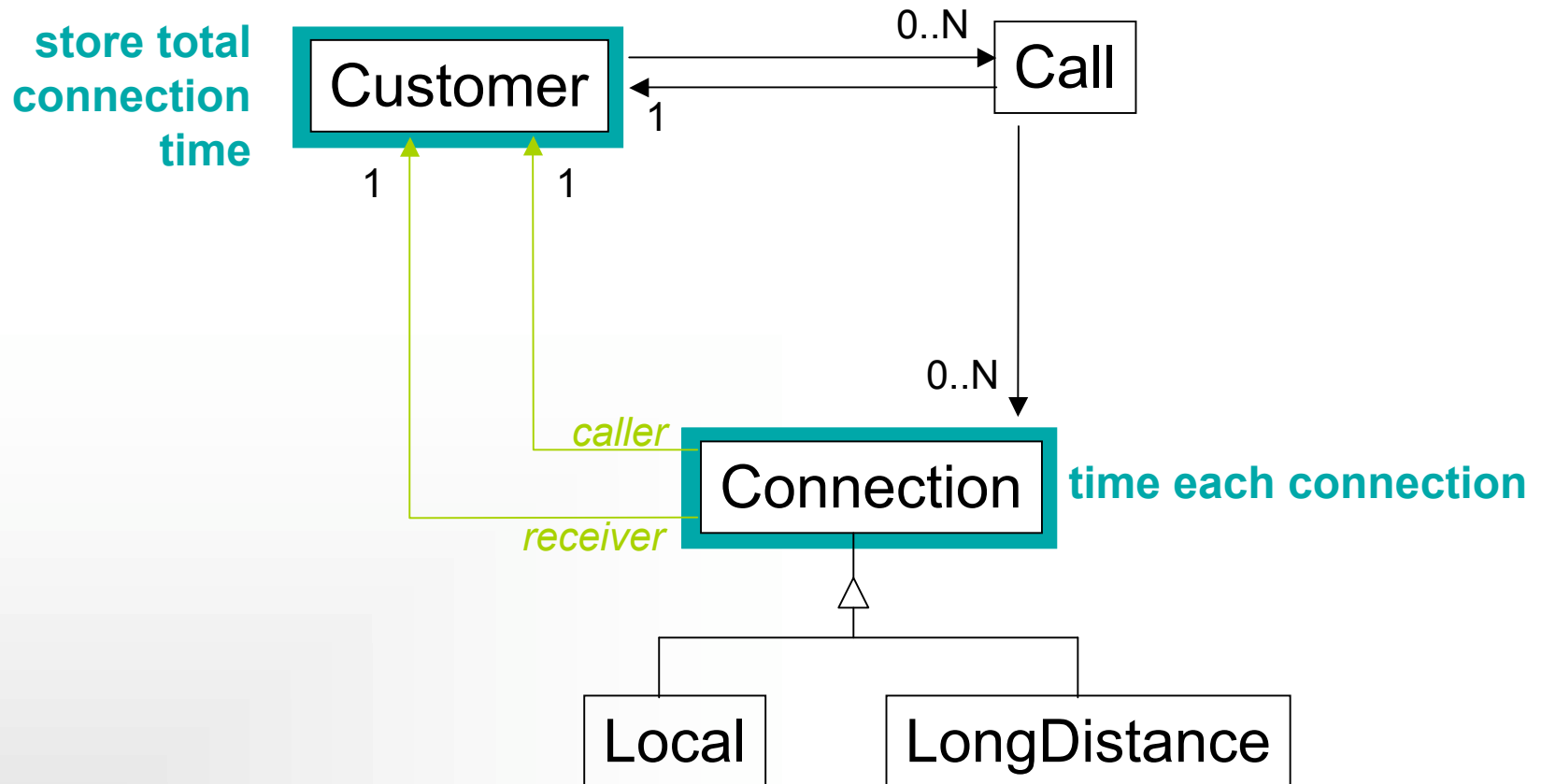
basic.lst

```
AbstractSimulation.java  
BasicSimulation.java  
Call.java  
Connection.java  
Local.java  
LongDistance.java  
Customer.java
```

- **ajc -argfile telecom/basic.lst**
- **java telecom.BasicSimulation**

Timing

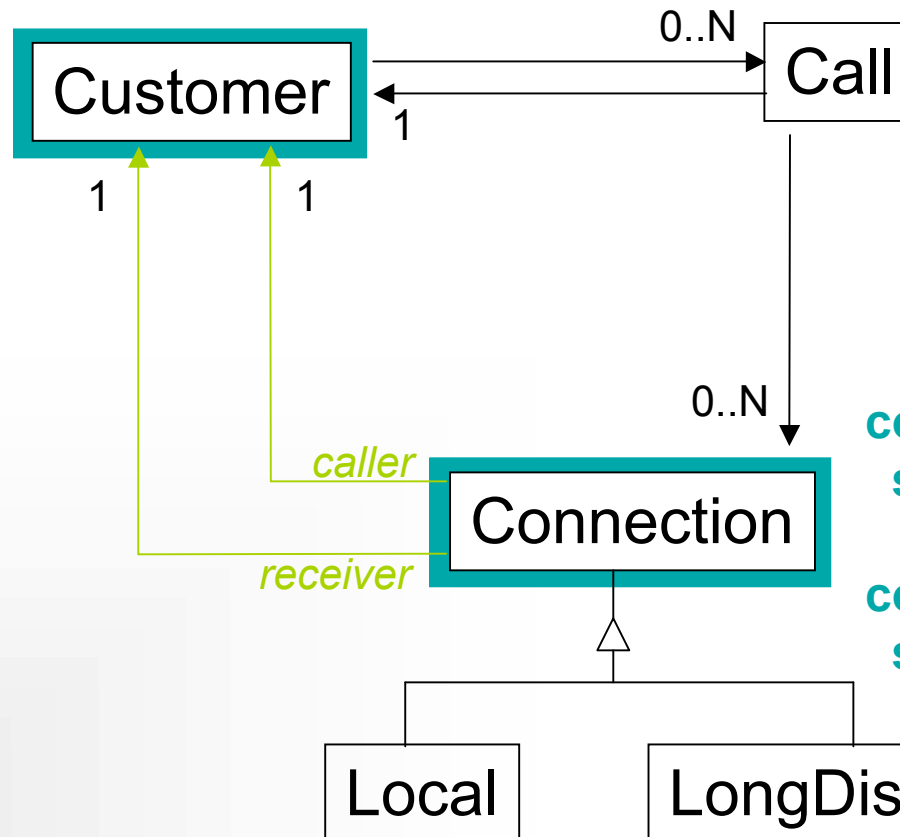
entities



Timing

some actions

connection dropped:
add time

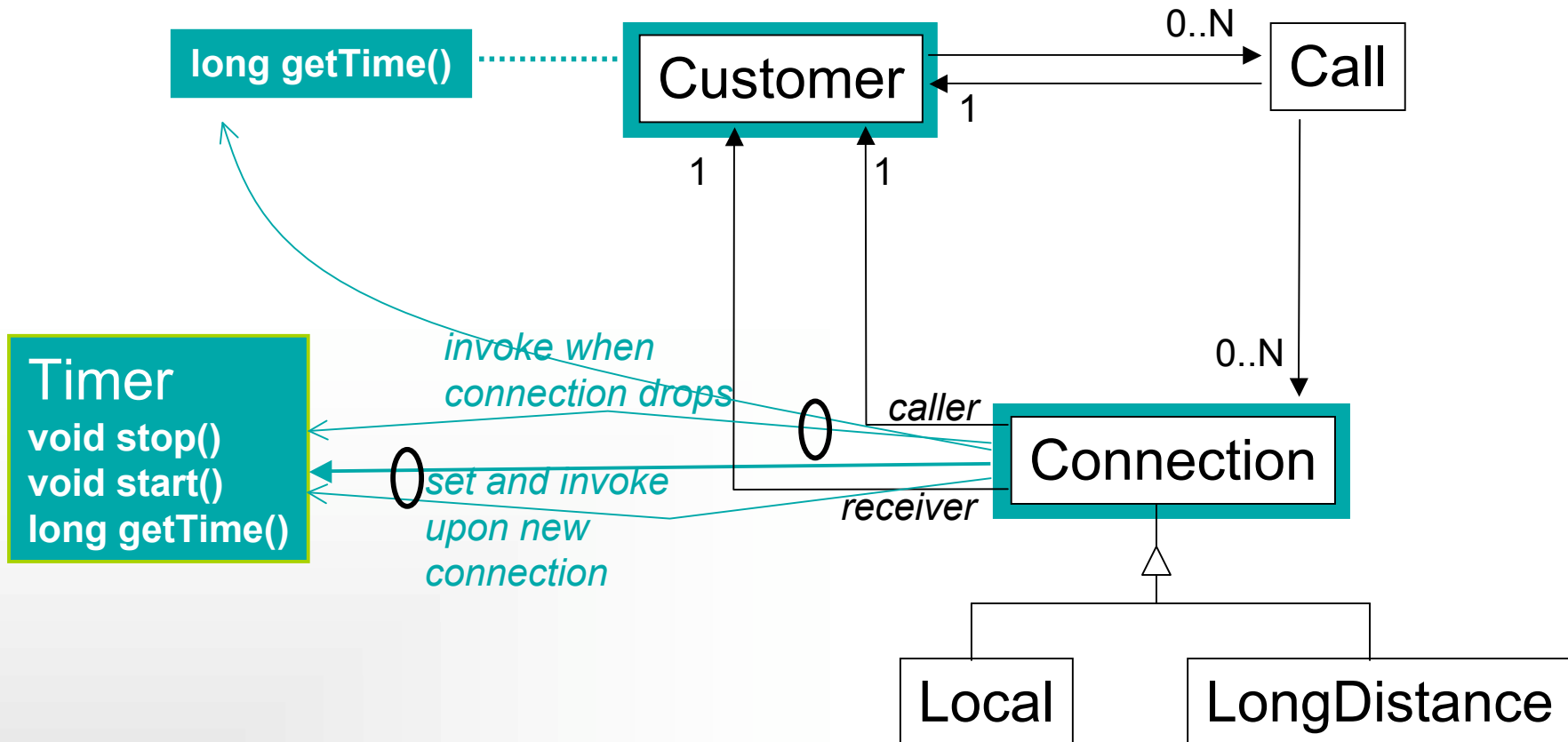


connection made:
start timing

connection dropped:
stop timing

Timing

additional design elements



Timing

what is the nature of the crosscutting?

- connections and calls are involved
- well defined protocols among them
- **pieces of the timing protocol** must be triggered by the execution of certain basic operations. e.g.
 - when connection completed, set and start a timer
 - when connection drops, stop the timer and add time to customers' connection time

Timing: An Aspect Implementation

```
aspect Timing {
```

```
    private Timer Connection.timer = new Timer();  
    private long Customer.totalConnectTime = 0;
```

```
    public static long getTotalConnectTime(Customer c) {  
        return c.totalConnectTime;}  
    public Timer getTimer(Connection conn) { return conn.timer; }
```

```
    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) { getTimer(c).start(); }
```

```
    after(Connection c): endTiming(c) {  
        Timer timer = getTimer(c);  
        timer.stop(); long currTime = timer.getTime();  
        c.getCaller().totalConnectTime += currTime;  
        c.getReceiver().totalConnectTime += currTime;
```

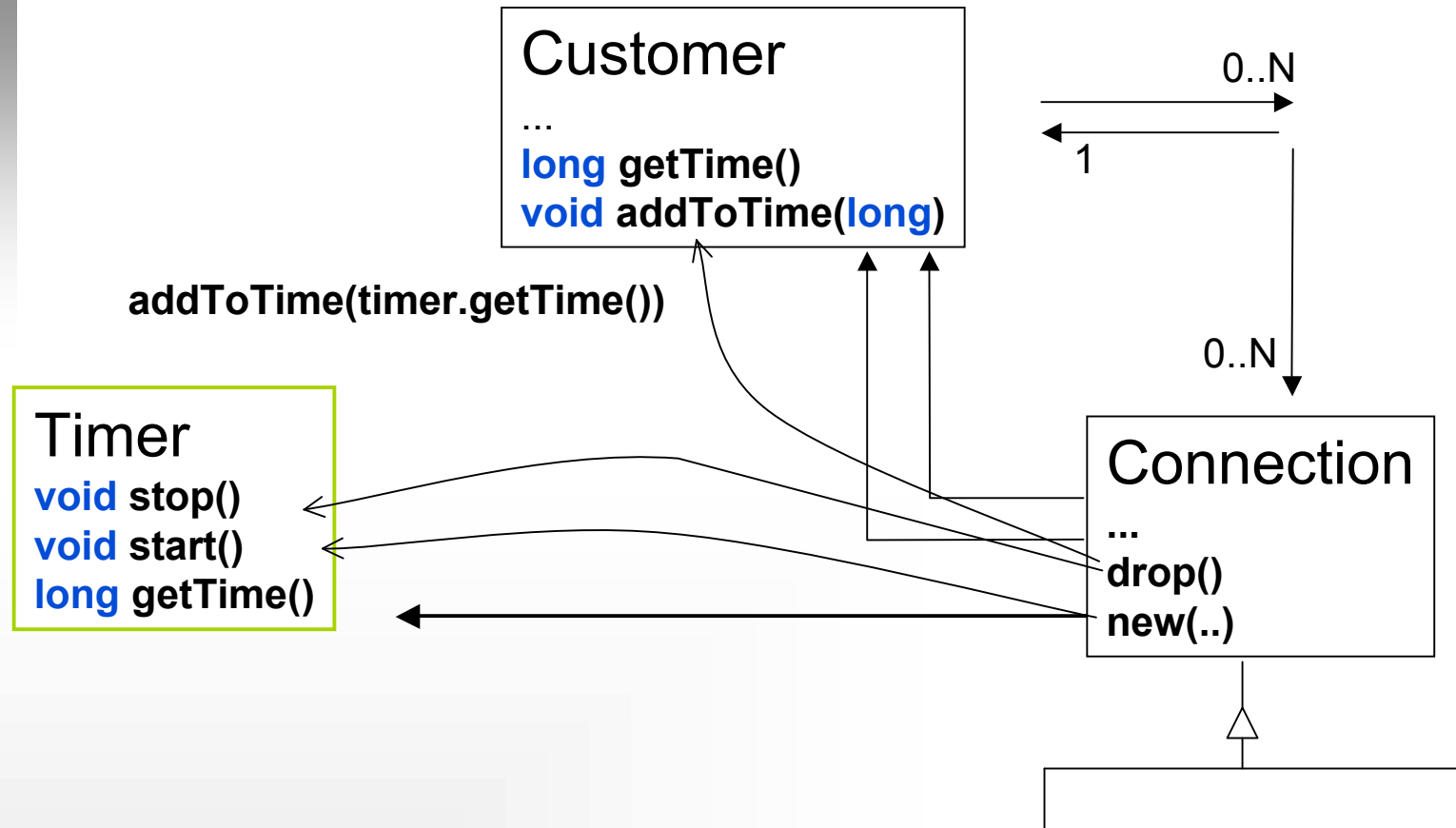
Running the Timing Version

timing.lst

```
AbstractSimulation.java  
TimingSimulation.java  
Call.java  
Connection.java  
Local.java  
LongDistance.java  
Customer.java  
Timer.java  
TimerLog.java  
Timing.java
```

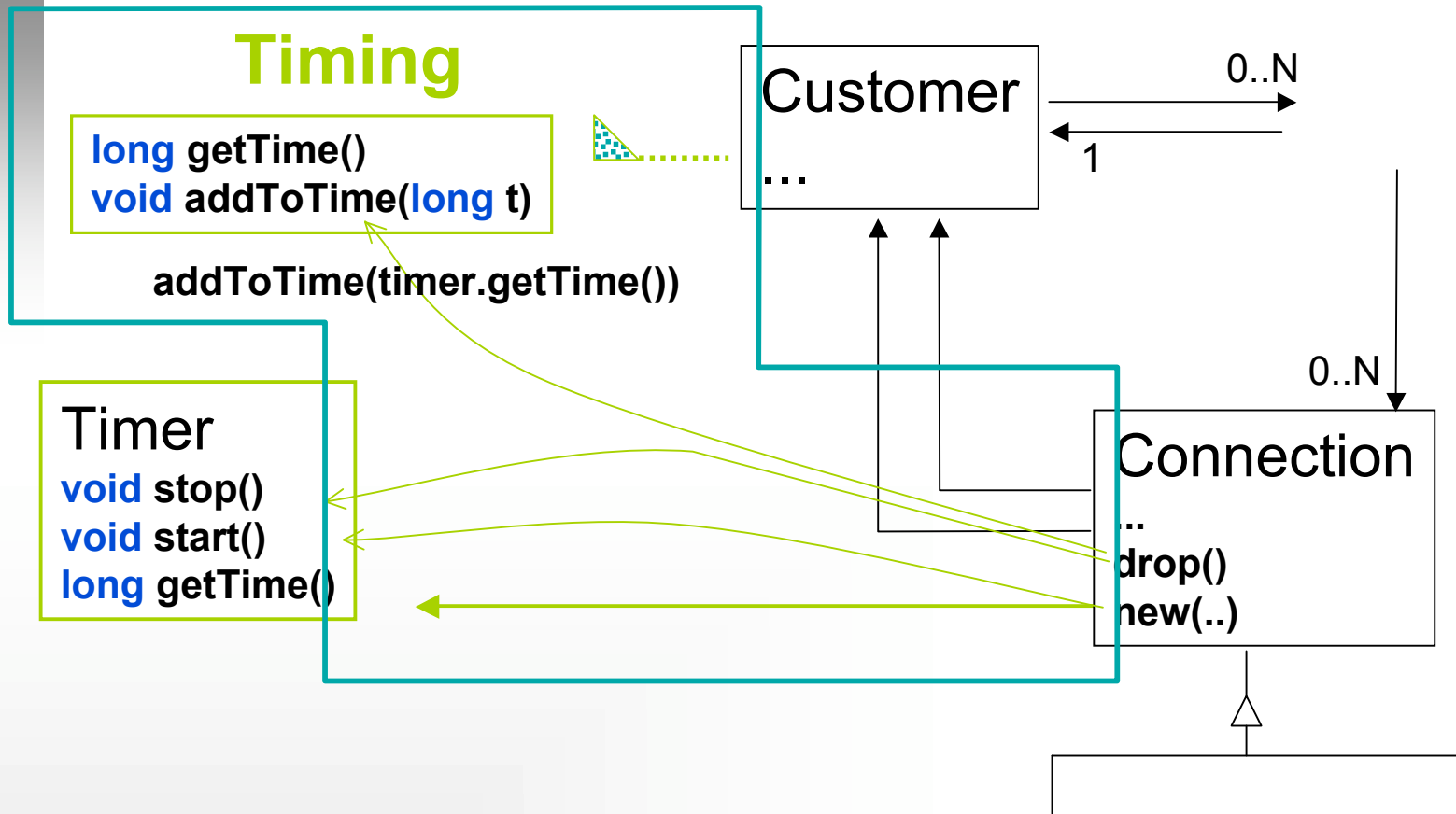
- **ajc -argfile telecom/timing.lst**
- **java telecom.TimingSimulation**

Timing as an Object



timing as an object captures timing support, but **does not capture the protocols involved in implementing the timing feature**

Timing as an Aspect



timing as an aspect captures the protocols involved in implementing the timing feature

Timing

interface change

- Consider a change to the timer interface

```
Timer  
void start()  
long stopAndGetTime()
```

- What changes are necessary in the program?

```
aspect Timing {  
  ...  
  after(Connection c): endTiming(c) {  
    Timer timer = getTimer(c);  
    long currTime = timer.stopAndGetTime();  
    c.getCaller().totalConnectTime += currTime;  
    c.getReceiver().totalConnectTime += currTime;  
  }  
}
```

Timing as an Aspect

has these benefits

- 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 objects
 - only the timing aspect is affected
- adding/removing timing from the design is trivial
 - just add/remove the timing aspect from configuration files

Billing Aspect

```
public aspect Billing {
    // precedence required to get advice on endtiming in the right order
    declare precedence: Billing, Timing;

    public static final long LOCAL_RATE = 3;
    public static final long LONG_DISTANCE_RATE = 10;
    public Customer Connection.payer; public long Customer.totalCharge = 0;
    public Customer getPayer(Connection conn) { return conn.payer; }

    after(Customer cust) returning (Connection conn):
        args(cust, ..) && call(Connection+.new(..)) { conn.payer = cust; }

    public abstract long Connection.callRate();
    public long LongDistance.callRate() { return LONG_DISTANCE_RATE; }
    public long Local.callRate() { return LOCAL_RATE; }

    after(Connection conn): Timing.endTiming(conn) {
        long time = Timing.aspectOf().getTimer(conn).getTime();
        long rate = conn.callRate(); long cost = rate * time;
        getPayer(conn).addCharge(cost);
    }
    public long getTotalCharge(Customer cust) { return cust.totalCharge; }
    public void Customer.addCharge(long charge){ totalCharge += charge; }
}
```

Running the Billing Version

billing.lst

```
AbstractSimulation.java  
BillingSimulation.java  
Call.java  
Connection.java  
Local.java  
LongDistance.java  
Customer.java  
Timer.java  
Billing.java  
Timing.java
```

- **ajc -argfile telecom/billing.lst**
- **java telecom.BillingSimulation**