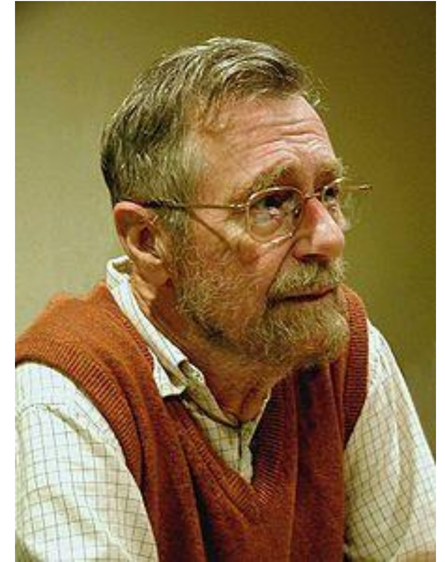
Part I - Introduction to Aspect-Oriented Software Development

Alessandro Garcia

© Alessandro Garcia 2016

Separation of Concerns



- A *concern* is a *"specific requirement or consideration of one or more stakeholders that must be addressed in order to satisfy the overall system goal"*
- Some developer-specific concerns (e.g. instrumentation) need to be modularized as well
- The principle was coined by **Edsger Dijkstra**
  - 1974 paper "On the Role of Scientific Thought"
  - he received the 1972 *A. M. Turing Award* for fundamental contributions in the area of programming languages

Example of “**concerns**” (1) Banking systems

- A banking system, for instance, is a realization of the following concerns:
  - business logic: customer and account management
    - account, loan, customer, etc…
    - taxes
  - Tracing
  - Error handling
  - Interbanking and ATM transaction management
  - Persistence of all entities
  - Authorization of access to various services
  - etc.

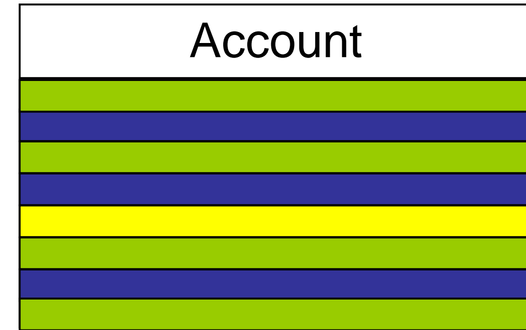
core concerns

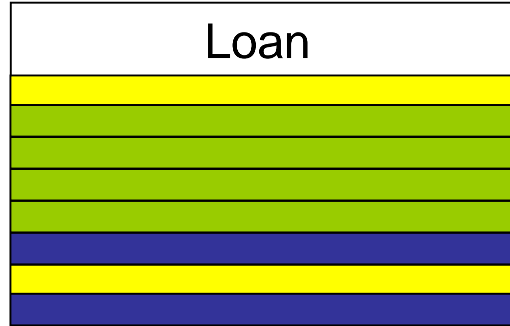
widely-scoped, peripheral concerns

The Problem…

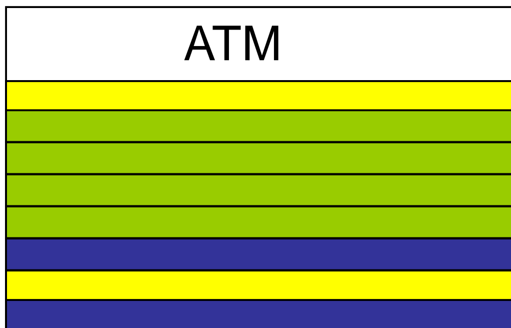
Primary Functionality | Persistence | Error Handling

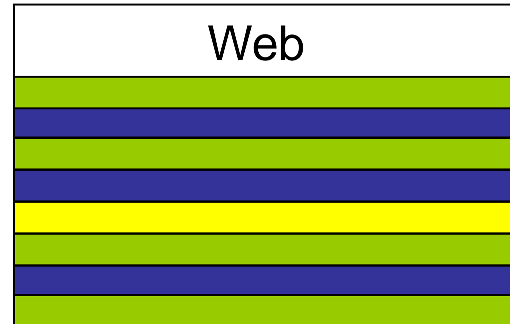
Data Classes

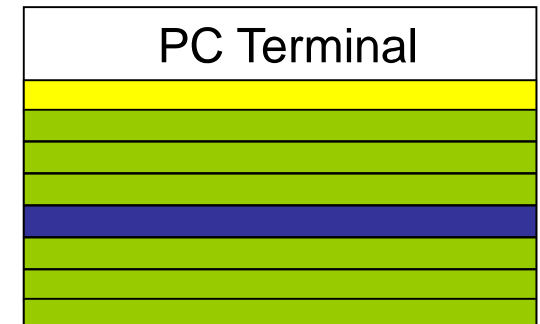




User Interface







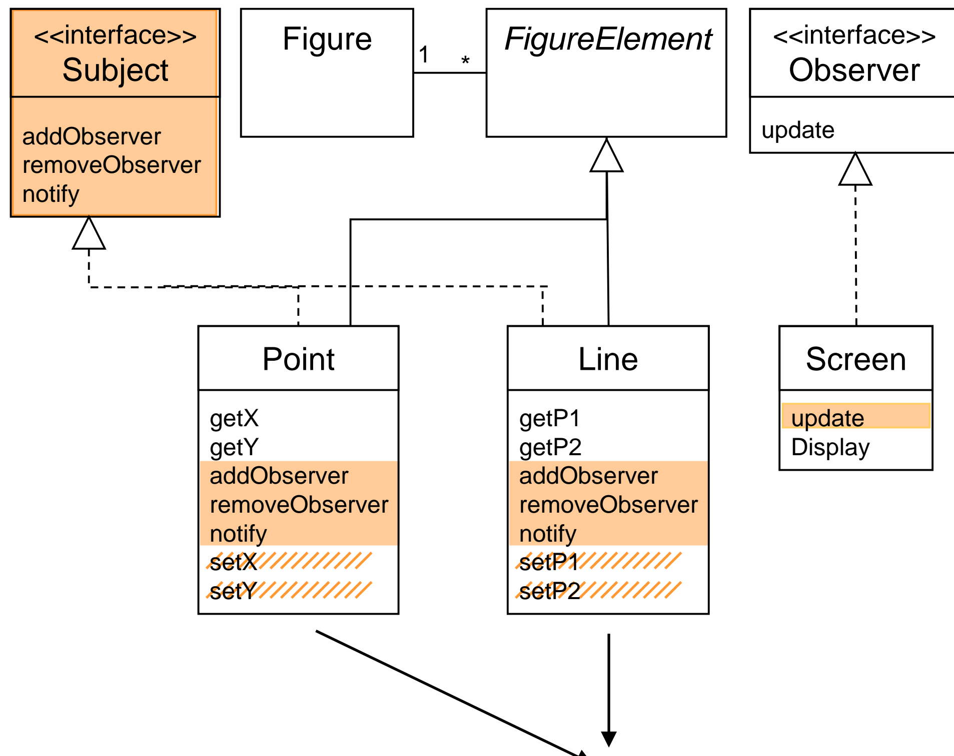
Crosscutting: The Tracing Concern

```
class A {
  // some attributes
  void m1( ) {
        System.out.println("Entering A.m1( )");
        // method code
        System.out.println("Leaving A.m1( )");
  }

  String m2( ) {
        System.out.println("Entering A.m2( )");
        // method code
        System.out.println("Leaving A.m2( )");
        // return a string
  }
```

```
class B {
  // some attributes
  void m2( ) {
        System.out.println("Entering B.m2( )");
  ...
  int m3( ) {
        System.out.println("Entering B.m3( )");
        // method code
        System.out.println("Leaving B.m3( )");
        // return an integer
  }
```

- Tracing is not an abstraction
- No explicit interface

The Observer Pattern: A Design-Specific Concern




```
public class Point
      implements Subject {

    private HashSet observers;

    private int x;
    private int y;

    public Point(int x, int y, Color color) {
        this.x=x;
        this.y=y;
        this.observers = new HashSet();
    }

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

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

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

    public void addObserver(Observer o) {
        this.observers.add(o);
    }

    public void removeObserver(Observer o) {
        this.observers.remove(o);
    }

    public void notifyObservers() {
        for (Iterator e = observers.iterator() ; e.hasNext() ;) {
            ((Observer)e.next()).update(this);
        }
    }
```

interfaces of the classes also become inherently more complicated

What is the problem? Symptoms

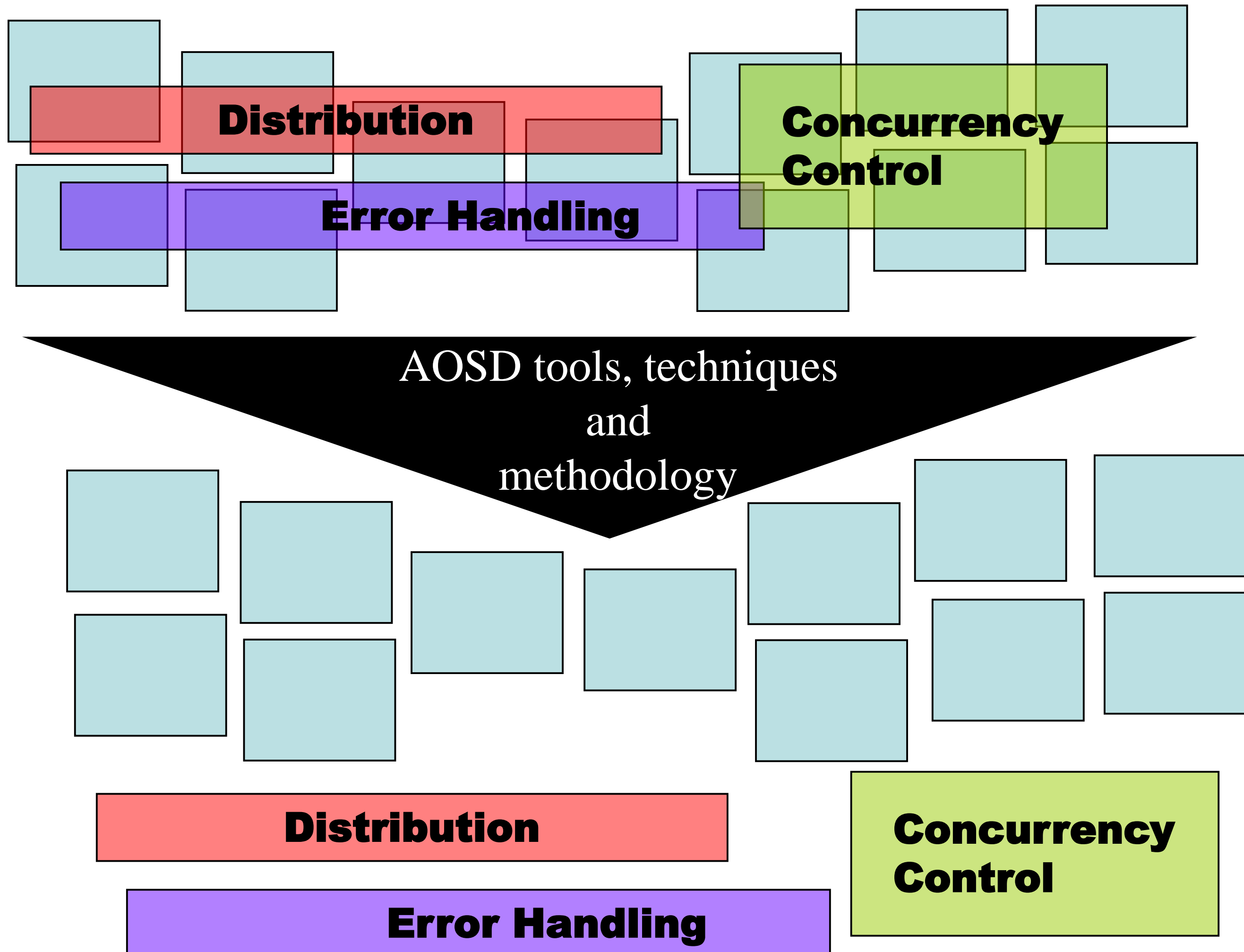
- Bad modularization of system-wide, peripheral concerns at the implementation level

- Code tangling
  - a module (e.g. SomeBusinessClass) handles multiple concerns simultaneously
- Code scattering
  - Duplicated code blocks
    - several modules contain repeated code of a nearly identical nature
  - Complementary code blocks
    - several modules implement complementary parts of the concern

From your experience…

- Which other system concerns are “typically” crosscutting?

Why Aspect-Oriented Software Development (AOSD)?



A Definition of AOSD

- **AOSD**: systematic *identification*, *modularisation*, *representation* and *composition* of crosscutting concerns [1]

Obs.: this definition is *problem-oriented* rather than *solution-oriented*

[1] Rashid, A., Moreira, A., Araujo, J. “Modularisation and Composition of Aspectual Requirements”, Proceedings of 2nd International Conference on Aspect-Oriented Software Development, ACM, 2003.

Revisiting the Tracing Example

```
class A {
   // some attributes
   void m1( ) {
         System.out.println("Entering A.m1( )");
         // method code
         System.out.println("Leaving A.m1( )");
   }

   String m2( ) {
         System.out.println("Entering A.m2( )");
         // method code
         System.out.println("Leaving A.m2( )");
         // return a string
   }
```

```
class B {
   // some attributes
   void m2( ) {
         System.out.println("Entering B.m2( )");
         // method code
         System.out.println("Leaving B.m2( )");
   }

   int m3( ) {
         System.out.println("Entering B.m3( )");
         // method code
         System.out.println("Leaving B.m3( )");
         // return an integer
   }
```

Wouldn't it be Nice if …

```
class A {
  // some attributes
  void m1( ) {
        // method code
}

  String m2( ) {
        // method code
        // return a string
  }
```

```
class B {
  // some attributes
  void m2( ) {
        // method code
        }

  int m3( ) {
        // method code
        // return an integer
  }
```

aspect Tracing {

when someone ***calls*** these methods

before the call {System.out.println("Entering " + methodSignature);}

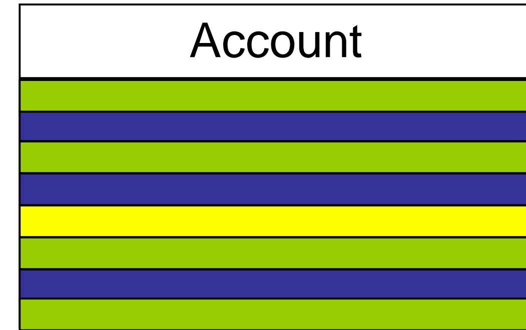
after the call {System.out.println("Leaving " + methodSignature);}

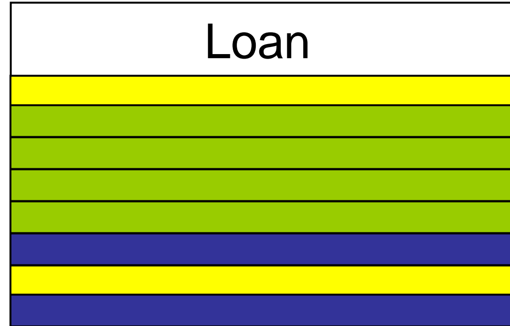
}

Revisiting the Bank Example

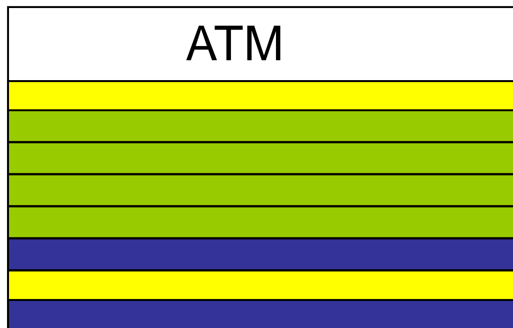
Primary Functionality | Persistence | Error Handling

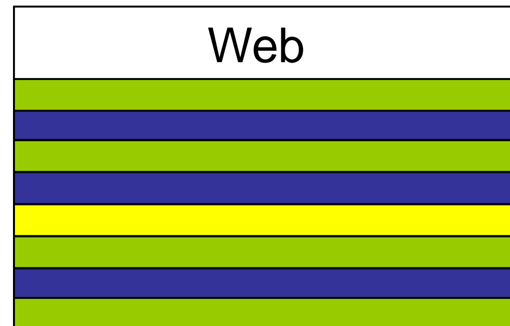
Data Classes

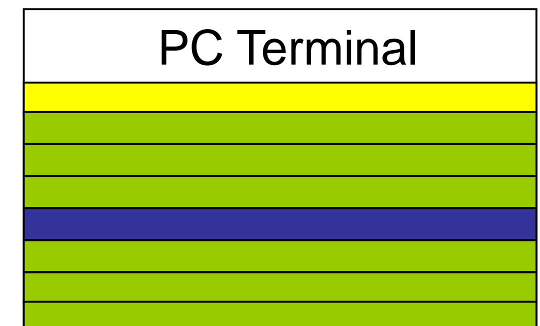




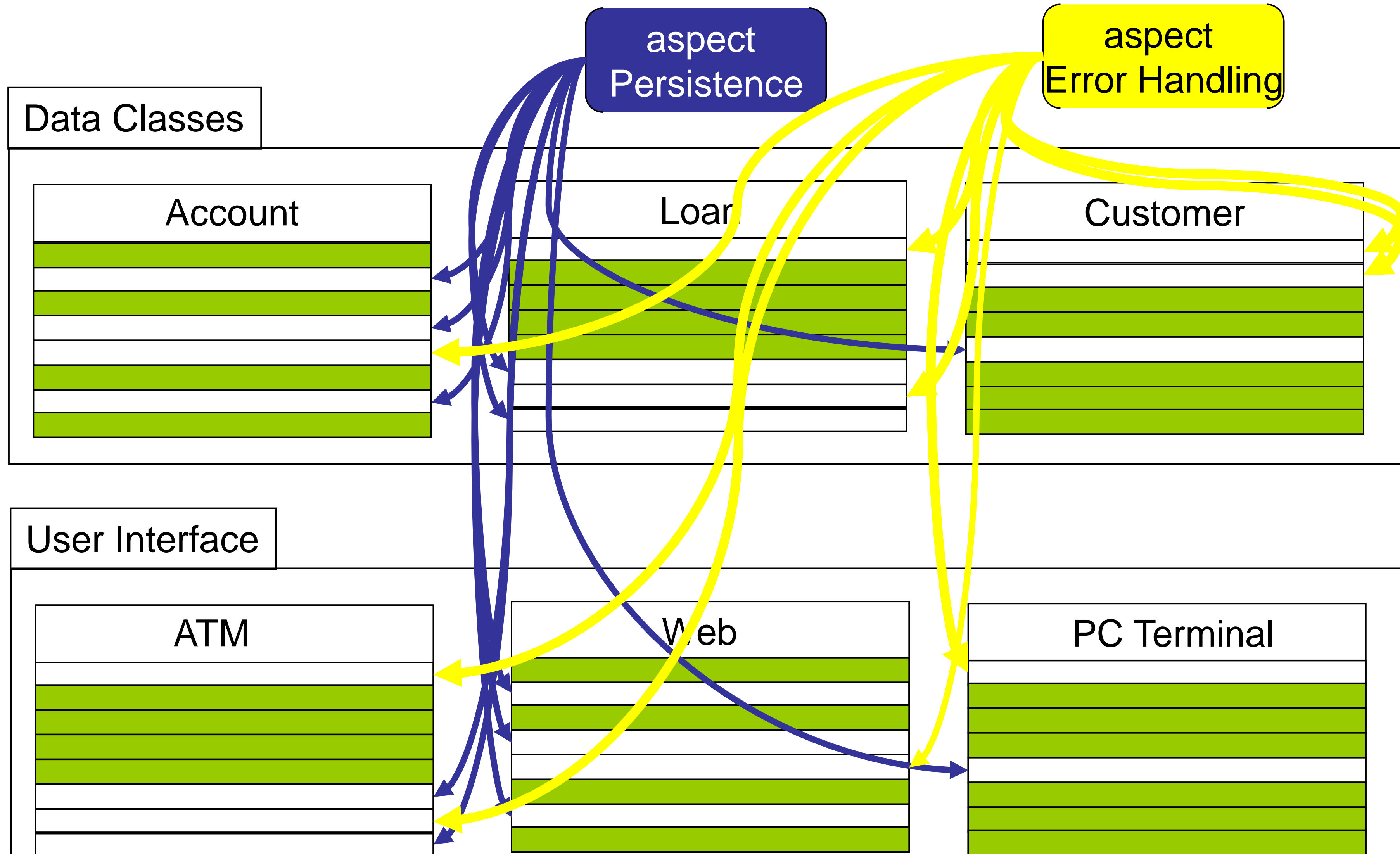
User Interface







Wouldn't it be Nice if …



Aspect-oriented programming (AOP)

- AOP is a new programming technique that allows programmers to modularize *crosscutting concerns*

- AOP defines a new modular unit, called aspect, for the modularization of crosscutting concerns

- AOP was proposed (and coined) by Gregor Kiczales and his research group from Xerox PARC (1996..1997)
  - AspectJ: 1st incarnation of general-purpose AOP
  - RIDL and COOL: 1st incarnations of domain-specific AOP
    - By Crista Lopes

AspectJ: Java Extension

- AspectJ provides additional keywords and constructs to write aspects
  - Aspects operate with reference to features in standard Java code

- An aspect can have ordinary Java code
  - Member variables and methods
  - Can implement interfaces
  - There is also aspect inheritance which we will discuss later

- AJDT: AspectJ Development Tools
  - An Eclipse plug-in for writing AspectJ programs

The Notion of a Join Point

```
class A {
  // some attributes
  void m1( ) {
        // method code
        // return a String
  }
```

```
class B {
  // some attributes
  void m2( ) {
        // method code

  int m3

  }
```

Type of Join Point

Pointcuts: Specific Join Points in this Program that we are Interested in

aspect Tracing {

when someone ***calls*** these methods

before the call {System.out.println("Entering " + methodSignature);}

after the call {System.out.println("Leaving " + methodSignature);}

}

AspectJ Join Point Model

- We looked at one type of join point
  - A Method Call
- Types of join points
  - Call to a method or constructor
  - Execution of a method or constructor
  - Getting or setting a class field
  - Exception throwing
  - Others which we will not discuss

Pointcut

- AspectJ construct
  - Specified using the **pointcut** keyword
- Used to specify which join points are of interest
  - pointcut constructorCall( ): call(Account.new(int));
  - pointcut getBalanceMethodCall( ): call(int getBalance( ));
  - pointcut setBalanceFieldValue( ): set(int balance);
  - pointcut setBalanceMethodExecution( ):
    execution(void setBalance(int));

Advice

- Method like construct
  - Used to specify behaviour we want to execute once a join point is matched by a specified pointcut
- Three types of advice
  - Before
  - After
  - Around
- Example:

```
before( ): constructorCall( ) {
   System.out.println("Constructor call about
                        to start");
}
```

After Advice

- Used to execute code after the code at the matched join point has executed

```
after( ): constructorCall( ) {
   System.out.println("Constructor call has
                          now finished");
}
```

Example (DisplayUpdating)

- Figure Editor

Display

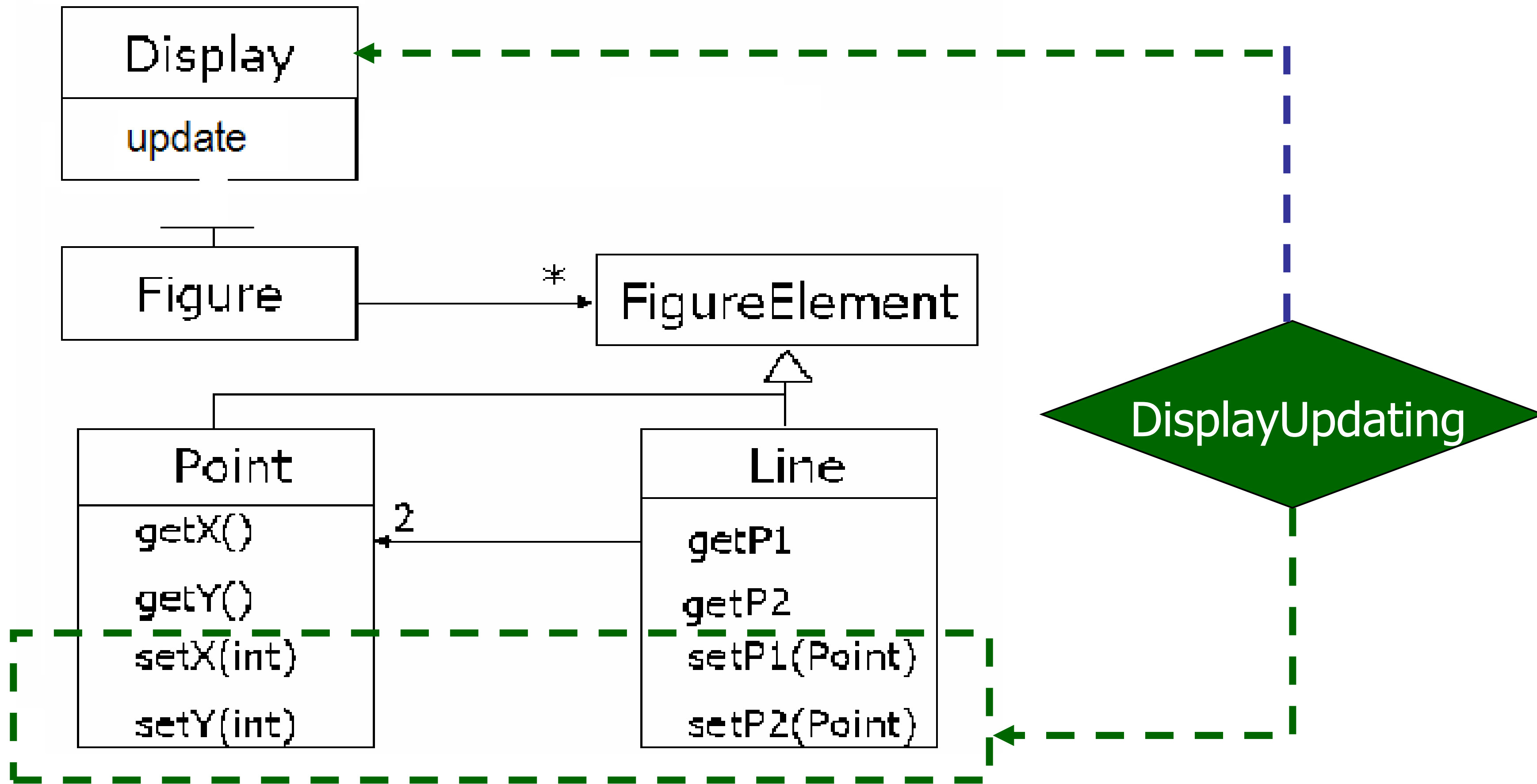
```
class Line {
  private Point p1, p2;

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

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

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

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

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

- Invasive modification
  - Calls to the update() method are scattered and tangled up in the code

Example
Figure Editor

DisplayUpdating:
Every time a FigureElement (Line ou Point) is modified, the Display must be updated.



Example
Figure Editor

```
class Line {
  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 {
  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; }
}
```

DisplayUpdating:

move(): for each call to methods C that modify some figure element:

```
void Point.setX(int),
void Point.setY(int),
void Line.setP1(Point),
void Line.setP2(Point)
```

AFTER the method call C,
execute Display.update()

Advice – Other Examples

```
before(): move() {
  System.out.println("An element will move.");
}
```

```
after() returning: move() {
   System.out.println("An element has moved. " };
}
```

```
void around(): move(FigureElement) {
  System.out.println("Enable Move = " + enableMove);
  if (enableMove) { proceed(); }
  System.out.printin("end of around advice");
}
```

Summary:
Outline of an AspectJ Aspect

```
aspect MyAspect {

    // pointcut definitions

    // advices

    // any Java members

    // inter-type declarations
    // to be discussed later
}
```

Using AJDT

- Similar to using Eclipse to write Java code
- There is:
  - An editor for AspectJ
  - AspectJ compiler

AspectJ HelloWorld

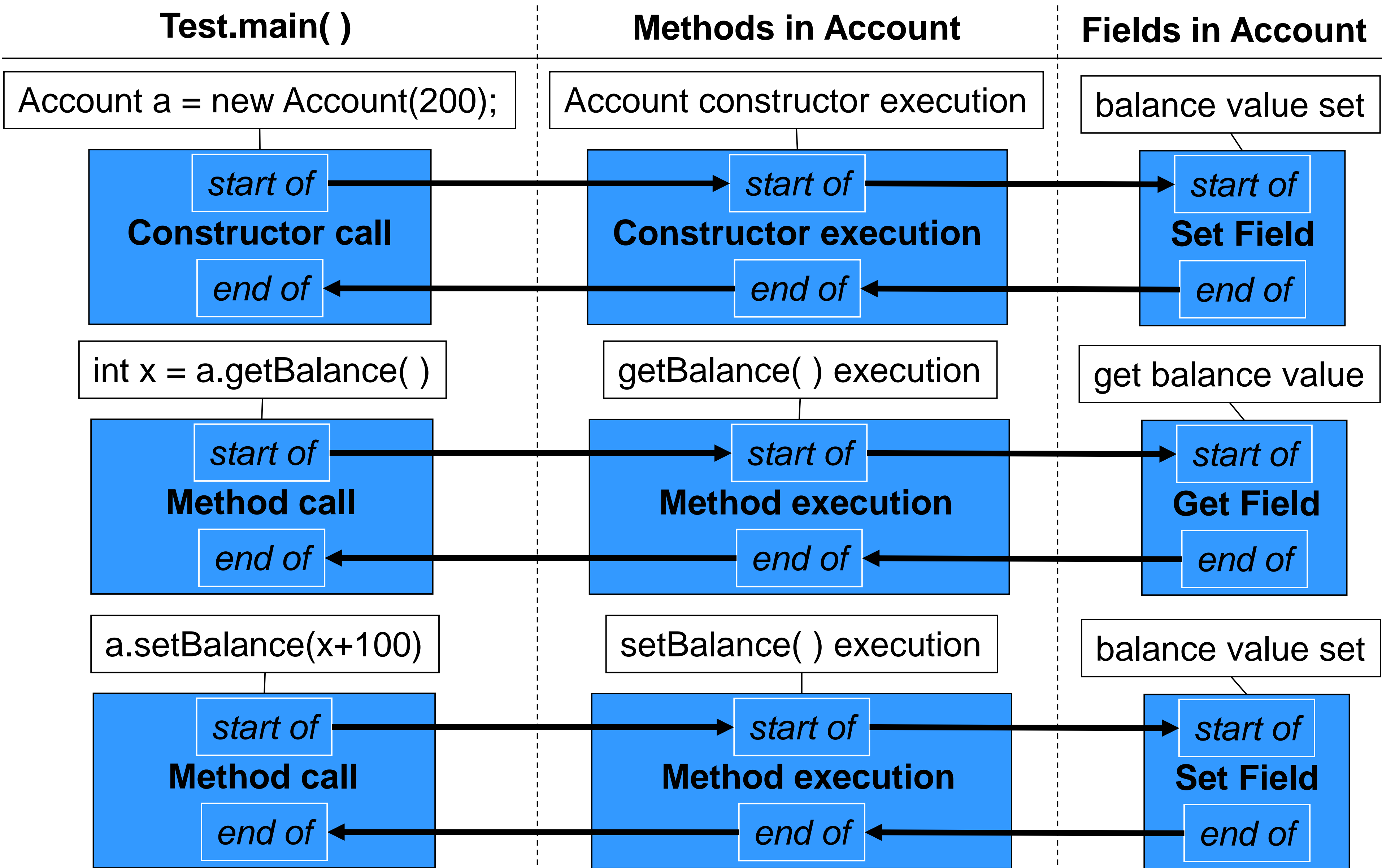
- Download the HelloWorld code from:
  - http://www.inf.puc-rio.br/~inf2007/
- Open the HelloWorld Java project in Eclipse
  - Import the files
  - Convert to AspectJ: click on the project, and choose Configure -> Convert…
- Write an aspect that does the following:
  - Captures the call to printMessage method and prints a message before and after to confirm it has captured the call.
  - ***Note:*** *You may want to use the call pointcut designator and before, after advice*

Understanding the Join Point Model (1)

```
class Account {

   private int balance;

   public Account(int startingBalance) {
            this.balance = startingBalance;
   }

   public void setBalance(int newBalance) {
            this.balance = newBalance;
   }

   public int getBalance( ) {
            return this.balance;
   }
}
```

```
class Test {

   public static void main(String args[ ]) {

            Account a = new Account(200);
            int x = a.getBalance( );
            a.setBalance(x+100);

   }
}
```

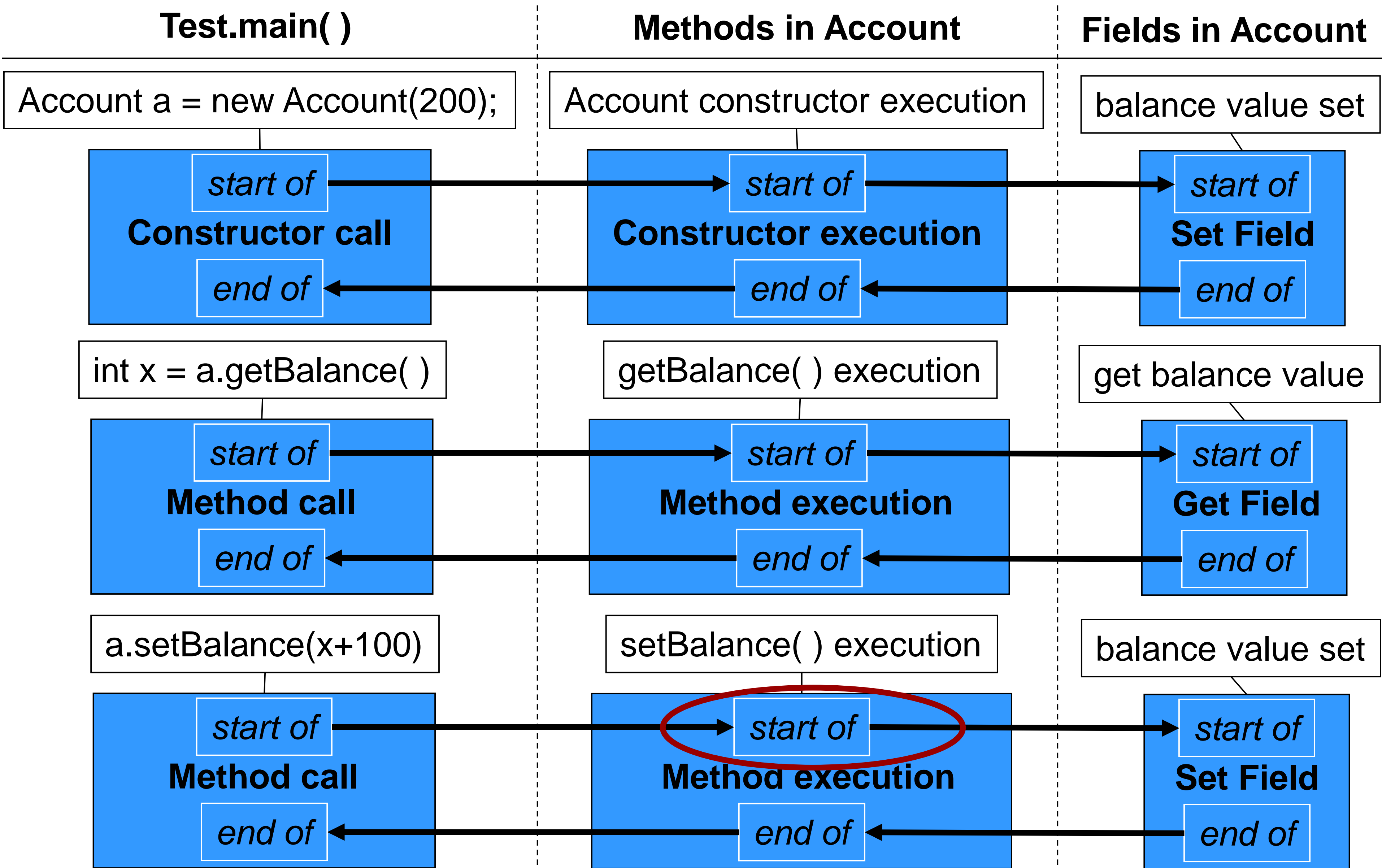
The Call Graph in Our Example



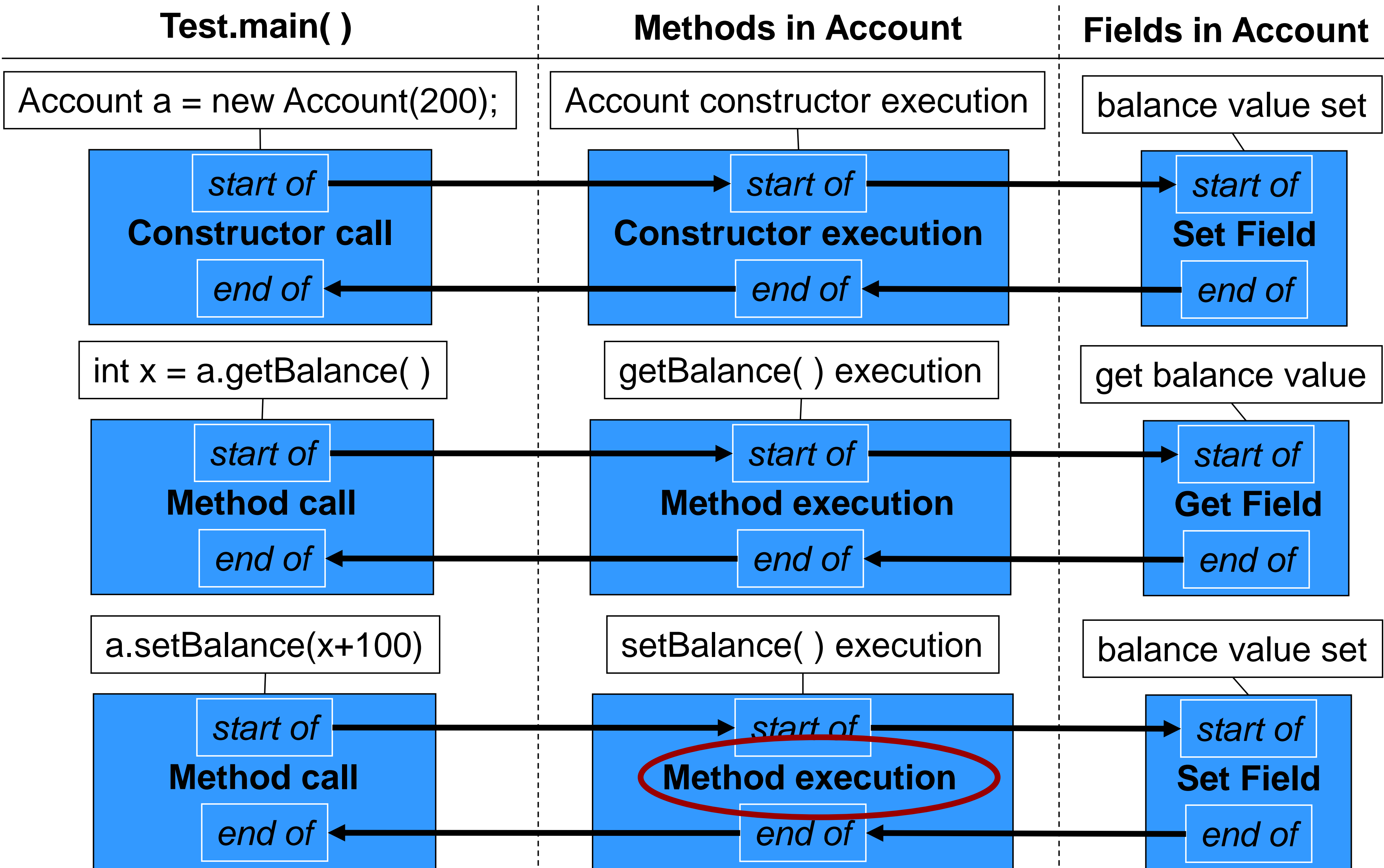
Around Advice: Wrapping Around a Join Point

```
void around( ): setBalanceMethodExecution( ) {

    System.out.println(“Who gives you extra?”);

    proceed( );

    System.out.println(“Howard does”);
}
```

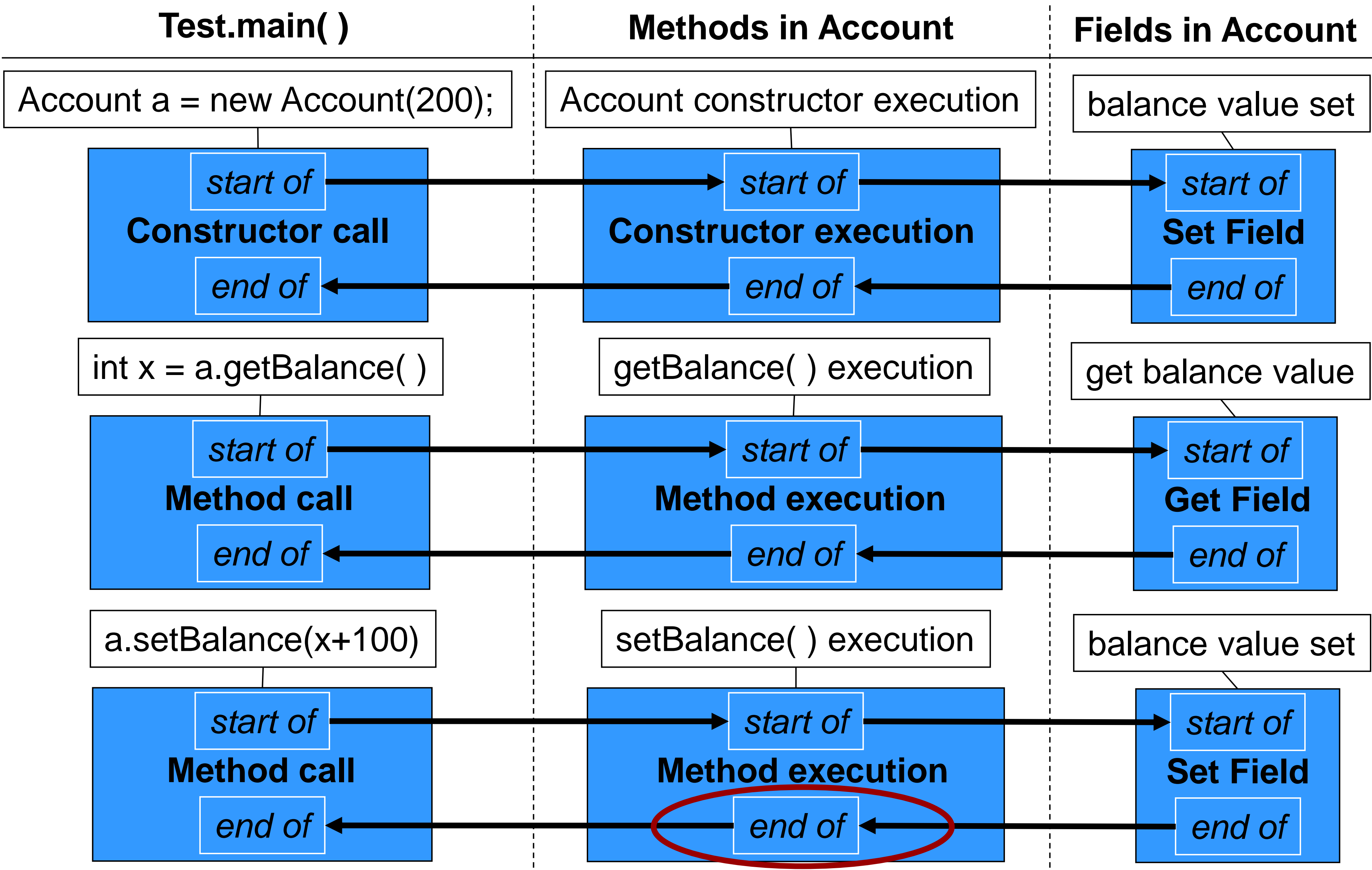
The Call Graph in Our Example



The Call Graph in Our Example



The Call Graph in Our Example



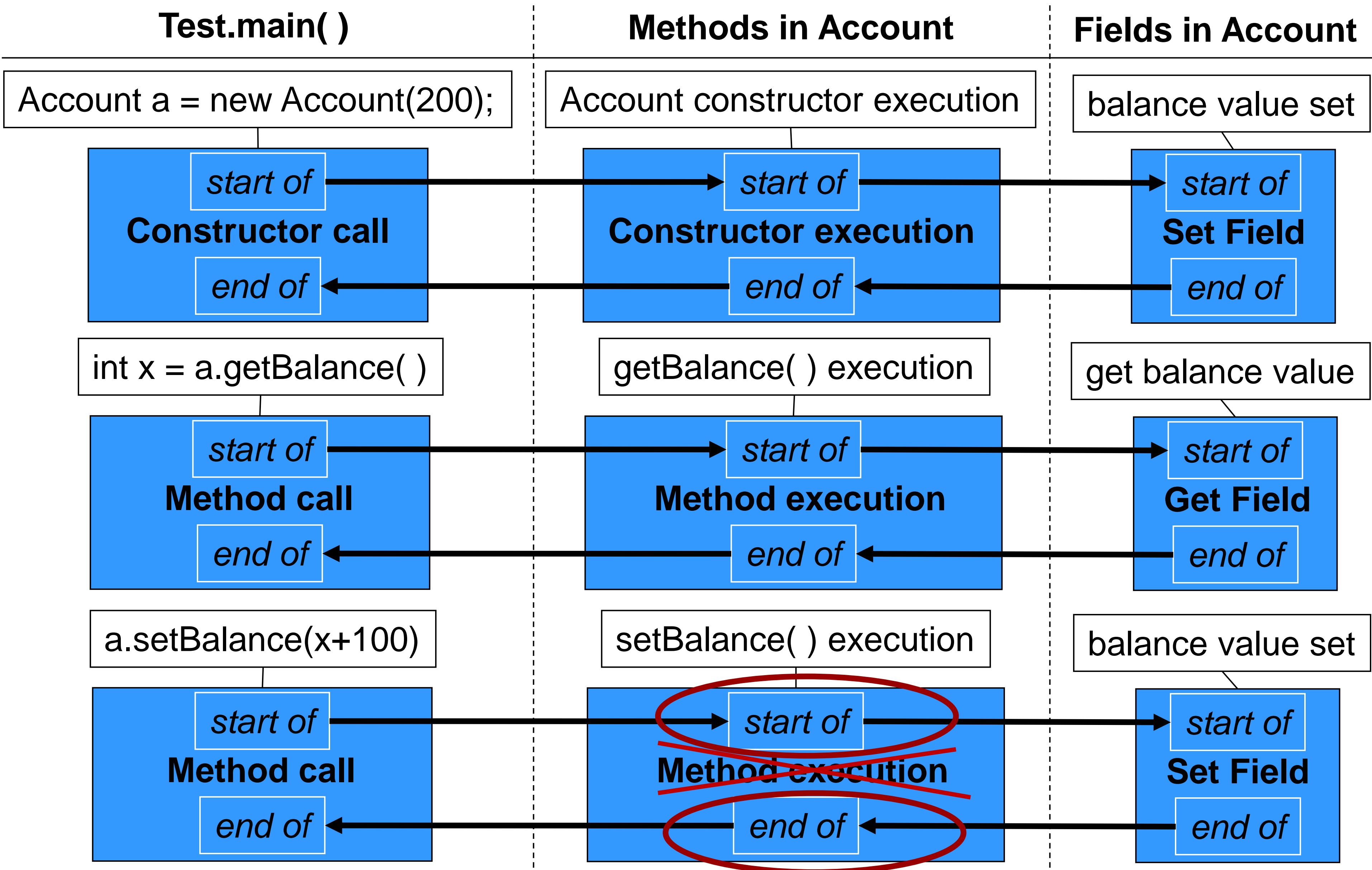
Around Advice (1)

- Can be used in two ways
  - Wrap code around the execution of the code for the matched join point
    - Requires a call to special **proceed** method in the advice body
    - Can be thought of as a combined **before** and **after** advice
  - Circumvent the code that would otherwise have executed at the matched join point
    - Requires that there is no call to **proceed** in the advice body

Around Advice: Circumventing a Join Point

```
void around( ): setBalanceMethodExecution( ) {

   System.out.println(“I won’t let you change
                                  your balance”);

}
```

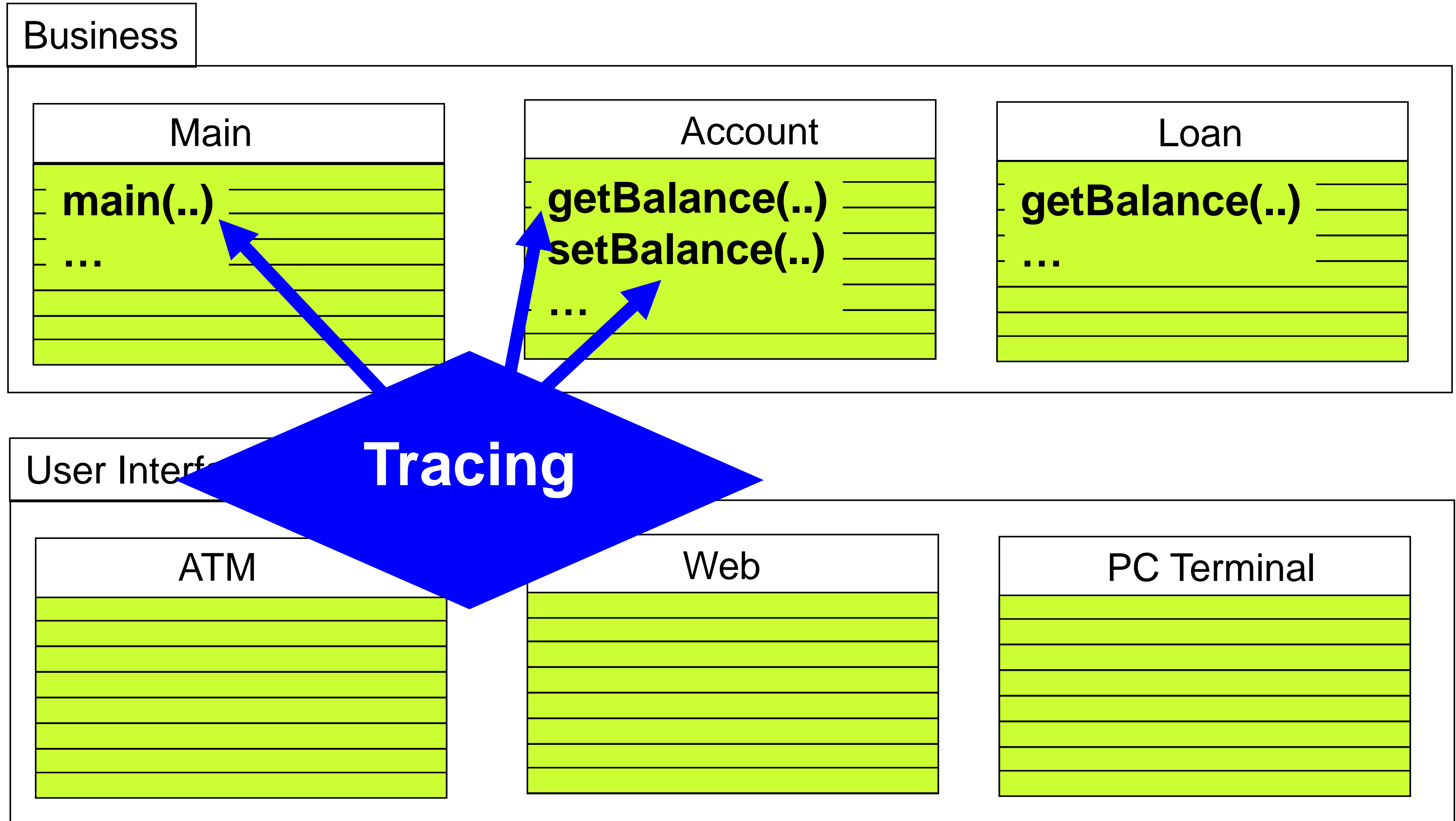
The Call Graph in Our Example



Tracing Exercise

- Open the TracingExample application in Eclipse
- Convert it to an AspectJ project
- Write a simple tracing aspect
  - Print trace messages before and after using **around**:
    - The setting of the balance field in Account
    - The call to getBalance() method in Account
    - The execution of the setBalance() method in Account

Revisiting the Tracing Example



Revisiting the Tracing Example (1)

- We have the following pointcut
  - pointcut **getBalanceMethodCall( ): call(int getBalance( ));**
  - Captures all calls to the *getBalance( )* method in our *Account* class
- Suppose we had another class *Loan* in the system
  - *Loan* also has a *getBalance( )* method
  - To find the amount still payable on a loan

Revisiting the Tracing Example (2)

Account
…
… getBalance( ): int …

Loan
…
… getBalance( ): int …

Matches
this join point

Matches
this join point too?

pointcut getBalanceMethodCall( ): call(int getBalance( ));

target pointcut designator

- *target(<type>)*
  - Used to identify the type of the object which is the target of a call
  - Used in conjunction with other pointcut designators

```
pointcut getBalanceMethodCall( ): call(int getBalance( ))
                                   && target(Account);
```

Note the **conjunction.** Used to combine multiple pointcut matches.

Only matches calls to getBalance( ) in objects of Account

target pointcut designator

- *target(<type>)*
  - Used to identify the type of the object which is the target of a call
  - Used in conjunction with other pointcut designators

```
pointcut getBalanceMethodCall( ): call(int getBalance( ))
                                      && target(Loan);
```

Note the **conjunction.** Used to combine multiple pointcut matches.

Only matches calls to getBalance( ) in objects of Loan

target pointcut designator

- *target(<type>)*
  - Used to identify the type of the object which is the target of a call
  - Used in conjunction with other pointcut designators

```
pointcut getBalanceMethodCall( ): call(int getBalance( ))
                                   && target(Loan);
```

What should I change to match calls to getBalance( ) in both objects of Loan and Account?

target pointcut designator

- *target(<type>)*
  - Used to identify the type of the object which is the target of a call
  - Used in conjunction with other pointcut designators

pointcut getBalanceMethodCall( ): call(int getBalance( ))

&& target(Loan)
&& target(Account);

Correct?

target pointcut designator

- *target(<type>)*
  - Used to identify the type of the object which is the target of a call
  - Used in conjunction with other pointcut designators

```
pointcut getBalanceMethodCall( ): call(int getBalance( ))
    && (
        target(Account)
        ||
        target(Loan)
        );
```

Note the **disjunction.** Used to combine multiple pointcut matches.

Matches calls to getBalance( ) in objects of Account and Loan

Exposing the Actual Instance

- *target(Account)*
  - Just determines that the target object is of type Account
  - What if you wanted to get a reference to the actual instance that receives the call
    - So that you can use it within your advice

```
pointcut getBalanceMethodCall(Account acc):
                                        call(int getBalance( ))
                                        && target(acc);
before(Account acc): getBalanceMethodCall(acc) {
        // use acc as reference
}
```

Exposing the Actual Instance

- You can expose the *this* instance in the same way

```
pointcut getBalanceMethodCall(Account acc, ATM atm):
                                        call(int getBalance( ))
                                        && target(acc)
                                        && this(atm);


before(Account acc, ATM atm):
            getBalanceMethodCall(acc, atm) {
      // use acc and atm references
}
```

Exercise

- Update your TracingExample as follows:
  - Add a print( ) method to the Account class
  - Add a Loan class with a print( ) method

```
public class Loan {
    int amount;
    public Loan(int amount) {
            this.amount = amount;
    }
    public int getBalance() {
            return this.amount;
    }
    public void setBalance(int balance) {
            this.amount = balance;
    }
    public String print() {
            return "Loan amount is " + this.amount;
    }
}
```

Exercise (continued)

- Change your pointcuts and advice to do the following:
  - Limit the context of getBalance calls to the Account class only
  - Expose the target of the call and print the Account object using the print( ) method

Which are the properties that characterise an aspect-oriented language?

- Quantification (yes: *common sense*)
- Obliviousness (*questionable*)
- Dependency Inversion (yes: *common sense*)
- Aspect-base dichotomy (no: almost common sense, but *theoretically questionable*)
  - explicit abstractions for aspectual and non-aspectual modules
    - No: historically-speaking
    - Yes: if symmetric languages (e.g. Hyper/J) would not be considered under the AOSD field

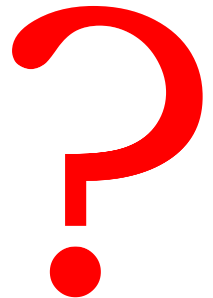
Making Pointcuts More Generic

Alessandro Garcia

© Alessandro Garcia, 2009

So far …

- We have used pointcuts in a restrictive way
  - pointcut getBalanceMethodCall( ): call(int getBalance( ));
  - pointcut getAmountMethodCall( ): call(int getAmount( ));
    - What is the problem with that?



- need to write a pointcut for each method call we are interested in

Revisiting the Tracing Example

- What if we want to trace *calls* to all getter methods in class Account?

```
pointcut getBalanceCalls( ): call(int getBalance()) && target(Account);

pointcut getNameCalls( ): call(String getName()) && target(Account);

pointcut getOwnerCalls( ): call(Owner getOwner()) && target(Account);

pointcut getCodeCalls( ): call(Sort getSortCode()) && target(Account);

…
```

Revisiting the Tracing Example

- What if we want to trace *calls* to all getter and setter methods in class Account?

```
pointcut getterCalls( ): call(* get*(  )) && target(Account);
```

Will match methods with any return type
names starting with *get*, e.g., getName, getAge, etc.
with 0 arguments

- … how to define a generic pointcut to capture setter calls?

```
pointcut setterCalls( ): call(* set*(..)) && target(Account);
```

Matching Arguments with Wildcards

- Capturing contextual information in generic pointcuts

```
pointcut getterCalls(Account acc, int value):  call(* get*(..))
                                                 && target(acc)
                                                 && args(value);
```

Matches all getter methods in Account with a single int argument and exposes the value

Note: You can, of course, use args to access multiple values

Matching Arguments with Wildcards

Suppose we have a class Customer with the *following constructor*:

```
public Customer(String firstName, String lastName, int age, Date dob)
```

```
pointcut newCustomer(String firstName, String lastName,
                     int age, Date dob):

                     call(Customer.new(String, String, int, Date))
                     && args(firstName, lastName, age, dob);
```

Reflective Features of AspectJ

- AspectJ has a reflection API
- We will look at two specific features
  - thisJoinPointStaticPart
  - thisJoinPoint

thisJoinPointStaticPart

- A special variable available in advice code
  - Just like ***this*** in Java
  - Gives access to information about a join point that can be determined at compile time
    - Kind of join point
      - Method call, method execution, constructor call, …
    - Source location
      - <<className>>.<<fileExtension>>:<<#LoC>>
    - Signature of join point
      - Method signature, constructor signature, field definition, …

thisJoinPointStaticPart

```
pointcut getterCalls( ): call(* get*(..)) && target(Account);

before( ): getterCalls( ) {

    System.out.println(“Kind: ”+ thisJoinPointStaticPart.getKind( ));

    System.out.println(“Signature: ” +
                                thisJoinPointStaticPart.getSignature( ));

    System.out.println(“Source Location:” +
                                thisJoinPointStaticPart.getSourceLocation( ));

}
```

Console:

```
Kind: method-call
Signature: void banking.Account.getBalance()
SourceLocation: Main.java:23
...
```

thisJoinPoint

- Similar to thisJoinPointStaticPart
  - With the difference that it has access to runtime information
  - Additional methods
    - getArgs( )
      - the actual argument values of the join point
    - getTarget( )
      - the target object
    - getThis( )
      - the currently executing object

thisJoinPoint

```
pointcut getterCalls( ): call(* get*(..)) && target(Account);

before( ): getterCalls( ) {

    Object[] args = thisJoinPoint.getArgs( );
    if (args.length > 0) System.out.println("1st Arg: " + args[0] );
            else System.out.println("No arguments!");

    System.out.println("Target: " + thisJoinPoint.getTarget( ));

    System.out.println("This: " + thisJoinPoint.getThis( ));

}
```

Console:

```
        No arguments!
        Target: banking.Account
        This: banking.Main
        ...
```

Exercise
Using wildcards: .., *,

- Open your TracingExample project and write **one** pointcut to:
  - Trap calls to constructors of Loan and Account
  - Trap all calls to getter methods
  - Trap all calls to setter methods
  - Print the message “Entering” added to the signature of the method before the calls
  - Print the message “Leaving” added to the signature of the method after the calls
  - **Note: You might want to write a few pointcuts and then combine them into a single one using the boolean operators**

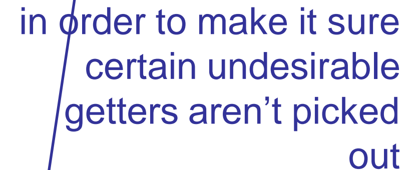
  - **20 minutes**

A Possible Solution

```
public aspect Tracing {

        pointcut constructorCalls(): call(Account.new(..)) || call(Loan.new(..));

        pointcut getterCalls(): call(* get*(..)) && (target(Account) || target(Loan));

        pointcut setterCalls(): call(* set*(..)) && target(Account) || target(Loan);

        pointcut tracer(): constructorCalls() ||  getterCalls() || setterCalls();

        before(): tracer() {

                System.out.println("Entering..." + thisJoinPointStaticPart.getSignature());

        }

        after(): tracer() {

                System.out.println("Leaving..." + thisJoinPointStaticPart.getSignature());
                System.out.println(" ");

        }
}
```

A Possible Solution - 1

```
public aspect Tracing {

        pointcut constructorCalls(): call(Account.new(..)) || call(Loan.new(..));

        pointcut getterCalls(): call(* get*(..)) && (target(Account) || target(Loan));

        pointcut setterCalls(): call(* set*(..)) && target(Account) || target(Loan);

        pointcut tracer(): constructorCalls() ||  getterCalls() || setterCalls();

        before(): tracer() {

                System.out.println("Entering..." + thisJoinPointStaticPart.getSignature());

        }

        after(): tracer() {

                System.out.println("Leaving..." + thisJoinPointStaticPart.getSignature());
                System.out.println(" ");

        }
}
```



A Possible Solution - 2

```
public aspect Tracing {

        pointcut constructorCalls(): call(Account.new(..)) || call(Loan.new(..));

        pointcut getterCalls(): call(* get*(..));

        pointcut setterCalls(): call(* set*(..));

        pointcut tracer(): (within(!Tracing)) && (constructorCalls() ||  getterCalls() || setterCalls();

        before(): tracer() {

                System.out.println("Entering..." + thisJoinPointStaticPart.getSignature());

        }

        after(): tracer() {

                System.out.println("Leaving..." + thisJoinPointStaticPart.getSignature());
                System.out.println(" ");

        }
}
```

in order to make it sure certain undesirable getters aren't picked out

So Far…

- Recovering Contextual Information
  - This
  - Target
  - Capturing arguments (args)
- advice
  - after returning
  - after throwing
- making pointcuts more generic
  - Wildcards: .. , *
- reflective features in AspectJ
  - static and run-time information