Conclusão

Parte 1 – Qualidade de Software por Construção

Alessandro Garcia

Departamento de Informática

Avisos

- envio de *resumo da proposta de trabalho* ***até 30 de abril***
  - falta receber de alguns alunos
- apresentações (aprox. 50 mins) do estudo: **15, 22 e 27 de junho**
  - vide critérios nos slides de introdução da disciplina
  - motiva-se: envie slides com antecedência
  - assume-se: já deve estar trabalhando no tema agora
- *website* do curso no ar, com:

  http://www.inf.puc-rio.br/~inf2007/
  - slides das aulas

Introduction to AOSD (conclusion)

Alessandro Garcia

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

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**

  - **15..20 minutes**

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(" ");

        }
}
```

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

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

Context-based Pointcuts in AspectJ

Alessandro Garcia

Today

- More advanced pointcut designators
  - context-based pointcuts
- Abstract aspects
- Inter-type declarations

Compile-time vs Runtime Context

- Compile-time
  - Determined from the source code
  - Based on scoping information that can be extracted *statically*
    - within
    - withincode
- Runtime
  - Based on *runtime* information from the object call graph
    - cflow
    - cflowbelow

Compile-time Context

- Consider the following simple piece of code

```
class Foo {
    ClassA a = new ClassA( );
    ClassB b = new ClassB( );
    ClassC cFoo = new ClassC( );
   // some code
   public void bar( ) {
          b.methodOfB( );
          a.methodOfA( );
          cFoo.methodOfC( );
   }
}
```

```
class B {
   ClassC cB = new ClassC( );
   // some code
   public void methodOfB( ) {
          cB.methodOfC( );
   }
}
```

```
// In some other piece of code
Foo foo = new Foo( );
foo.bar( );
```

Using *within*

- *within(<type pattern>)*
- Suppose we are interested in all calls to **methodOfC( )** that happen when it is called from any code within **ClassB**

```
call(* methodOfC( ))
&& target(ClassC)
&& within(ClassB);
```

```
call(* ClassC.methodOfC( ))
&& within(ClassB);
```

Either one would work fine!

Using *within*

```
call(* ClassC.methodOfC( ))
&& within(!ClassA);

call(* ClassC.meth*( ))
&& within(comp.lancs.*);

call(* *.*(..))
&& within(ClassB);
```

Using *withincode*

- *withincode(<method signature>)*
- Similar to within but the context is specified as a method signature rather than a type pattern
  - Again, the method signature is a pattern and can use wild cards

```
call(* ClassC.methodOfC( ))
&& withincode(* ClassB.methodOf*(..));
```

Compile-time vs Runtime Context

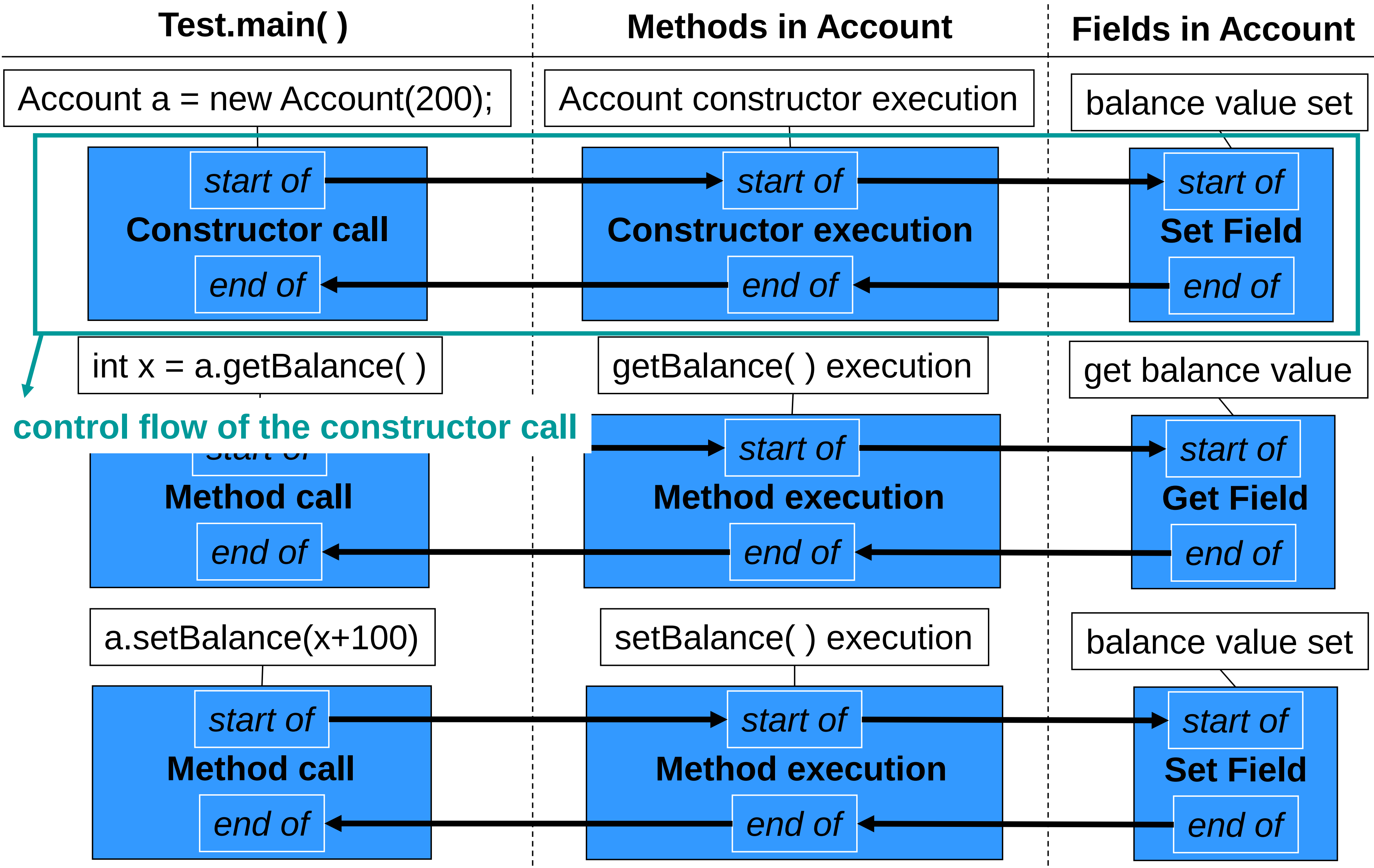
- Compile-time
  - Determined from the source code
  - Based on scoping information that can be extracted *statically*
    - within
    - withincode
- Runtime
  - Based on *runtime* information from the object call graph
    - cflow
    - cflowbelow

Understanding the Join Point Model (1)

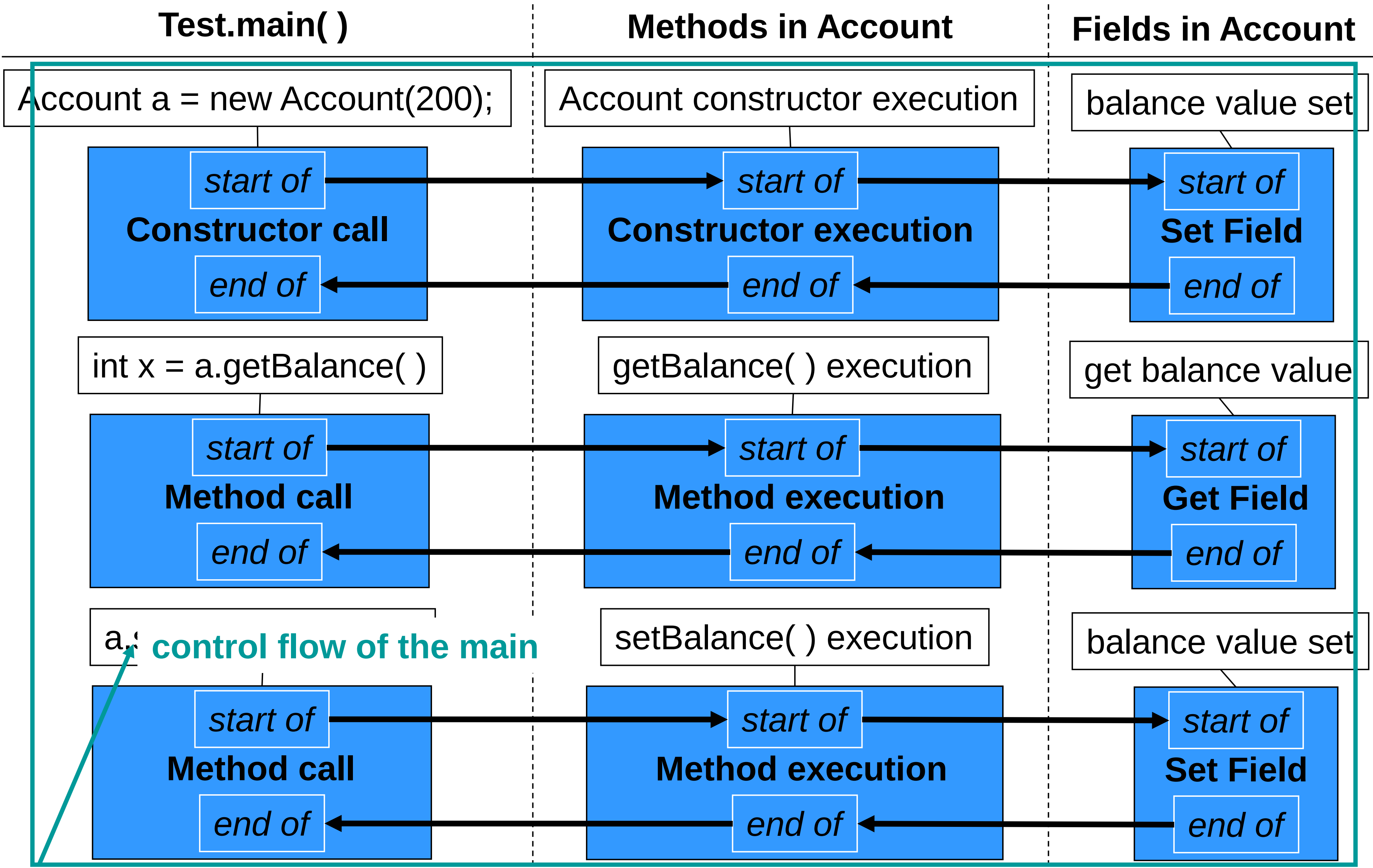
```
class Account {

   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



The Call Graph in Our Example



Runtime Context

- Consider the following simple piece of code

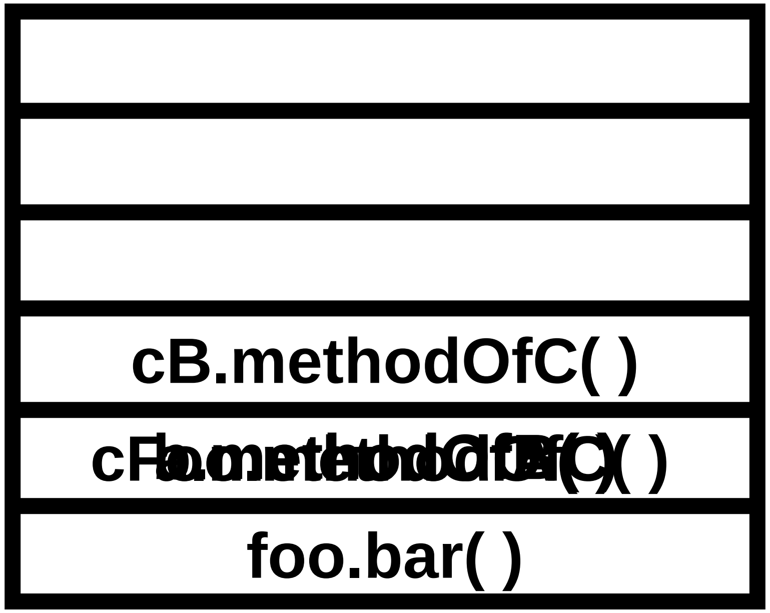
```
class Foo {

     ClassA a = new ClassA( );
     ClassB b = new ClassB( );
     ClassC cFoo = new ClassC( );
    // some code

    public void bar( ) {
            b.methodOfB( );
            a.methodOfA( );
            cFoo.methodOfC( );
    }
}
```

```
class B {
    ClassC cB = new ClassC( );
    // some code

    public void methodOfB( ) {
            cB.methodOfC( );
    }
}
```

```
// In some other piece of code
Foo foo = new Foo( );
foo.bar( );
```

A Simple Call Stack

- control flow of **foo.bar()**



```
class Foo {

     ClassA a = new ClassA( );
     ClassB b = new ClassB( );
     ClassC cFoo = new ClassC( );
    // some code

    public void bar( ) {
            b.methodOfB( );
            a.methodOfA( );
            cFoo.methodOfC( );
    }
}
```

```
class B {
    ClassC cB = new ClassC( );
    // some code

    public void methodOfB( ) {
            cB.methodOfC( );
    }
}
```

```
// In some other piece of code
Foo foo = new Foo( );
foo.bar( );
```

Using *cflow*

- Suppose we are interested in all calls to **ClassC.methodOfC( )** that happen in the context of the **bar( )** method in **Foo**
  - In our example these are calls to… ?

```
class Foo {

     ClassA a = new ClassA( );
     ClassB b = new ClassB( );
     ClassC cFoo = new ClassC( );
    // some code

    public void bar( ) {
            b.methodOfB( );
            a.methodOfA( );
            cFoo.methodOfC( );
    }
}
```

```
class B {
    ClassC cB = new ClassC( );
    // some code

    public void methodOfB( ) {
            cB.methodOfC( );
    }
}
```

```
// In some other piece of code
Foo foo = new Foo( );
foo.bar( );
```

Using *cflow*

- *cflow(<pointcut specification>)*
- cflow pointcut descriptor takes as argument other pointcut descriptors

We are interested in calls to methodOfC in objects of ClassC

We want to specify that happens during

We know that we want to restrict to bar( ) in Foo

```
call(* methodOfC( ))
&& target(ClassC)

&& cflow(

execution(void bar( ))
&& this(Foo)

);
```

Using *cflow*

- We could equally well use cflow to help us capture all calls to **methodOfC( )** in objects of **ClassC *except the calls that happen in the context of Foo.bar( )***

```
call(* methodOfC( ))
&& target(ClassC)
&& ! cflow(
                execution(void bar( ))
                && this(Foo)
        );
```

Using *cflow*

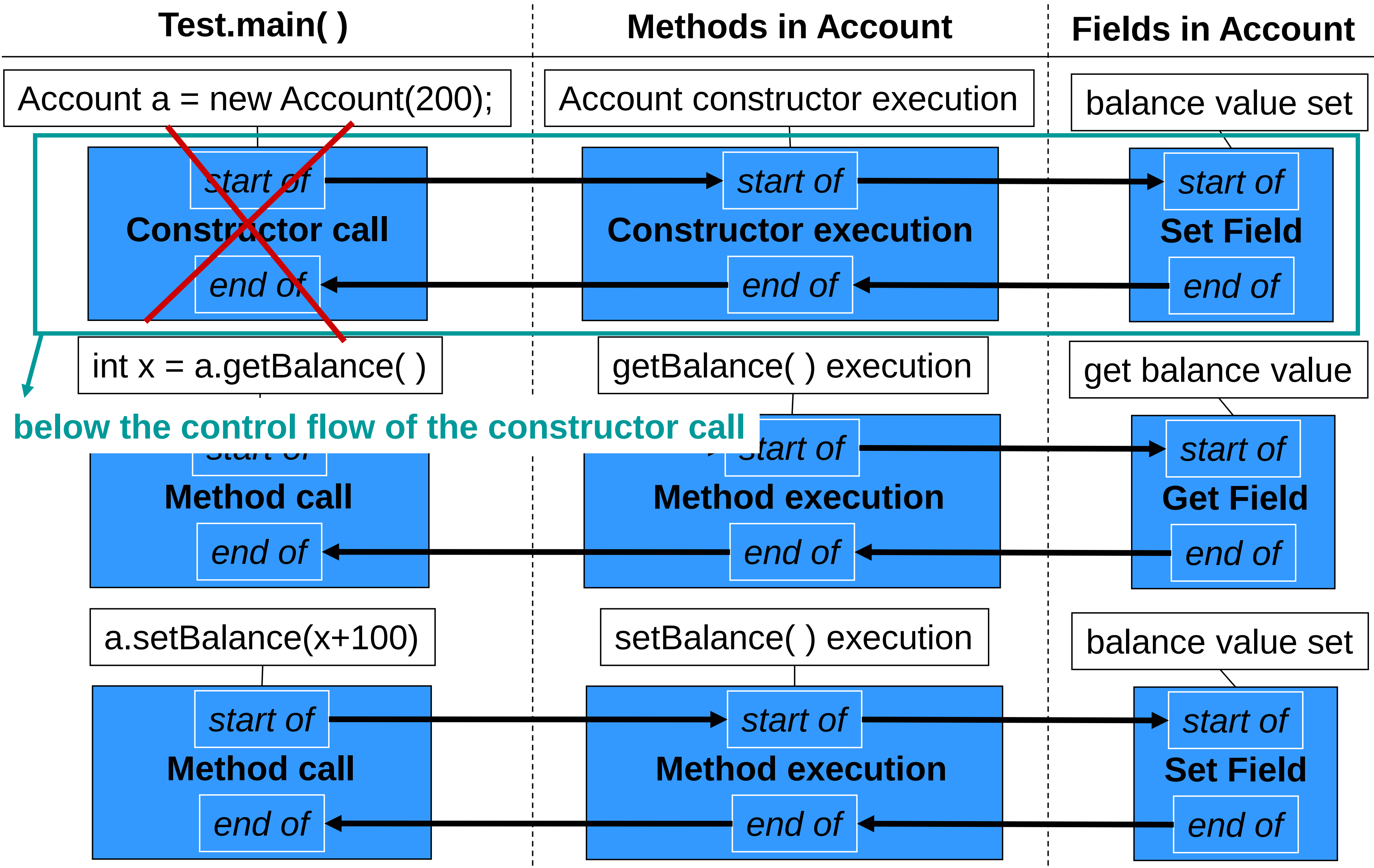
- You can of course use wild cards to provide a wider range of *cflow* matches

```
call(* Iterator.next*(..))
&&   cflow(
                call(* get*(..))
                && this(com.lancs.Fred)
         );
```

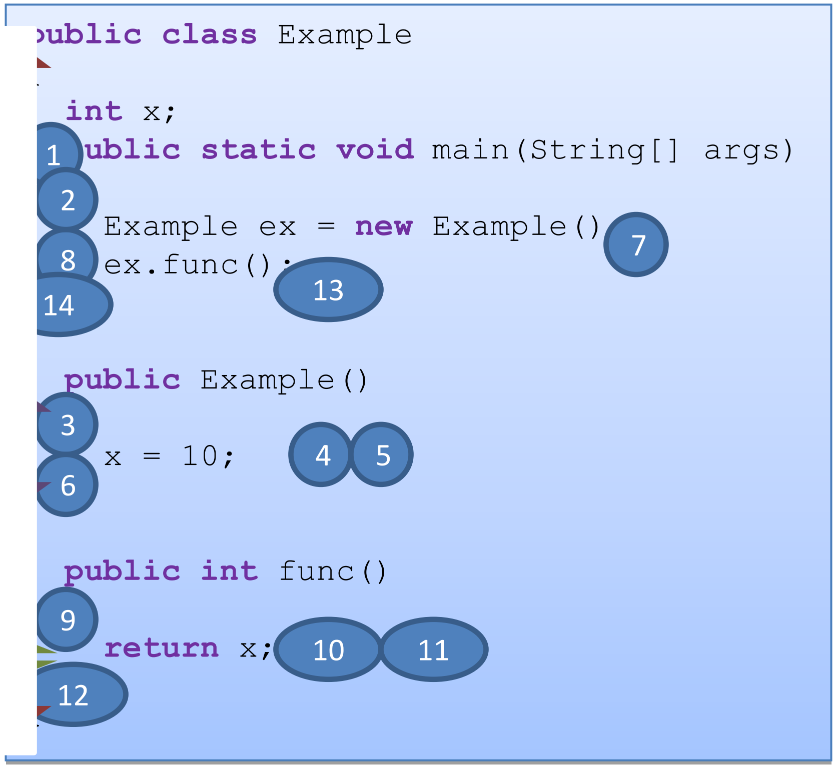
cflowbelow

- A variant of *cflow*
- Matches all join points except the one where the control flow begins
- Refer to AspectJ website documentation for a discussion of distinction between *cflow* and *cflowbelow*

The Call Graph in Our Example



Comparison: within, withincode, cflow, cflowbelow



```
ublic class Example

 int x;
ublic static void main(String[] args)

  Example ex = new Example()
  ex.func();


 public Example()

  x = 10;


 public int func()

  return x;
```

```
 1. Before execution(void Example.main(String[])
 2. Before call(Example())
 3. Before execution(Example())
 4. Before set(int Example.x)
 5. After set(int Example.x)
 6. After execution(Example())
 7. After call(Example())
 8. Before call(int Example.func())
 9. Before execution(int Example.func())
10. Before get(int Example.x)
11. After get(int Example.x)
12. After execution(int Example.func())
13. After call(int Example.func())
14. After execution(void Example.main(String[]))
```

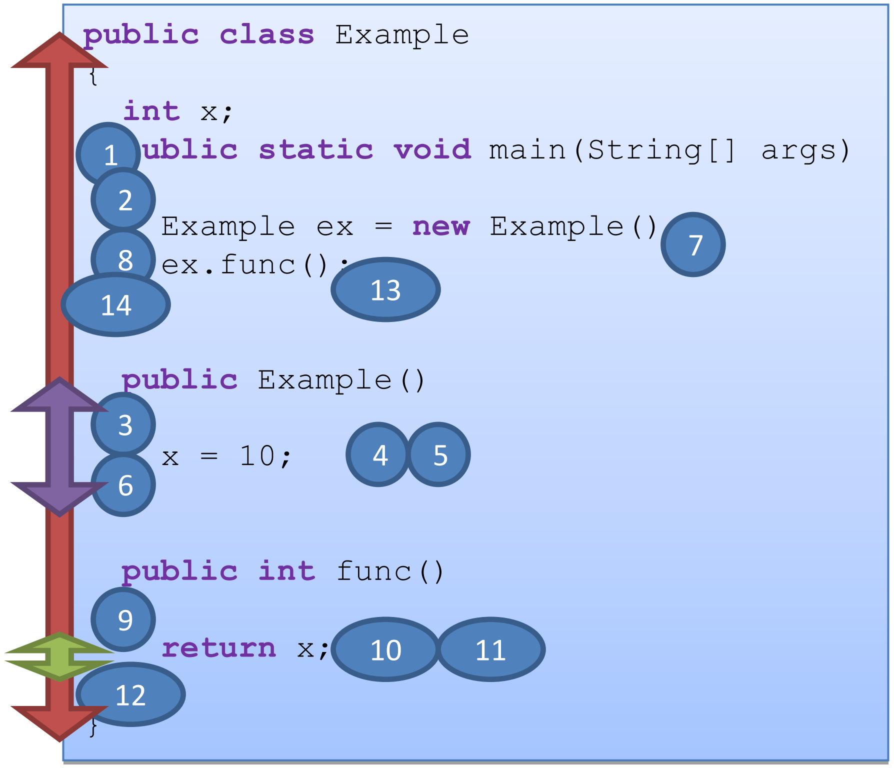
🟥 `within(Example)`

🟩 `withincode(int Example.func())`

🟩 `cflowbelow(execution(int Example.func()`

🟪 `cflow(execution(Example.new()))`

Comparison: within, withincode, cflow, cflowbelow



```
public class Example
{
  int x;
  public static void main(String[] args)

    Example ex = new Example()
    ex.func();


  public Example()

    x = 10;


  public int func()

    return x;

}
```

```
 1. Before execution(void Example.main(String[])
 2. Before call(Example())
 3. Before execution(Example())
 4. Before set(int Example.x)
 5. After set(int Example.x)
 6. After execution(Example())
 7. After call(Example())
 8. Before call(int Example.func())
 9. Before execution(int Example.func())
10. Before get(int Example.x)
11. After get(int Example.x)
12. After execution(int Example.func())
13. After call(int Example.func())
14. After execution(void Example.main(String[]))
```

1-14	🟥	within(Example)
10-11	🟩	withincode(int Example.func())
10-11	🟩	cflowbelow(execution(int Example.func()
3-6	🟪	cflow(execution(Example.new()))

Abstract Aspects and Pointcuts

- Aspects can inherit from each other
- Abstract aspects and pointcuts help us write more reusable aspects

```
abstract aspect MyAbstractAspect {

   abstract pointcut pc1( );

   before( ): pc1( ) {
           // do something
   }

   after( ): pc1( ) {
           // do something more
   }
}
```

```
aspect MyConcreteAspect
           extends MyAbstractAspect {

   pointcut pc1( ):
           call(* get*(..))
           && target(Account);

}
```

Note: You can only extend an abstract aspect and not a concrete aspect.

Abstract Transactional Aspect

- Make certain methods to be executed as a transaction

```
public abstract aspect TransactionalMethods {

    abstract public pointcut MethodToBeMadeTransactional();

    void around() : MethodToBeMadeTransactional() {
        ProceduralInterface.beginTransaction();
        boolean aborted = false;
        try {
            proceed();
        } catch (TransactionException e) {
                ProceduralInterface.abortTransaction();
                aborted = true;
                throw e;
        } finally {
                if (!aborted) {
                        ProceduralInterface.commitTrasaction();
                }
        }
     }
 }
```

Concrete Transactional Aspects

- Make the calls to “Account” methods to be executed as transactions

```
aspect MakeAccountMethodsTransactional extends TransactionalMethods {
    public pointcut MethodToBeMadeTransactional() :
    call (public * Account.*(..));
}
```

Inter-type Declarations

- Also known as *introductions*
- Means to inject additional code into classes
  - The introduced fields, methods, etc. ***belong*** to the class though are declared in an aspect

Inter-type Field Declarations

public Customer Account.owner;

Visibility: *public*, or *private*. Cannot be *protected*.

Type of the introduced field

Type in which the field is injected

The name of the field being introduced

```
public Customer Account.owner =
                    new Customer("John Doe");
```

Can also initialise the introduced field

Inter-type Method Declarations

private void Account.addMoney(int amount) { // code }

Visibility: *public*, or *private*. Cannot be *protected*.

Type in which the method is injected

Signature of the method. May include a throws clause

Note: *this* here will be bound to the Account instance on which the method is being executed and not to the aspect from which the introduction is done.

Inter-type Constructor Declarations

- Similar to inter-type method declarations

```
public  Account.new(int accoutID) { // code }
```

Visibility: *public*, *private* or default. Cannot be *protected*.

Type in which the constructor is injected

Signature of the constructor. May include a throws clause

Note: Would probably be good to introduce a field to capture the account ID supplied as an argument to the constructor.

Declaring Parents

- You can also use inter-type declarations to add *extends* or *implements* relationships to your classes
- Takes the form
  - declare parents: <type pattern> extends <type>;
  - declare parents: <type pattern> implements <type>;

Declaring Parents

```
declare parents:
(CheckingAccount || SavingsAccount) extends Account;
```

```
declare parents: AnInterface extends AnotherInterface;
```

```
declare parents: com.lancs.* implements Serializable;
```

Declaring Warnings

- A compile-time mechanism to warn developers of constraints on a program
- Takes the form:
  - declare warning: <pointcut expression>: “<warning message>”;

```
declare warning:
              call(String *.toString(..)):
                  “Not a good idea to call toString( ) methods”;
```

Declaring Errors

- A compile-time mechanism to ensure that developers don’t violate certain constraints
- Takes the form:
  - declare error: <pointcut expression>: “<error message>”;

```
declare error:
              call(String *.toString(..)):
                  “Not allowed to call toString( ) methods”;
```

Declaring Warnings and Errors

- A very good way to find relevant code for refactoring
  - Suppose you wanted to factor out all calls to **addListener( )** methods from your code

```
aspect HelpRefactor {
        declare warning: call(* *.addListener(..)):
                "Point out all calls to addListener methods";
}
```

When you build the project with Eclipse the compiler will warn you of all the place where the above warning applies. You can navigate to those points and factor out the calls into an aspect. Later you can declare it as an error so that other developers only put calls to addListener in an aspect.

Declaring Errors and Warnings

- Only suitable with pointcuts specification that can be evaluated at compile-time
  - Cannot be used with *this( ), args( ), target( ), cflow( )*, etc.
    - Depend on dynamic context

Aspect Interaction

```
aspect Security {
    pointcut getCalls( ): call(* get*(..)) && target(Account);
    before( ): getCalls( ) {
        // do security
    }
}
```

There is no guarantee which advise is going to be executed first!

```
aspect ShowBalance {
    pointcut getCalls( ): call(* get*(..)) && target(Account);
    before( ): getCalls( ) {
        // show remaining balance
    }
}
```

Declaring Precedence

- We would want the Security check to happen before the balance is shown by the ShowBalance aspect
- We can declare such precedence explicitly
  - Takes the form:
    - declare precedence: <type pattern list>;
  - Can have a declare precedence statement in any aspect
  - Good practice to have it as a separate aspect
    - Precedence cuts across aspects!

Declaring Precedence

```
aspect AspectPrecedence {
    declare precedence: Security, ShowBalance;
}
```

```
aspect AspectPrecedence {
    declare precedence: Security, *;
}
```

Always do Security check before anything else.

In Summary

- We have looked at a range of features of the AspectJ language
  - Notion of join point
  - Pointcuts and the various pointcut designators
  - Advice
  - Aspect inheritance
  - Inter-type declarations
  - Aspect interaction resolution

In Summary

- There are a number of features of AspectJ we have not covered
  - Some pointcut designators
    - Advice execution, etc.
  - Aspect instantiation models
    - Our aspects have been singletons
    - Aspects can be instantiated based on join point matching
  - Exception softening
  - Etc…

Contributions and Adoption of AOP

- IBM Websphere Application Server (WAS) uses AspectJ
- The core of JBoss Application Server (JBoss AS) is integrated with the JBoss AOP; used to deploy services such as security and transaction management
- Dependency injection (in Spring, e.g.) was recognizably an influence of AOP
- IBM, Motorola, SAP, Siemens, and Sun Microsystems, ASML have used AspectJ
- Glassbox is a troubleshooting agent for Java applications that automatically diagnoses common problems. The Glassbox inspector monitors the activity of the Java virtual machine using AspectJ
- .NET 3.5 supports Aspect Oriented concepts through the Unity container
- Etc…

Conclusão

Parte 1 – Qualidade de Software por Construção

Alessandro Garcia

Departamento de Informática