Ambientes de Desenvolvimento Avançados
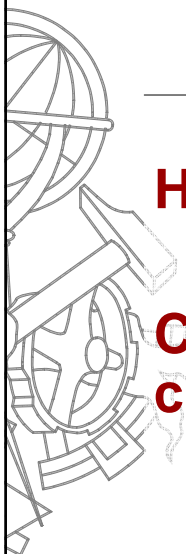
Aula 14
Engenharia Informática

2004/2005

José António Tavares
jrt@isep.ipp.pt
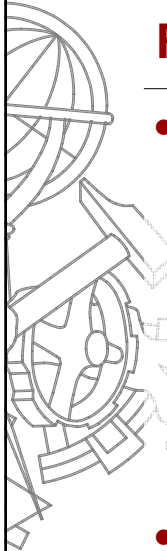
1

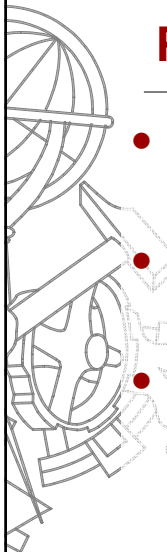# Herança *versus* Composição

Como implementar composição

1

Formas de Herança

- Three facets of inheritance
  - *Implementation inheritance*
    (sub-classing) sharing of implementation fragments
  - *Interface inheritance*
    (sub-typing) sharing of contract fragments
  - *Substitutivity*
    Promise of substitutability

- How to avoid inheritance ?

Problema da classe base frágil

- can super-class (base class) evolve without breaking subclasses?

- eg old applications with new revision of OS

- two issues: *syntactic* and *semantic* fragile base class problem

Problema da classe base frágil

- Semantic
  - How can a subclass remain valid in the presence of different version of its super-classes ?
    - Parameters
    - Methods name
    - Return type
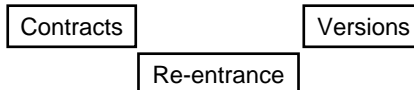
Contracts		Versions
	Re-entrance	

Das classes à composição de objectos

- motivation for *implementation inheritance* is flexible code reuse
- improving super-class improves sub-classes? re-entrance and up-calls make this difficult
- *object composition* a simpler alternative ('has-a' instead of 'is-a')
- *outer object* has the only reference to *inner object*
- outer object *forwards* messages to inner object
- improving inner object improves outer object
- object composition and forwarding a close approximation to implementation inheritance, without some of the problems

Das classes à composição de objectos

- Object composition is a much simpler form of composition than implementation inheritance;
- Shares several of the often quoted advantages of implementation inheritance;
- The idea is very simple – whenever an object does not have the means to perform some task locally, it can send messages to other objects, asking for support, and if the helping object is a part of the helped object, this is called *object composition*;
- An object is part of another one if references to it do note leave that object.

Das classes à composição de objectos

- Sending a message on from one object to another is called **forwarding (*re-encaminhamento*)**;
- The combination of object composition and forwarding comes fairly close to what is achieved by implementation inheritance;
- However, it does not get so close that it also has the disadvantages of implementation inheritance.
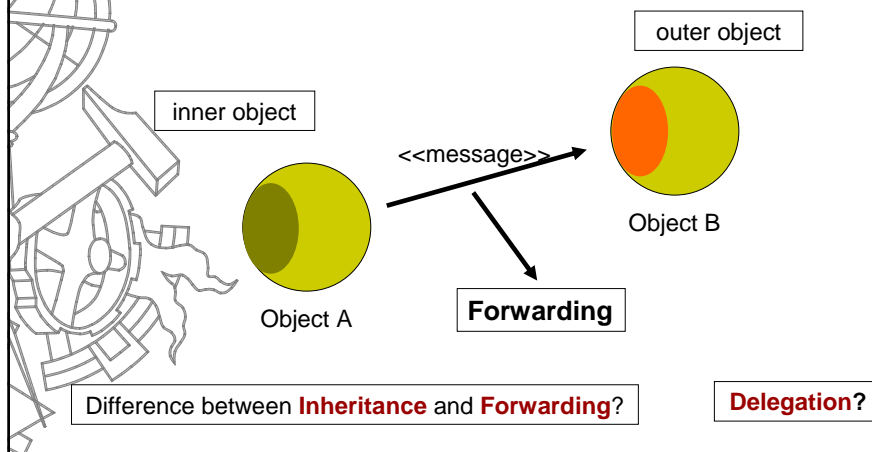
Das classes à composição de objectos

- An outer object does not re-implement the functionality of the inner object when it forwards messages;
- It reuses the implementation of the inner object;
- If the implementation of the inner object is changed, then this change will "spread" to the outer object;
- The difference between object composition with forwarding and implementation inheritance is called "implicit self-recursion" or "possession of a common self"

Das classes à composição de objectos

outer object

inner object

<<message>>

Object B

Object A

Forwarding

Difference between **Inheritance** and **Forwarding**?

Delegation?

Das classes à composição de objectos

Exemplo : Herança

```
class Fruit
{
    //...
}

class Apple extends Fruit
{
    //...
}
```
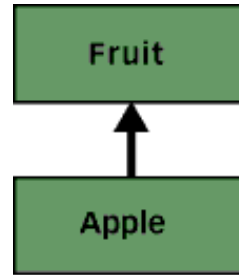
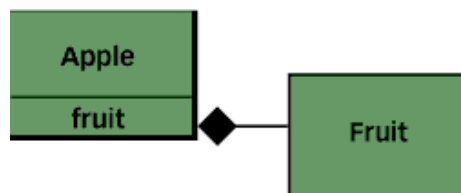# Das classes à composição de objectos

Exemplo : Composição

```
class Fruit
{
    //...
}

class Apple
{
    private Fruit fruit = new Fruit();
    //...
}
```

6

Das classes à composição de objectos

- **Changing the super-class interface**
  In an inheritance relationship, super-classes are often said to be "fragile," because one little change to a super-class can ripple out and require changes in many other places in the application's code.

Das classes à composição de objectos

- **The composition alternative**
  - Given that the inheritance relationship makes it hard to change the interface of a super-class, it is worth looking at an alternative approach provided by composition.
  - It turns out that when your goal is code reuse, composition provides an approach that yields easier-to-change code.

Das classes à composição de objectos

```
class Fruit {
    // Return int number of pieces of peel that
    // resulted from the peeling activity.
    public int peel() {
        System.out.println("Peeling is appealing.");
        return 1;
    }
}

class Apple extends Fruit {
}

class Example1 {
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Das classes à composição de objectos

Change the return value of peel() to type Peel, will break the code for Example1

```
class Peel {
    private int peelCount;

    public Peel(int peelCount) {
        this.peelCount = peelCount;
    }

    public int getPeelCount() {
        return peelCount;
    }
    //...
}
```

Das classes à composição de objectos

```
class Fruit {
    // Return a Peel object that
    // results from the peeling activity.
    public Peel peel() {
        System.out.println("Peeling is appealing.");
        return new Peel(1);
    }
}

// Apple still compiles and works fine
class Apple extends Fruit {
}

// This old implementation of Example1
// is broken and won't compile.
class Example1 {

    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Das classes à composição de objectos

- **Code reuse via composition**
  - Composition provides an alternative way for Apple to reuse Fruit's implementation of peel().
  - Instead of extending Fruit, Apple can hold a reference to a Fruit instance and define its own peel() method that simply invokes peel() on the Fruit.

Das classes à composição de objectos

```
class Fruit {
    // Return int number of pieces of peel that
    // resulted from the peeling activity.
    public int peel() {
        System.out.println("Peeling is appealing.");
        return 1;
    }
}

class Apple {
    private Fruit fruit = new Fruit();
    public int peel() {
        return fruit.peel();
    }
}

class Example2 {
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Original

Das classes à composição de objectos

Change the return value of peel() to type Peel, will break the code for Example1

```
class Peel {
  private int peelCount;

  public Peel(int peelCount) {
      this.peelCount = peelCount;
  }

  public int getPeelCount() {
      return peelCount;
  }
  //...
}
```

10

Das classes à composição de objectos

```
class Fruit {
    // Return int number of pieces of peel that resulted from the peeling activity.
    public Peel peel() {
        System.out.println("Peeling is appealing.");
        return new Peel(1);
    }
}
// Apple must be changed to accommodate the change to Fruit
class Apple {
    private Fruit fruit = new Fruit();
    public int peel() {
        Peel peel = fruit.peel();                    Alterado
        return peel.getPeelCount();
    }
}
// This old implementation of Example2 - still works fine.
class Example1 {
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Das classes à composição de objectos

- The composition approach to code reuse provides stronger encapsulation than inheritance, because a change to a back-end class needn't break any code that relies only on the front-end class.

- For example, changing the return type of Fruit's peel() method from the previous example doesn't force a change in Apple's interface and therefore needn't break Example2's code.

Das classes à composição de objectos

Possession of a common self

- instance of sub-class shares identity with that of its super-class;
- control can return from a super-class back to a sub-class – invocation of the last overriding version of the method;
- composition of objects has no single identity;
- once control passed from outer to inner object, outer object cannot interfere.

Das classes à composição de objectos

- **Delegation**
  - Although object composition is very useful, one may question whether it is as powerful for reuse as inheritance.
  - Specifically, using inheritance, an inherited method can always refer to the receiving object using the `this` member variable in C++ or C#.
  - In delegation, the receiving object delegates operations to a delegate (one of the objects it is composed of).
  - The same effect (as this) is achieved by having the receiving object pass a reference to itself to the delegate.
  - Using delegation, a method can always refer to the original recipient of the message, regardless of the number of indirections due to object composition.
  - Delegation allows object composition to be as powerful for reuse as inheritance. Several design patterns make use of this strategy.

Das classes à composição de objectos

```
class Fruit {
    Object outer = null;

    Fruit(Object o) {
        outer = o;
    }
    public int peel() {
        outer.method();
        System.out.println("Peeling is appealing.");
        return 1;
    }
}

class Apple {
    private Fruit fruit = new Fruit(this);

    public int peel() {
        return fruit.peel();
    }
    public void method() { ... }
}
```
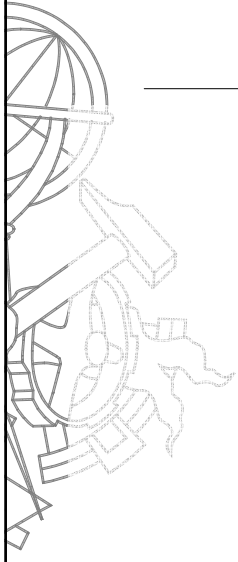
Re-encaminhamento x Delegação

Resumo

- Forwarding
  - Regular Message
- Delegation
  - Self-recursive one
  - Strengthened
  - Identity is remembered
- What the difference between Forwarding and Delegation?

Questões

?

ADAV
Ambientes de Desenvolvimento Avançados