

# Ambientes de Desenvolvimento Avançados

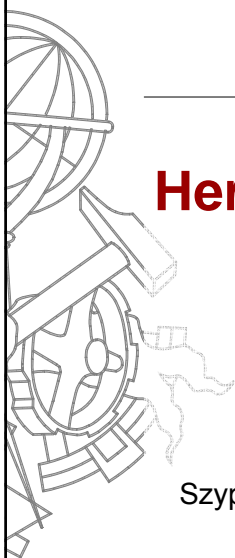
<http://www.dei.isep.ipp.pt/~jtavares/ADAV/ADAV.htm>

## Aula 12 Engenharia Informática

2006/2007

José António Tavares  
jrt@isep.ipp.pt

1




## Herança versus Composição

**Capítulo 7** de:  
Szyperski, Clemens et al. *Component Software - Beyond  
Object-Oriented Programming*. Second Edition

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

2




## Conteúdo

---

- Visão geral
- Vários aspectos da Herança
- Problemas
  - Problemas da classe base frágil
- Abordagens para disciplinar a herança
- Das classes à composição de objectos
  - Reencaminhamento x Delegação

2006/2007 ADAV 3  
Ambientes de Desenvolvimento Avançados




## 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 ?

2006/2007 ADAV 4  
Ambientes de Desenvolvimento Avançados




## Enquadramento

---

“Also, despite the provocative chapter title, there is no intention of banning implementation inheritance outright. Rather, it seems appropriate to analyze carefully what implementation inheritance gives, what it costs, and where the tradeoffs are. **The deeper implications of the implementation inheritance on components rather than objects need to be worked out clearly.**”

2006/2007 ADAV 5  
Ambientes de Desenvolvimento Avançados



## Herança

---

- Simula 67 - 1970
  - Inheritance of implementation
  - Inheritance of interfaces
  - Establishment of substitutability
- Smalltalk - 1983
  - Inheritance of implementation
  - Inheritance of interfaces

2006/2007 ADAV 6  
Ambientes de Desenvolvimento Avançados

# Herança

- Eiffel
  - Possible to undefined inheritance interface feature
- Emerald (1987), Java, C#
  - Interface and implementation inheritance have been separated
- COM and OMG IDL
  - Interface definition language

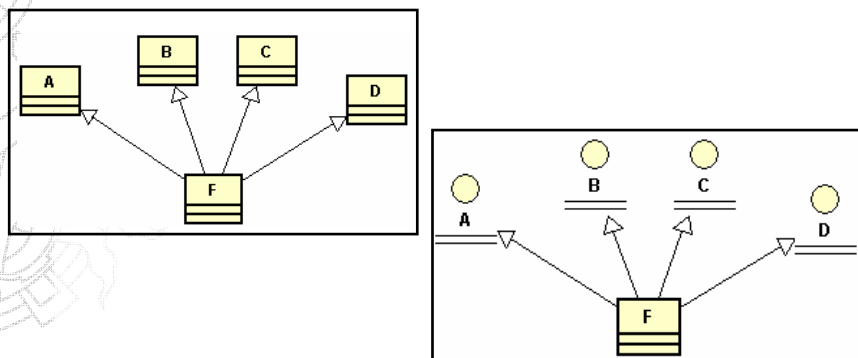
2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

7

# More flavors to the soup

- Multiple Inheritance



2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

8

## More flavors to the soup

- Multiple Inheritance
  - Establish compatibility with multiple independent context is important. Multiple interface is one way to achieve this.
  - OMG IDL, Java, C# → support multiple interface inheritance
  - COM → not support multiple interface inheritance, but permit that a component support multiple interface simultaneous (that is much the same thing).
  - Multiple interface inheritance does not introduce any major technical problems beyond those already introduced by single interface inheritance.

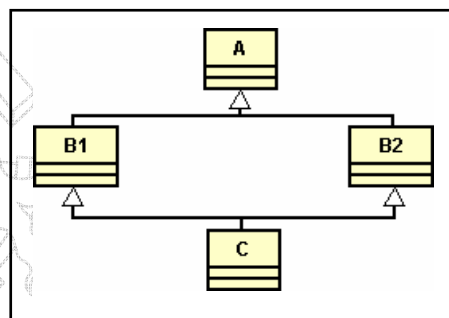
2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

9

## More flavors to the soup

- Mixing implementation fragments...



Do both superclasses B1 and B2 get their own copy of the state defined by the superclass A?

Diamond inheritance problem

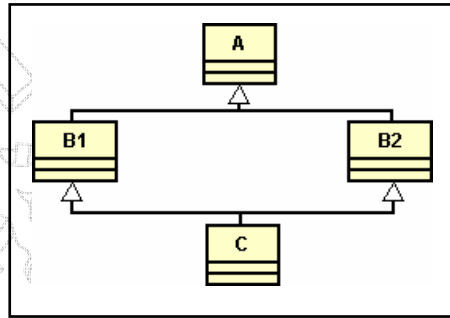
2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

10

## More flavors to the soup

- Mixing implementation fragments...



### About C class ?

- State...
- Methods ...

Diamond inheritance problem

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

11

## More flavors to the soup

- Some approaches to discipline...
  - CLOS (Common Lisp Object System)
    - Linear order of inheritance
  - C++
    - Maintaining the integrity of sub-objects
  - Java
    - Limited to single implementation inheritance
  - OMG IDL and COM
    - Not support implementation inheritance at all

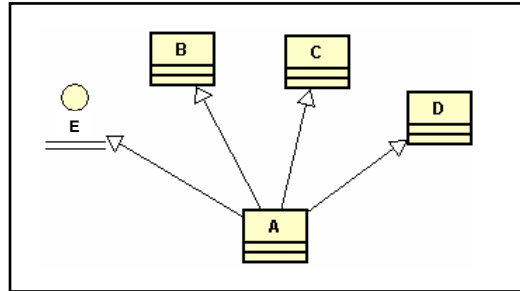
2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

12

## More flavors to the soup

- Mixins



The idea is that a class inherits interfaces from one superclass and implementations from several superclasses, each focusing on distinct parts of the inherited interface.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

13

## More flavors to the soup

- Mixins

```
Interface B
{
    void X ();
    void Y ();
}
```

```
abstract class X1 implements B
{
    void X () {
        ... // X2.Y ();
    }
}
```

```
abstract class X2 implements B
{
    void Y () {
        ...
    }
}
```

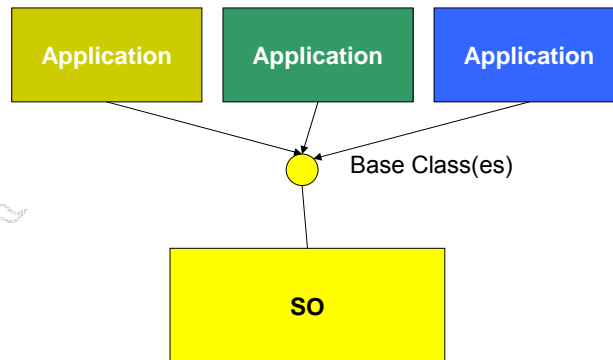
2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

14

## Back to basic ingredients...

- The **Fragile Base Class** (FBC) problem



2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

15

## 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

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

16





## Problema da classe base frágil

---

- The problem is that the 'contract' between components in an implementation hierarchy is not clearly defined. When the parent or child component changes its behavior unexpectedly, the behavior of the related components may become undefined.
- By completely encapsulating the implementation of an object, SOM overcomes what Microsoft refers to as the 'fragile base class problem', i.e., the inability to modify a class without recompiling clients and derived classes dependent upon that class.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

17



## Problema da classe base frágil

---

### Syntactic

- a matter of binary compatibility of compiled sub-classes with new binary releases of super-classes
- 'release-to-release binary compatibility'
- nothing to do with semantics of inherited code
- sub-class should not need recompilation, just because of 'syntactic' changes to super-class's interface
- e.g. moving methods up the class hierarchy
- IBM's SOM solves this problem by initializing method dispatch tables at load time
- cannot address all 'syntactic' changes, e.g. splitting a method in two, or joining two methods into one, or changing a parameter list

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

18

## 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

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

19

## Problema da classe base frágil

### Semantic

- how can subclasses remain valid in the face of evolution of the *implementation* of super-classes?
- syntactic FBC addresses problems with immature libraries, but evolution of mature libraries more likely to raise semantic FBC
- to answer this question, it is necessary to understand the semantics of implementation inheritance
- subject of the remainder of this section

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

20

## Up-calls via Herança

- implementation inheritance usually combined with overriding
- selected inherited methods are overridden with new implementations
- new implementations may call overridden code at arbitrary point; abstract methods, or methods of interfaces, may have implementations provided
- invocation of overridden method similar to up-call (method in super-class calling implementation in a sub-class)
- calls span sub-class and super-class in both directions
- but: *every method* is now potentially a callback
- similar problems arise (practical!)
- how to control complexity?

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

21

## Inheritance – more knots than meet the eye

```
abstract class Text
{
    ...
    void write (pos, ch)
    {
        ...
        setCaret(pos);
    }
    void setCaret (int pos)
    {
        caret = pos;
    }
    ...
}
```

```
class SimpleText extends Text
{
    ...
    void setCaret (int pos)
    {
        int old = caretPos();
        if (old != pos)
        {
            hideCaret();
            super.setCaret(pos);
            showCaret();
        }
    }
    ...
}
```

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

22

## Inheritance – more knots than meet the eye

```
abstract class Text
{
    ...
    void write (pos, ch)
    {
        ...
        pos++;
    }
    void setCaret (int pos)
    {
        caret = pos;
    }
    ...
}
```

```
class SimpleText extends Text
{
    ...
    void showCaret (int pos)
    {
        int oldCaretPos = caretPos();
        if (oldCaretPos != pos)
        {
            hideCaret();
            super.setCaret(pos);
            showCaret();
        }
    }
    ...
}
```

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

23

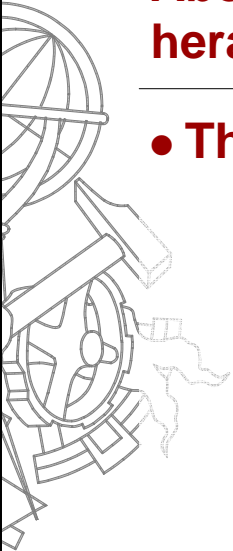
## Abordagens para disciplinar a herança

- these problems known for a while:  
*inheritance breaks encapsulation* (Snyder, 1986)
- early attempts at solution addressed language weaknesses
- but still, sub-class can interfere with and break super-class implementation
- likewise, evolution of super-class can break sub-classes
- some attempts to control use of implementation inheritance:
  - specialization interface
  - partitioning objects
  - reuse contracts (covered in book, not here)

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

24

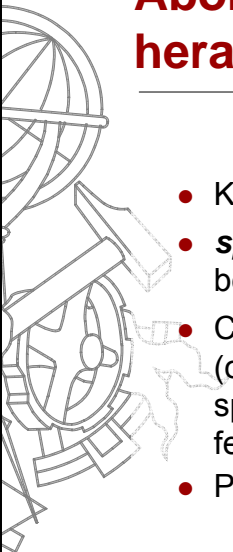


## Abordagens para disciplinar a herança

---

- **The specialization interface**

2006/2007 ADAV 25  
Ambientes de Desenvolvimento Avançados



## Abordagens para disciplinar a herança

---

### The specialization interface

- Kiczales and Lamping, 1992
- **specialization interface** is the special interface between class and sub-class
- C++, Java and C#, for example, client interface (outside package) includes only *public* features; specialization interface includes also *protected* features
- Protected - Accessible only to sub-classes

2006/2007 ADAV 26  
Ambientes de Desenvolvimento Avançados



## Abordagens para disciplinar a herança

### The specialization interface (*cont*)

- private features can be used to solve problems pointed by Snyder
- In C++, Java and C#, a private feature is private to a class, not an object
- Java, C# and Component Pascal also support the important notion of package-private (or internal) interfaces.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

27



## Abordagens para disciplinar a herança

- **Typing the specialization interface**

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

28

## Abordagens para disciplinar a herança

### Typing the specialization interface

- What are the legal modifications a sub-class can apply?
- Protected interface
- 1993, John Lamping
- Statically
  - Acyclic - Arranged in layers
  - Cyclic - Form a group
- The idea is declare statically which other methods of the same class a given method might **DEPEND** on.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

29

## Abordagens para disciplinar a herança

### Typing the specialization interface (cont)

- Where dependencies form acyclic graphs, methods can be arranged in layers;
- Where dependencies form cycles, all the methods in the cycle together form a group;
- If a method need to call another method, it either has to be a member of the called method's group or of a higher layer's group;
- In such an approach, a sub-class has to override methods group by group – either all methods of a group are overridden or none.
- Grouping and layering of methods is seen as a design activity.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

30

## Abordagens para disciplinar a herança

### Typing the specialization interface (cont)

- The developer determines the **groups** or **layers**

```
specialization interface Text {  
  state caretRep  
  state textRep  
  abstract posToXCoord  
  abstract posToYCoord  
  concrete caretPos {caretPos}  
  concrete setCaret {caretRep}  
  concrete write {textRep, caretPos, setCaret}  
  concrete delete {textRep, caretPos, setCaret}  
  ...  
}
```

No dependencies

- Today no language directly supports Lamping's specialization interface typing

## Abordagens para disciplinar a herança

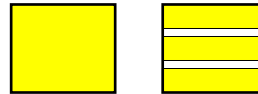
- **Behavioral specification of the specialization interface**



## Abordagens para disciplinar a herança

### Behavioral specification of the specialization interface

- Lamping's proposal improves information available to sub-classes, but does not address semantic issues of inheritance
- *behavioral* aspects of inheritance (Stata and Guttag, 1995)
- 1995, Stata & Guttag
  - Class as a combined definition of interacting parts objects
    - Method groups
    - Algebraic specification techniques
    - Notion of behavioral sub-typing



2006/2007

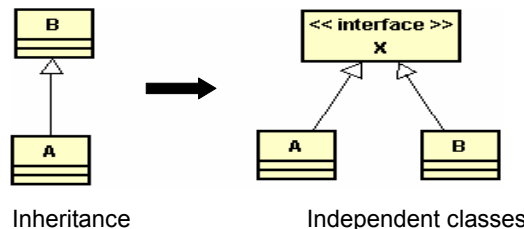
ADAV  
Ambientes de Desenvolvimento Avançados

33

## Abordagens para disciplinar a herança

### Behavioral specification of the specialization interface (cont)

- to transform ordinary object to Stata-Guttag object group: use only a single sub-object
- Sub-class may change nothing or everything; implementation inheritance useless
- might as well share interface, provide new implementation



2006/2007

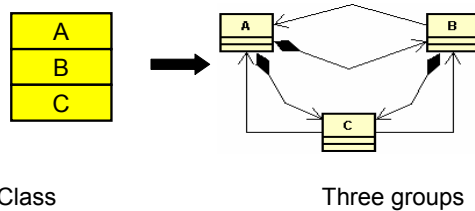
ADAV  
Ambientes de Desenvolvimento Avançados

34

## Abordagens para disciplinar a herança

### Behavioral specification of the specialization interface (*cont*)

- conversely, transform object group into collection of objects
- 'self' is lost; how to refer to peers?
- provide each sub-object with references to the others
- to handle object identity, nominate one sub-object the 'main part'



2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

35

## Abordagens para disciplinar a herança

- Reuse and cooperation contracts

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

36

# Abordagens para disciplinar a herança

## Reuse and cooperation contracts

- 1996, Steyaert, et. al.
- Returned to the idea of statically verifiable annotations
- Reuse contract

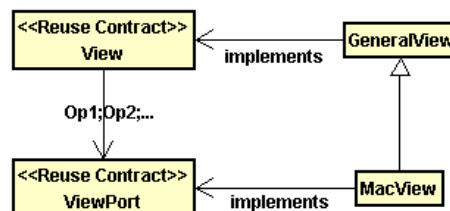
```
reuse contract Text {
  abstract
  posToXCoord
  posToYCoord
  concrete
  caretPos
  setCaret
  write {caretPos, setCaret}
  delete {caretPos, setCaret}
  ...
}
```

Only among methods

# Abordagens para disciplinar a herança

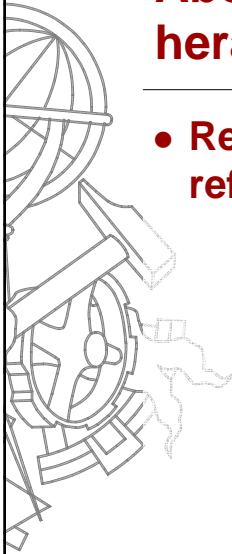
## Reuse and cooperation contracts (cont)

- Real innovation - Set of modification operators
  - **Concretization** - replace abstract methods by concrete methods (its inverse is abstraction)
  - **Extension** - add new method that depend on new or existing methods
  - **Refinement** - override methods, introducing new dependencies to possibly new methods.



## Abordagens para disciplinar a herança

- Representing invariants and method refinements



2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

39

## Abordagens para disciplinar a herança

### Representation invariants and methods refinements

- 1996, Edwards
  - Generalization of the Stata & Guttag
  - Overriding a method in a method group
  - Associating invariants with a class



- Protected
- Public
- Private
- Etc.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

40



## Abordagens para disciplinar a herança

---

### Representation invariants and methods refinements (*cont*)

- Demonstrate that the overriding of individual methods in a method is permissible if the subclass maintains the representation invariant of the group's variables.
- The idea is to explicitly associate invariants with a class specification that refers to protected variables, which are variables that are only accessible by class and sub-class code (but not external client code)



## Abordagens para disciplinar a herança

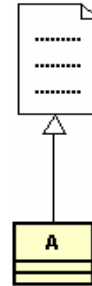
---

- **Disciplined inheritance to avoid FBC problems**

## Abordagens para disciplinar a herança

### Disciplined inheritance to avoid fragile base class problems

- 1998, Mikhajlov & Sekerinski  
Construir a sub-classe baseada na especificação da super classe, assim a sub-classe ainda será valida mesmo que a implementação da super-classe mude.



2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

43

## Abordagens para disciplinar a herança

- **Creating correct sub-classes without seeing the super-class code**

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

44

## Abordagens para disciplinar a herança

### Creating correct subclasses without seeing superclass code

- 2000, Ruby & Leavens
- Inverse problem of the semantic FBC problem



2006/2007

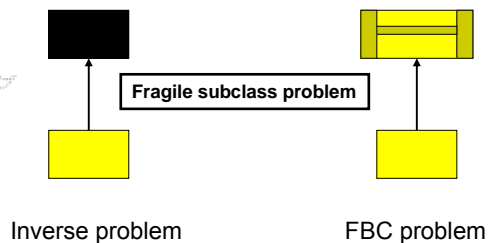
ADAV  
Ambientes de Desenvolvimento Avançados

45

## Abordagens para disciplinar a herança

### Creating correct subclasses without seeing superclass code (*cont*)

- 2000, Ruby & Leavens
- Inverse problem of the semantic FBC problem



2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

46



## Abordagens para disciplinar a herança

---

### Creating correct subclasses without seeing superclass code (*cont*)

- Provide 3 parts to a class specification - for the sub-class can be safely created without requiring access to the source code of the base class
- Public
- Protected – reveals information such invariants over protected variables and conditions on protected methods
- Automatic analysis of the initial source code of the base class – provides information on which variables are accessed and which methods are called by any given method.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

47

- 
- 
- **Das classes à composição de objectos**

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

48





## Das classes à composição de objectos

---

- Kiczales and Lamping, 1992
- *specialization interface* is the special interface between class and subclass
- eg in Java, client interface (outside package) includes only *public* features; specialization interface includes also *protected* features
- restricts access to interfaces, but doesn't restrict usage by those with access
- distinction between client and descendent interfaces important for controlling implementation inheritance
- sub-class needs to know something about implementation of class

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

49



## 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

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

50



## 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 not leave that object.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

51



## 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.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

52

## Das classes à composição de objectos

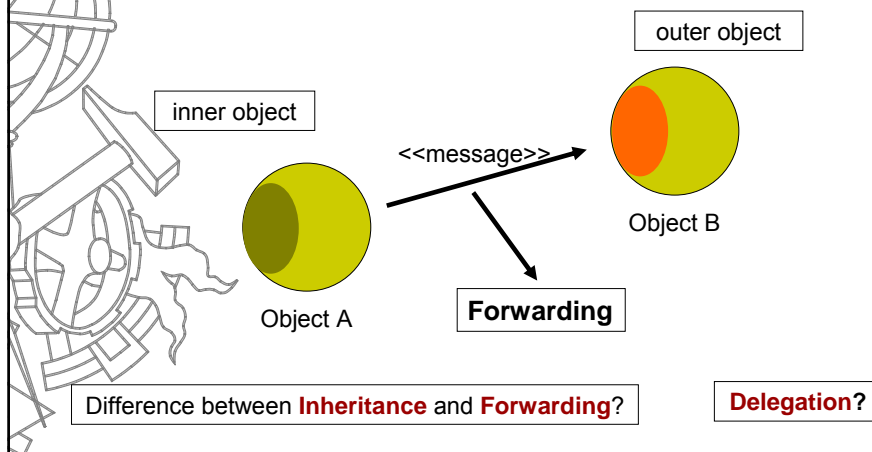
- An **outer object** does not re-implement the functionality of the **inner object** when it forwards messages;
- It reuses the implementation of the inner object;
- If the implementation of the inner object is changed, then this change will “spread” to the outer object;
- The difference between **object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

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

53

## Das classes à composição de objectos



2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

54



## 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.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

55



## Das classes à composição de objectos

---

### Delegation

- **Composition + forwarding** lacks the notion of a common “self”;
- If a common identity is required, it has to be designed in;
- If an object was not designed for composition under a common identity, it cannot be used in such context – mechanisms build in to resend messages to an outer object;
- Object composition supports dynamic and late composition.

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

56

## Das classes à composição de objectos

### Delegation (cont)

- The concept of message passing by delegation is relatively simple;
- Each message-send is classified either as regular send (forwarding) or self-recursive one (delegation)
- Whenever a message is delegated (instead of forwarded), the identity of the first delegator in the current message is remembered;
- Any subsequently delegated message is dispatched back to the original delegator.

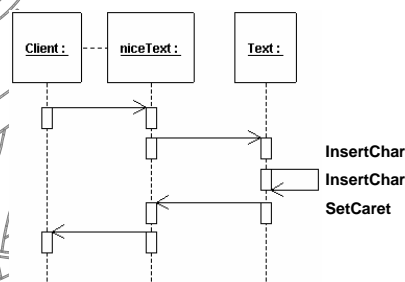
2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

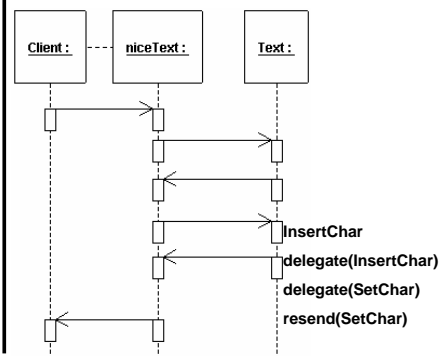
57

## Re-encaminhamento x Delegação

### Forwarding



### Delegation



2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

58

## Re-encaminhamento x Delegação

### Resumo

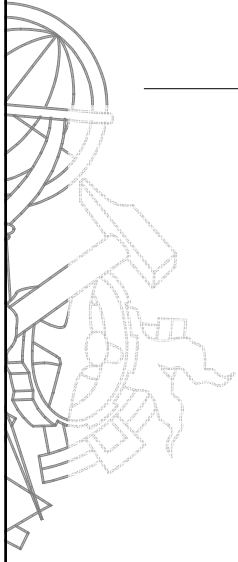
- Forwarding
  - Regular Message
- Delegation
  - Self-recursive one
  - Strengthened
  - Identity is remembered
- What the difference between Forwarding and Delegation?



## Delegação x Herança

Gamma et al. (1995)

“Delegation has a disadvantage that it shares with other techniques that make software more flexible through object composition: dynamic, highly parameterized software is harder to understand than more static software. [...] Delegation is a good design choice only when it simplifies more than it complicates. [...] Delegation works best when it is used in highly stylized ways – that is, in standard patterns.”



---

**Questões**

**?**

2006/2007

ADAV  
Ambientes de Desenvolvimento Avançados

61