Ambientes de Desenvolvimento Avançados

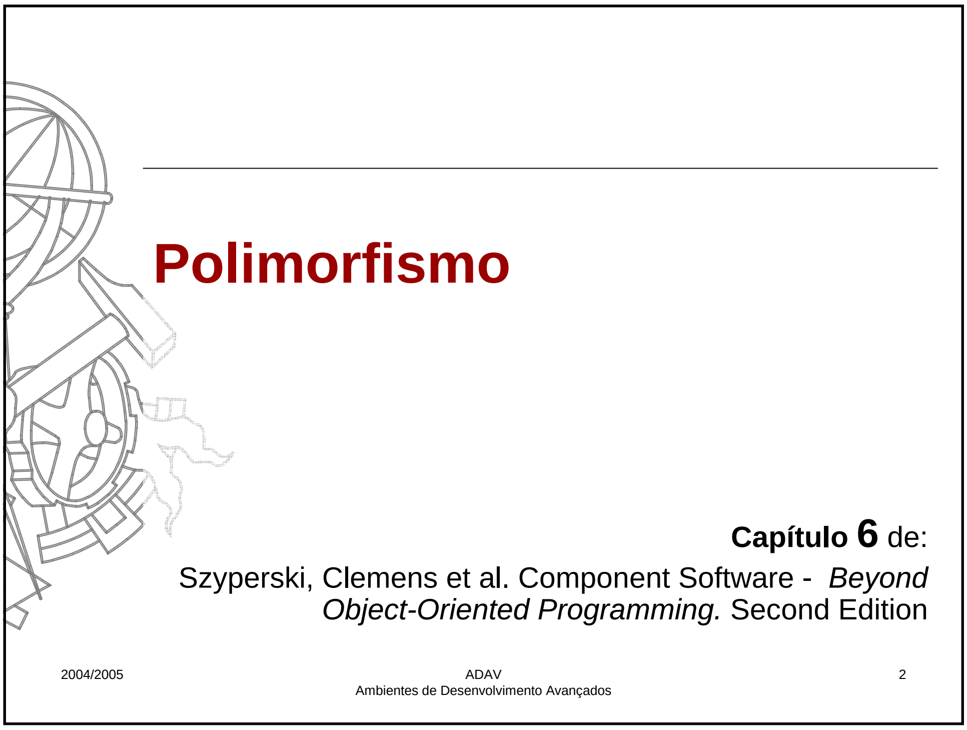
http://www.dei.isep.ipp.pt/~jtavares/ADAV/ADAV.htm

Aula 9

Engenharia Informática

2004/2005

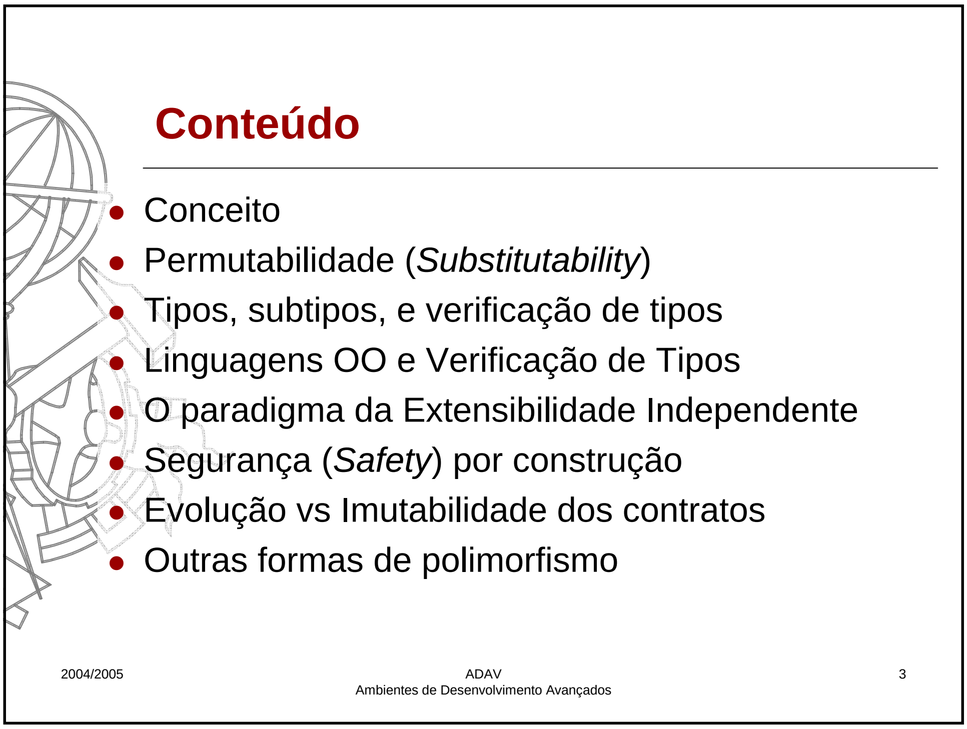
José António Tavares
jrt@isep.ipp.pt



Polimorfismo

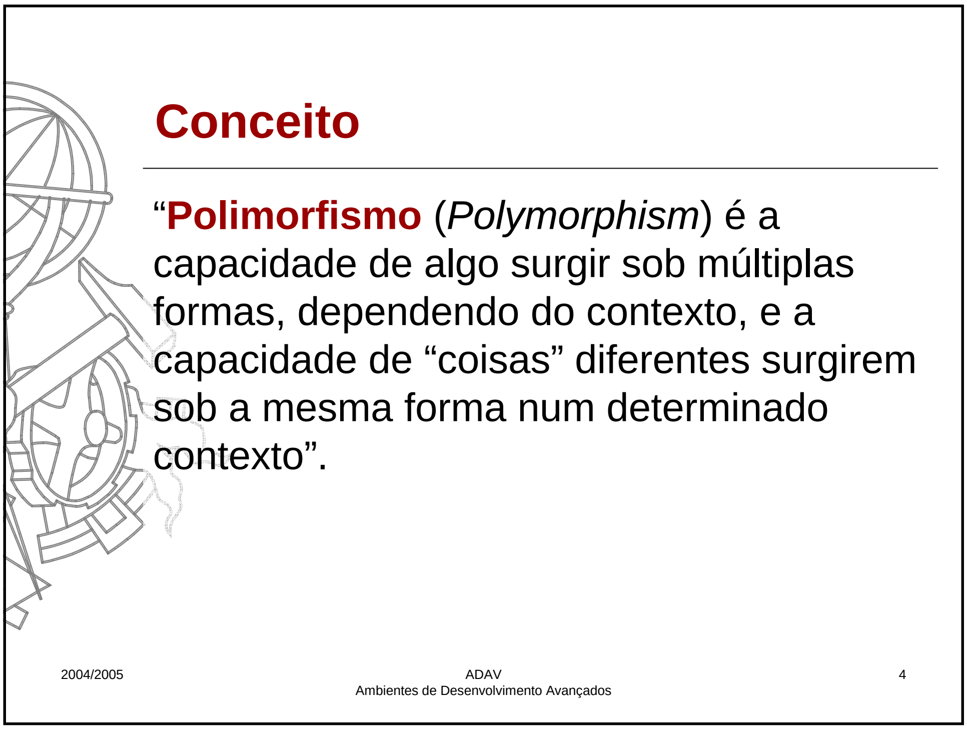
Capítulo 6 de:

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



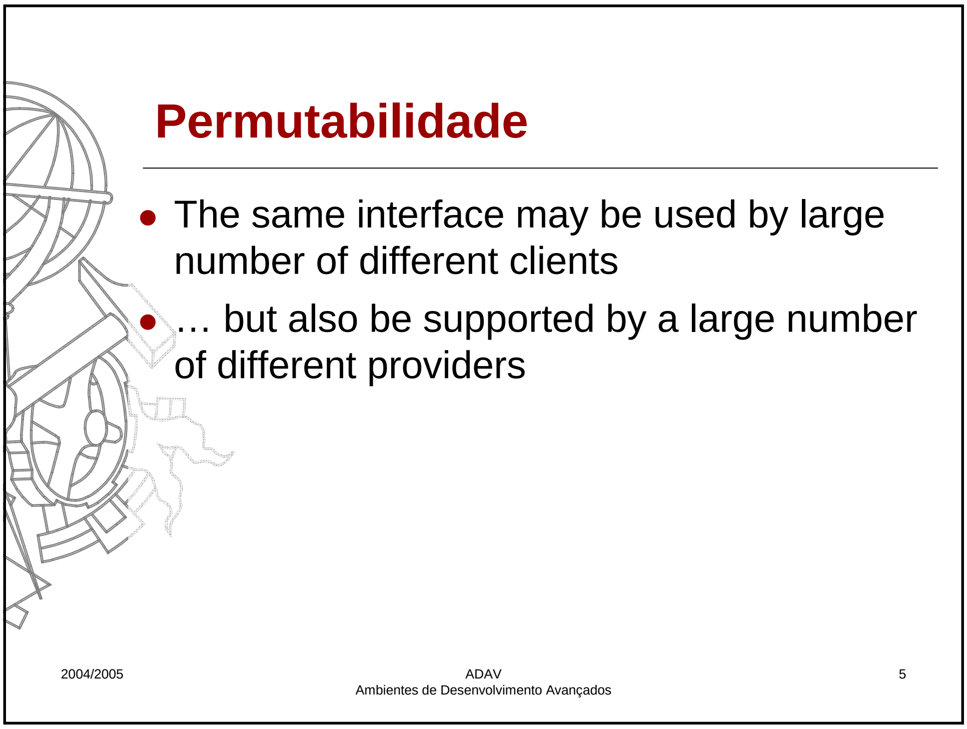
Conteúdo

- Conceito
- Permutabilidade (*Substitutability*)
- Tipos, subtipos, e verificação de tipos
- Linguagens OO e Verificação de Tipos
- O paradigma da Extensibilidade Independente
- Segurança (*Safety*) por construção
- Evolução vs Imutabilidade dos contratos
- Outras formas de polimorfismo



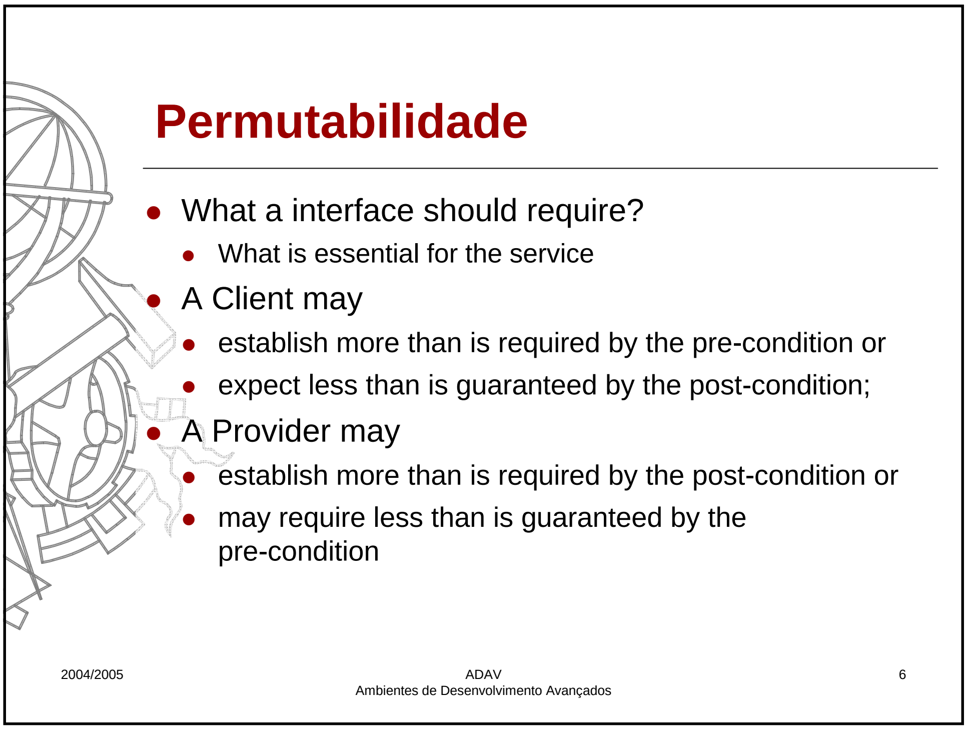
Conceito

“**Polimorfismo** (*Polymorphism*) é a capacidade de algo surgir sob múltiplas formas, dependendo do contexto, e a capacidade de “coisas” diferentes surgirem sob a mesma forma num determinado contexto”.



Permutabilidade

- The same interface may be used by large number of different clients
- … but also be supported by a large number of different providers

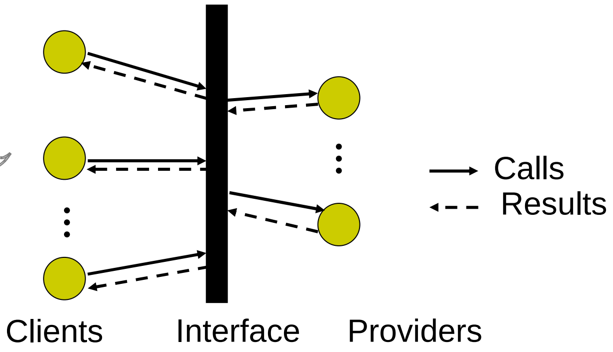


Permutabilidade

- What a interface should require?
  - What is essential for the service
- A Client may
  - establish more than is required by the pre-condition or
  - expect less than is guaranteed by the post-condition;
- A Provider may
  - establish more than is required by the post-condition or
  - may require less than is guaranteed by the pre-condition

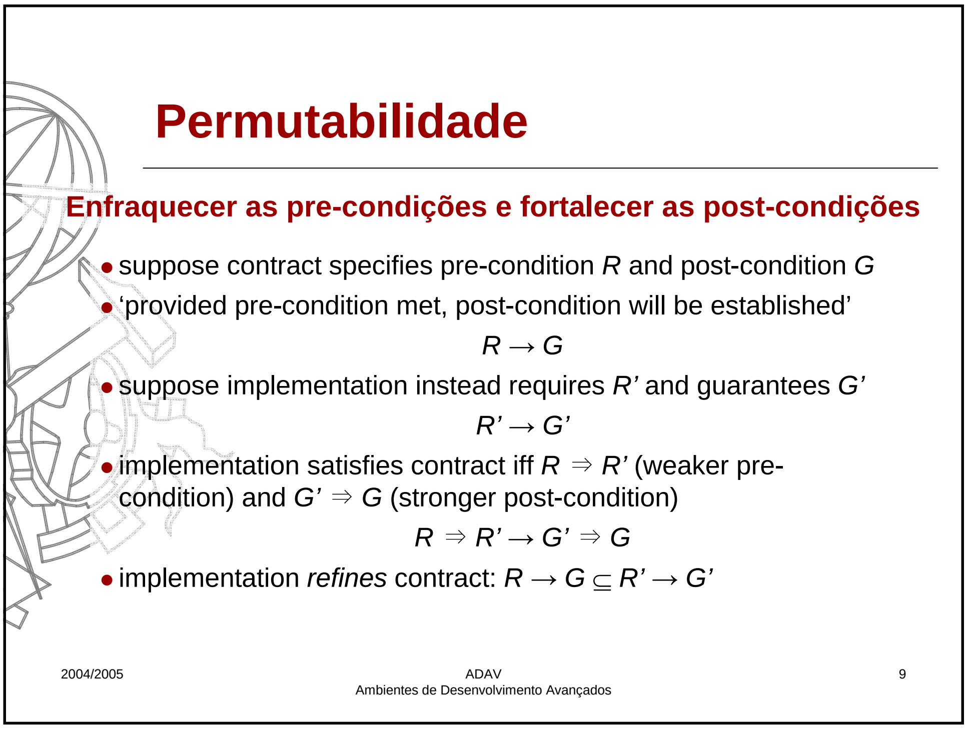
Permutabilidade

Pivotal role of the interface contract when considering multiple clients and multiple providers of the services advertised by an interface.



Permutabilidade

- When is it legal to substitute one service provider for another?
- An unknown number of clients may rely on the service simply by relying on what is contractually promised by the service interface;
- Therefore, another service provider can come in if it satisfies the same contract;
- If a provider satisfies the same contract as another, the former is said to be **substitutable** for the latter.



Permutabilidade

Enfraquecer as pre-condições e fortalecer as post-condições

- suppose contract specifies pre-condition R and post-condition G
- ‘provided pre-condition met, post-condition will be established’

$$R \rightarrow G$$

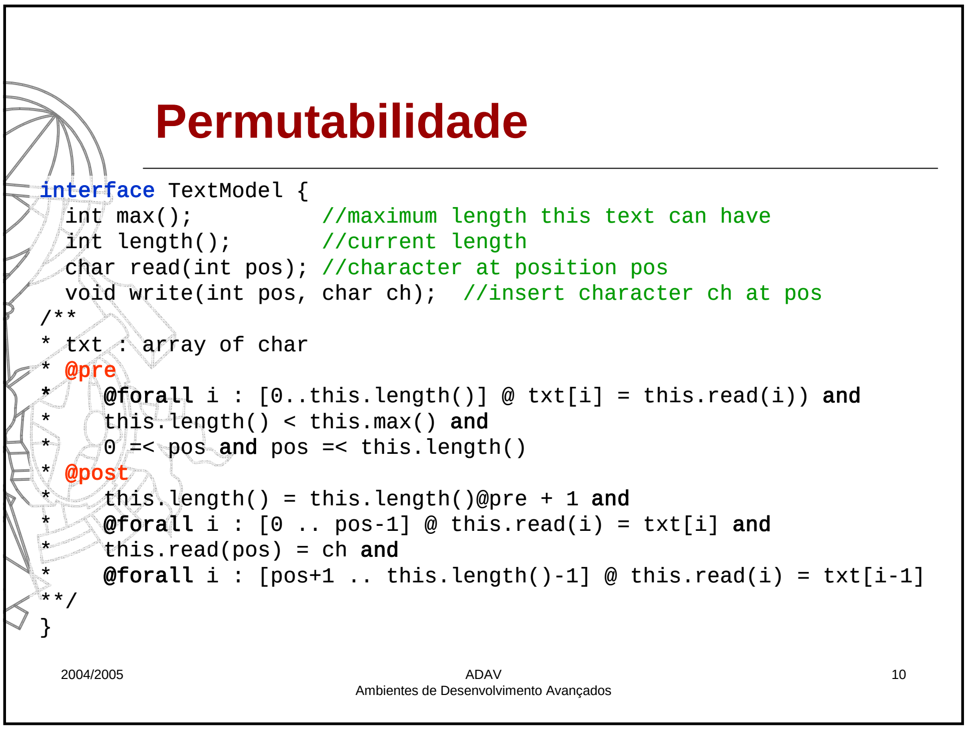
- suppose implementation instead requires R' and guarantees G'

$$R' \rightarrow G'$$

- implementation satisfies contract iff $R \Rightarrow R'$ (weaker pre-condition) and $G' \Rightarrow G$ (stronger post-condition)

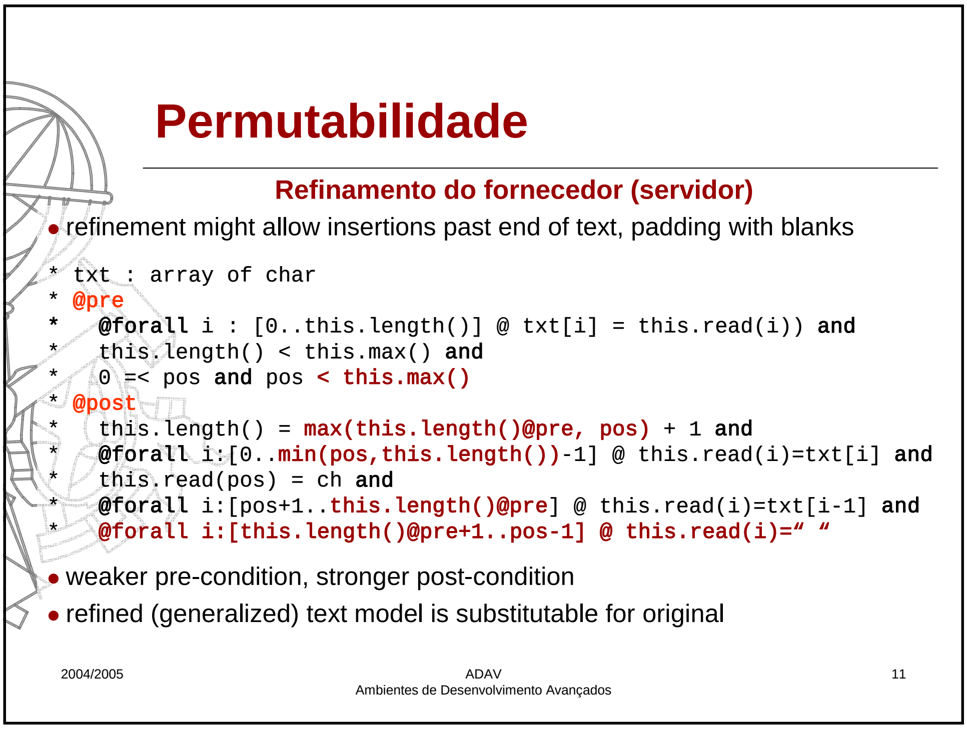
$$R \Rightarrow R' \rightarrow G' \Rightarrow G$$

- implementation *refines* contract: $R \rightarrow G \subseteq R' \rightarrow G'$



Permutabilidade

```
interface TextModel {
  int max();          //maximum length this text can have
  int length();       //current length
  char read(int pos); //character at position pos
  void write(int pos, char ch);  //insert character ch at pos
/**
* txt : array of char
* @pre
*    @forall i : [0..this.length()] @ txt[i] = this.read(i)) and
*    this.length() < this.max() and
*    0 =< pos and pos =< this.length()
* @post
*    this.length() = this.length()@pre + 1 and
*    @forall i : [0 .. pos-1] @ this.read(i) = txt[i] and
*    this.read(pos) = ch and
*    @forall i : [pos+1 .. this.length()-1] @ this.read(i) = txt[i-1]
**/
}
```



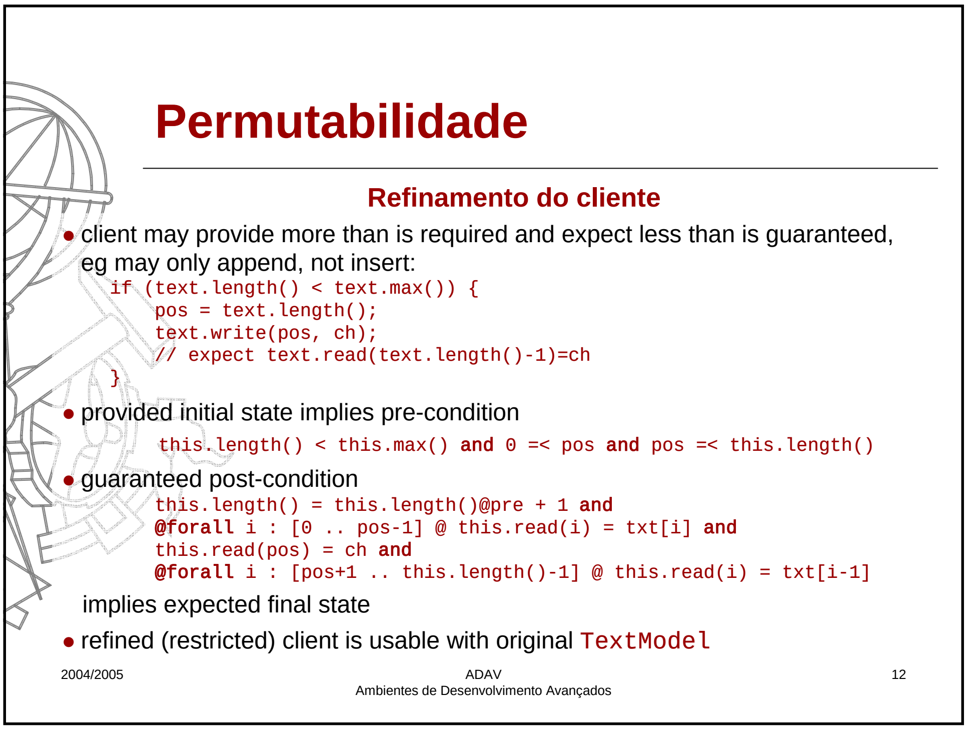
Permutabilidade

Refinamento do fornecedor (servidor)

- refinement might allow insertions past end of text, padding with blanks

```
* txt : array of char
* @pre
*   @forall i : [0..this.length()] @ txt[i] = this.read(i)) and
*   this.length() < this.max() and
*   0 =< pos and pos < this.max()
* @post
*   this.length() = max(this.length()@pre, pos) + 1 and
*   @forall i:[0..min(pos,this.length())-1] @ this.read(i)=txt[i] and
*   this.read(pos) = ch and
*   @forall i:[pos+1..this.length()@pre] @ this.read(i)=txt[i-1] and
*   @forall i:[this.length()@pre+1..pos-1] @ this.read(i)=“ “
```

- weaker pre-condition, stronger post-condition
- refined (generalized) text model is substitutable for original



Permutabilidade

Refinamento do cliente

- client may provide more than is required and expect less than is guaranteed, eg may only append, not insert:

```
  if (text.length() < text.max()) {
      pos = text.length();
      text.write(pos, ch);
      // expect text.read(text.length()-1)=ch
  }
```

- provided initial state implies pre-condition

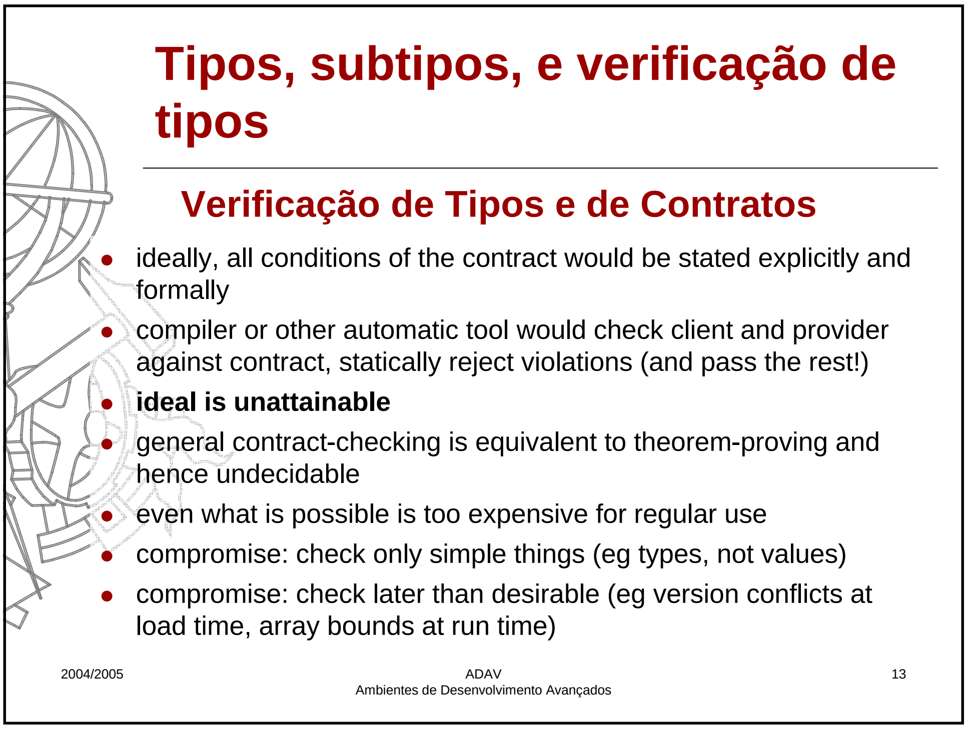
```
      this.length() < this.max() and 0 =< pos and pos =< this.length()
```

- guaranteed post-condition

```
      this.length() = this.length()@pre + 1 and
      @forall i : [0 .. pos-1] @ this.read(i) = txt[i] and
      this.read(pos) = ch and
      @forall i : [pos+1 .. this.length()-1] @ this.read(i) = txt[i-1]
```

implies expected final state

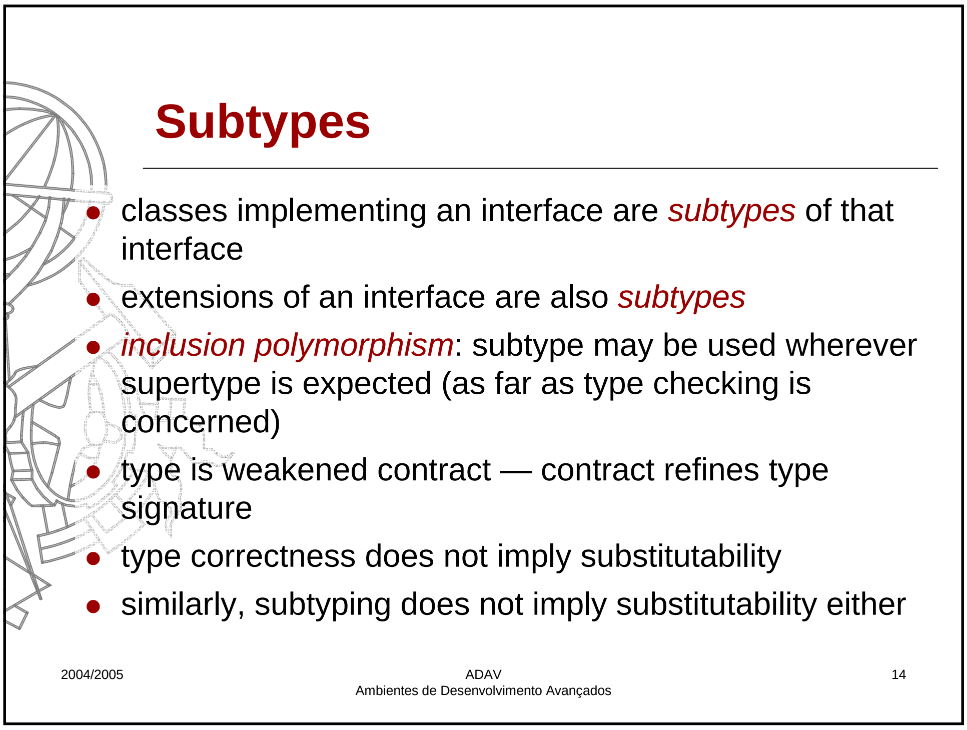
- refined (restricted) client is usable with original `TextModel`



Tipos, subtipos, e verificação de tipos

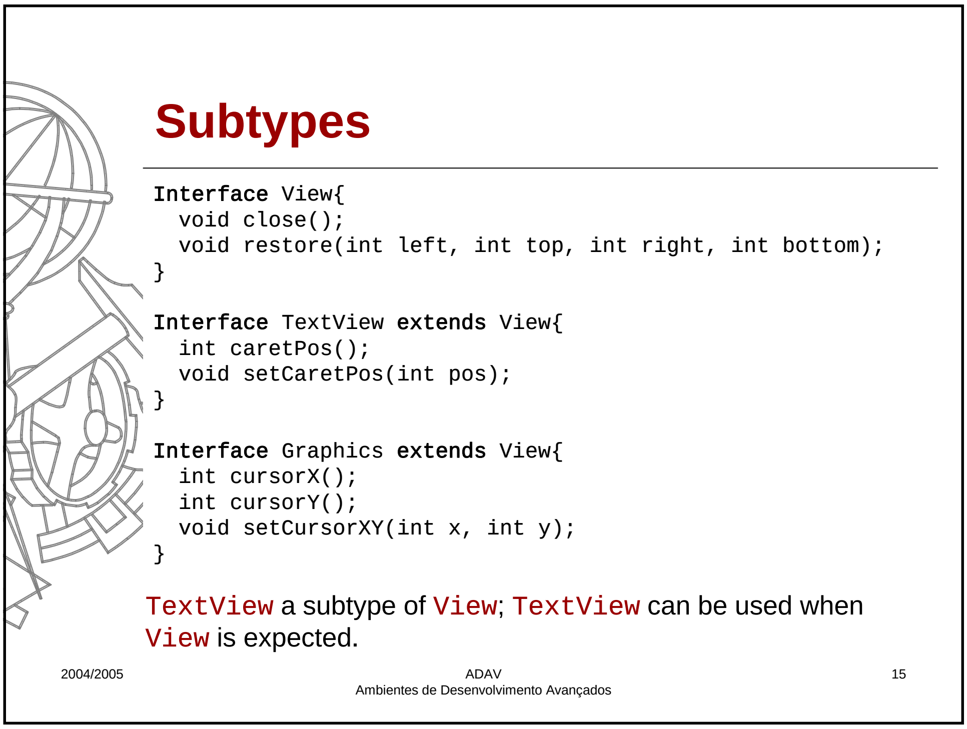
Verificação de Tipos e de Contratos

- ideally, all conditions of the contract would be stated explicitly and formally
- compiler or other automatic tool would check client and provider against contract, statically reject violations (and pass the rest!)
- **ideal is unattainable**
- general contract-checking is equivalent to theorem-proving and hence undecidable
- even what is possible is too expensive for regular use
- compromise: check only simple things (eg types, not values)
- compromise: check later than desirable (eg version conflicts at load time, array bounds at run time)



Subtypes

- classes implementing an interface are *subtypes* of that interface
- extensions of an interface are also *subtypes*
- *inclusion polymorphism*: subtype may be used wherever supertype is expected (as far as type checking is concerned)
- type is weakened contract — contract refines type signature
- type correctness does not imply substitutability
- similarly, subtyping does not imply substitutability either



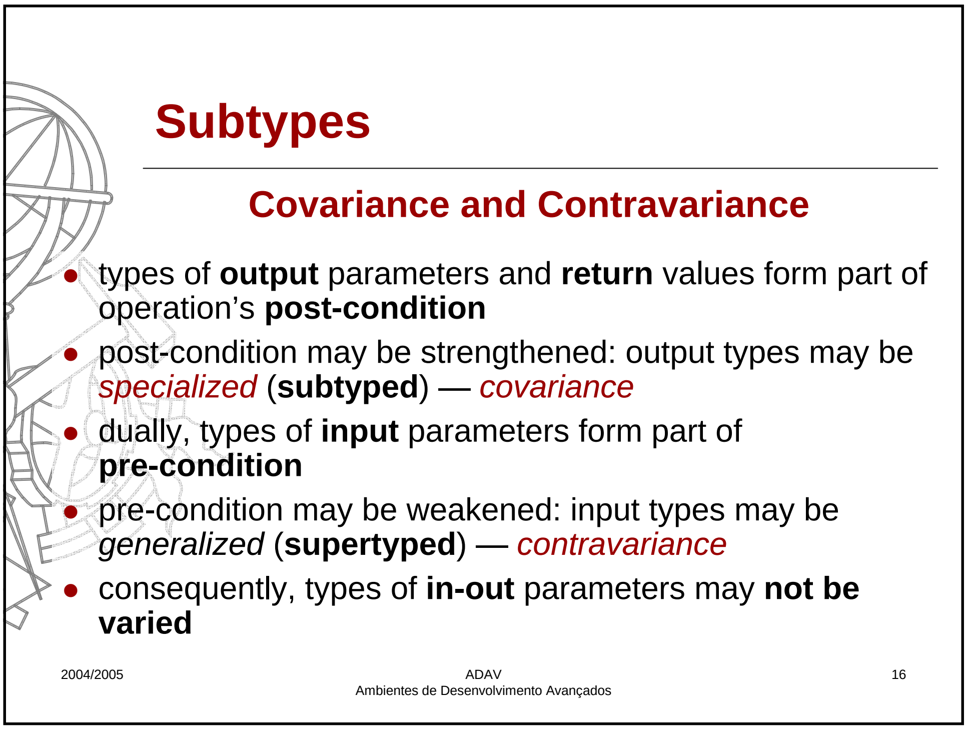
Subtypes

```
Interface View{
  void close();
  void restore(int left, int top, int right, int bottom);
}

Interface TextView extends View{
  int caretPos();
  void setCaretPos(int pos);
}

Interface Graphics extends View{
  int cursorX();
  int cursorY();
  void setCursorXY(int x, int y);
}
```

`TextView` a subtype of `View`; `TextView` can be used when `View` is expected.



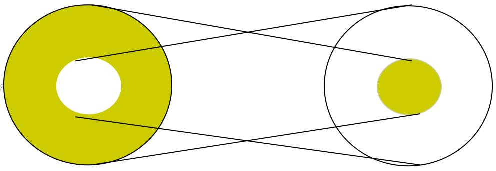
Subtypes

Covariance and Contravariance

- types of **output** parameters and **return** values form part of operation's **post-condition**
- post-condition may be strengthened: output types may be *specialized* (**subtyped**) — *covariance*
- dually, types of **input** parameters form part of **pre-condition**
- pre-condition may be weakened: input types may be *generalized* (**supertyped**) — *contravariance*
- consequently, types of **in-out** parameters may **not be varied**

Subtypes

- Subcontract
- Covariance and Contravariance



Function F

Function G

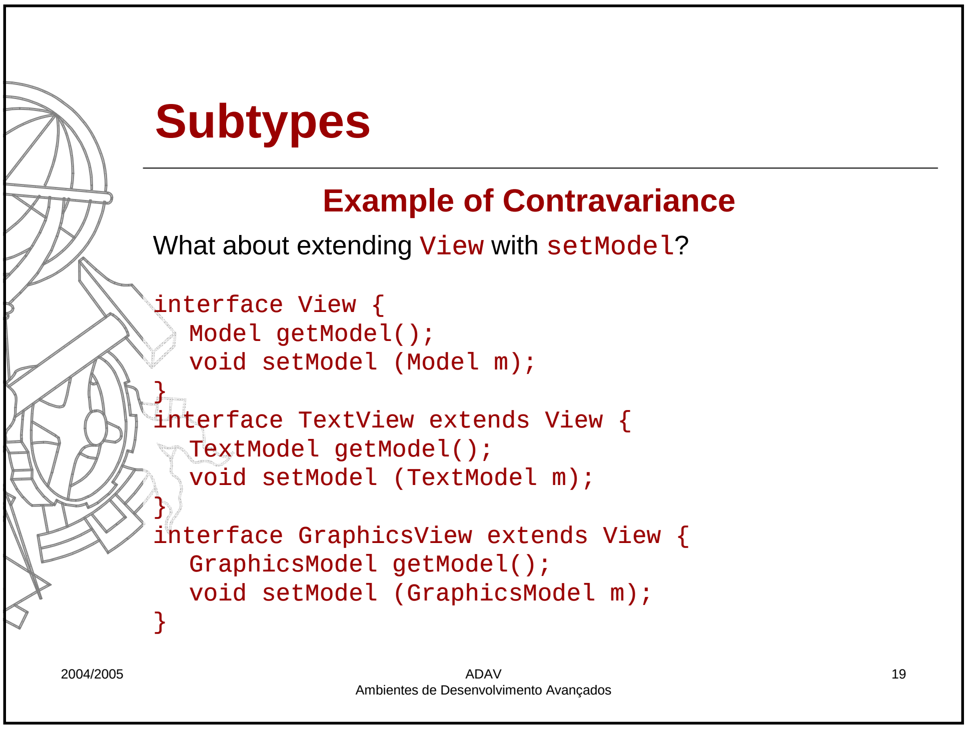
Subtypes

Example of Covariance

Suppose `TextModel` and `GraphicsModel` subtype `Model`.

```
interface View {
   Model getModel();
}
interface TextView extends View {
   TextModel getModel(); // not legal Java!
}
interface GraphicsView extends View {
   GraphicsModel getModel();
}
```

- covariant redefinition in subtypes of output type of `getModel`
- client expecting `Model` may get `TextModel` instead

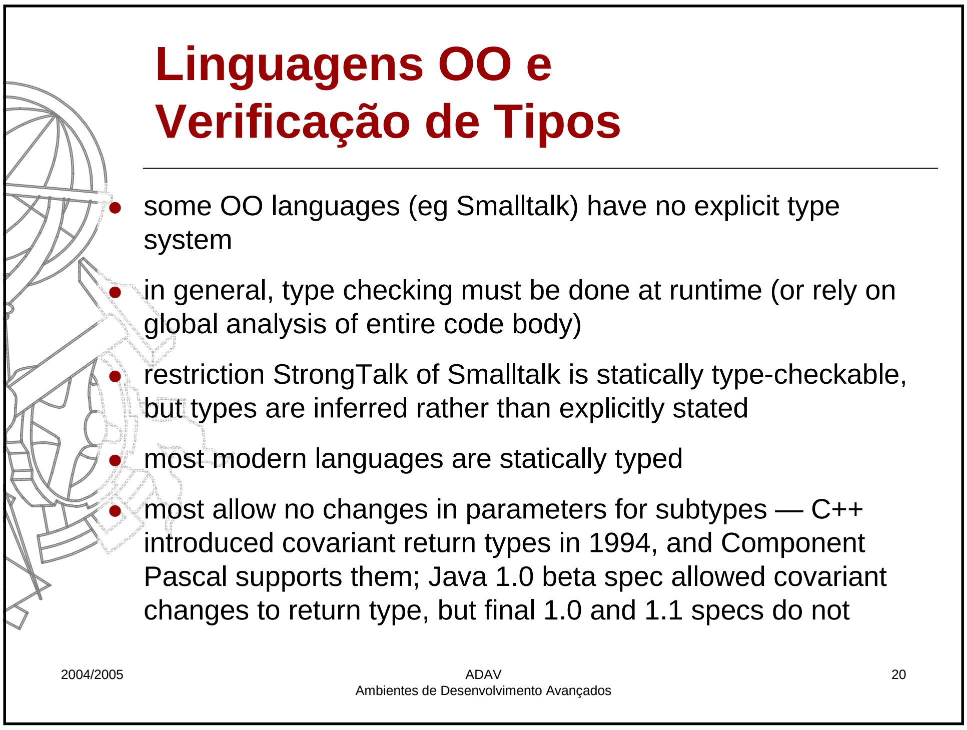


Subtypes

Example of Contravariance

What about extending `View` with `setModel`?

```
interface View {
   Model getModel();
   void setModel (Model m);
}
interface TextView extends View {
   TextModel getModel();
   void setModel (TextModel m);
}
interface GraphicsView extends View {
   GraphicsModel getModel();
   void setModel (GraphicsModel m);
}
```



Linguagens OO e Verificação de Tipos

- some OO languages (eg Smalltalk) have no explicit type system
- in general, type checking must be done at runtime (or rely on global analysis of entire code body)
- restriction StrongTalk of Smalltalk is statically type-checkable, but types are inferred rather than explicitly stated
- most modern languages are statically typed
- most allow no changes in parameters for subtypes — C++ introduced covariant return types in 1994, and Component Pascal supports them; Java 1.0 beta spec allowed covariant changes to return type, but final 1.0 and 1.1 specs do not

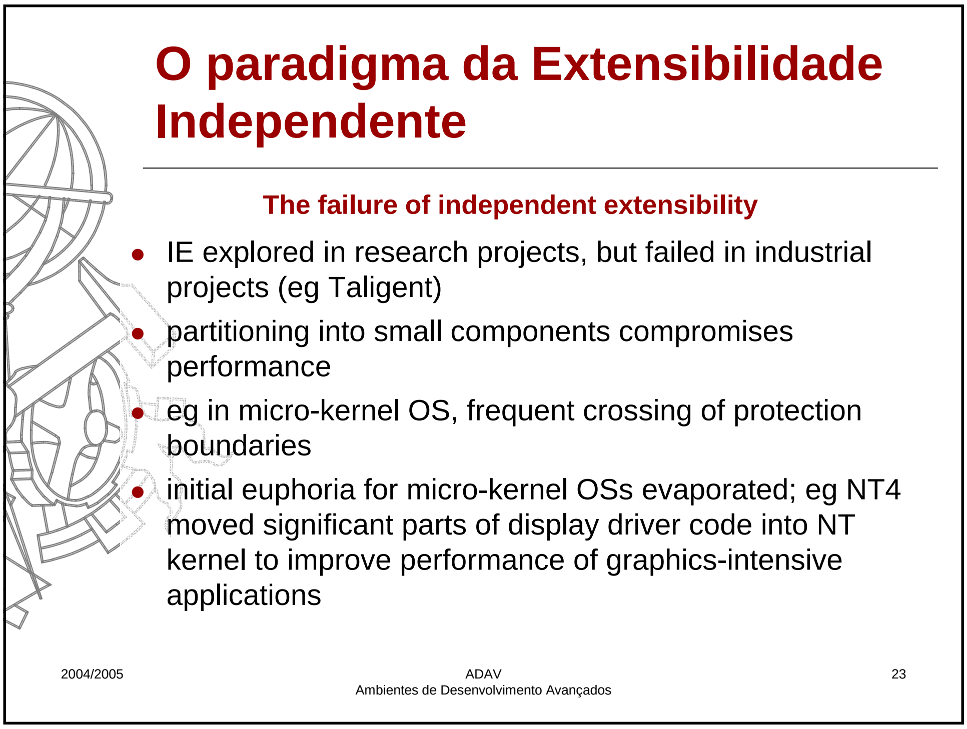
Subtipos estruturais vs subtipos declarados

- some programming languages (eg StrongTalk, Haskell) can *infer* types by analyzing code
- not possible for interface: there may be no code!
- others establish subtyping by examining structure of types: if interface of one type contains all methods of a second, with appropriate signatures, it is a subtype
- known as *structural subtyping* as opposed to *declared subtyping*
- dangerous, because of coincidences
- (graphics editor accepting all objects implementing `draw`. . .)
- accidental subtyping unlikely? not with abstract classes, which have small interfaces (eg `java.lang.Cloneable` has no entries!)

O paradigma da Extensibilidade Independente

(The paradigm of Independent Extensibility - IE)

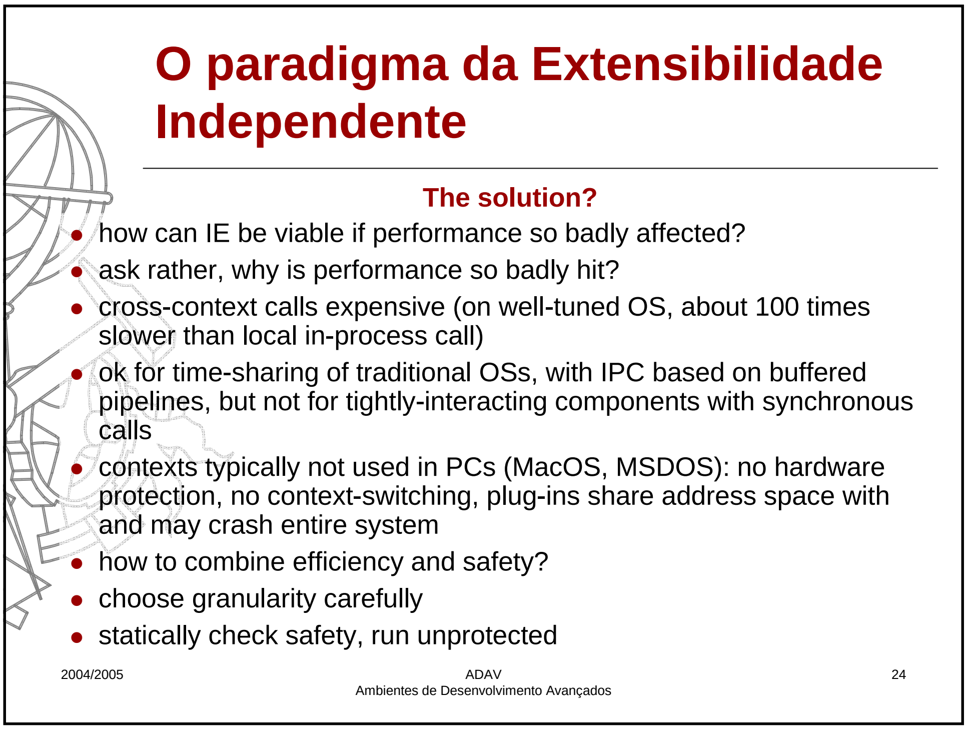
- principle function of component orientation is to allow *independent extensibility*
- independently-developed extensions should be freely combinable
- eg OS with applications
- eg plug-in architectures (Netscape, QuickTime)
- eg micro-kernel OS architectures (influencing NT)
- want uniform independent extensibility recursively through all levels



O paradigma da Extensibilidade Independente

The failure of independent extensibility

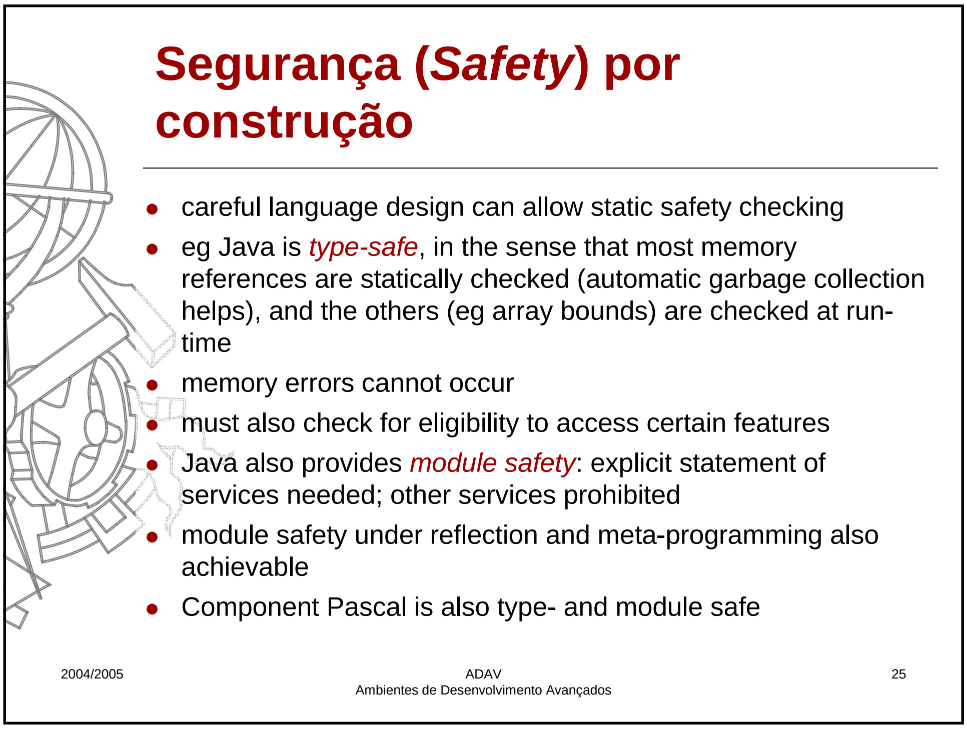
- IE explored in research projects, but failed in industrial projects (eg Taligent)
- partitioning into small components compromises performance
- eg in micro-kernel OS, frequent crossing of protection boundaries
- initial euphoria for micro-kernel OSs evaporated; eg NT4 moved significant parts of display driver code into NT kernel to improve performance of graphics-intensive applications



O paradigma da Extensibilidade Independente

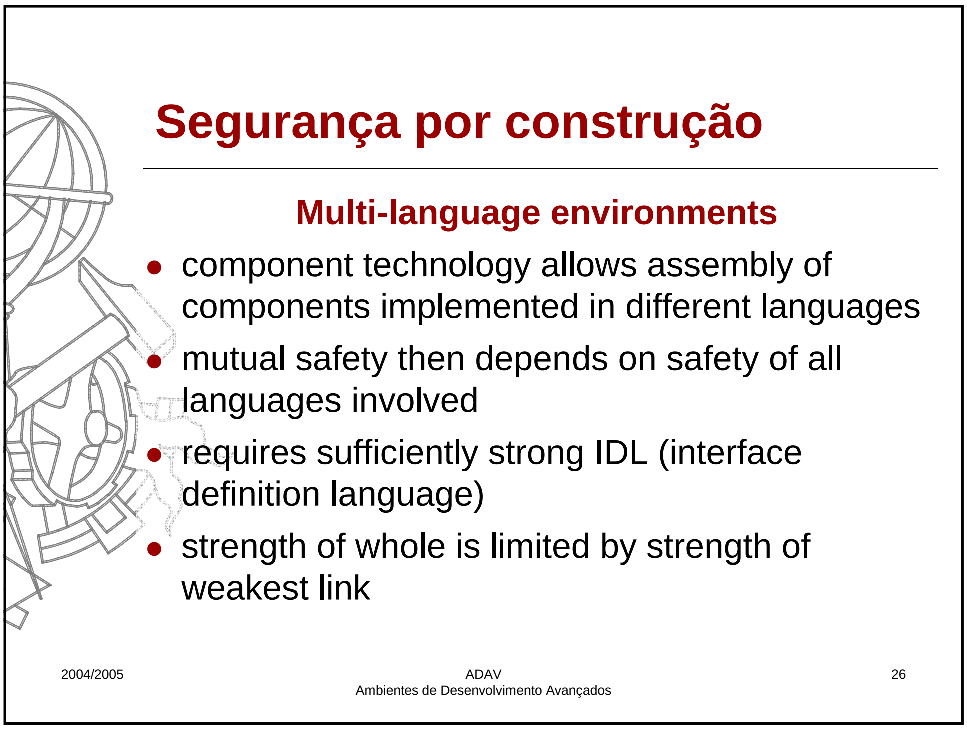
The solution?

- how can IE be viable if performance so badly affected?
- ask rather, why is performance so badly hit?
- cross-context calls expensive (on well-tuned OS, about 100 times slower than local in-process call)
- ok for time-sharing of traditional OSs, with IPC based on buffered pipelines, but not for tightly-interacting components with synchronous calls
- contexts typically not used in PCs (MacOS, MSDOS): no hardware protection, no context-switching, plug-ins share address space with and may crash entire system
- how to combine efficiency and safety?
- choose granularity carefully
- statically check safety, run unprotected



Segurança (*Safety*) por construção

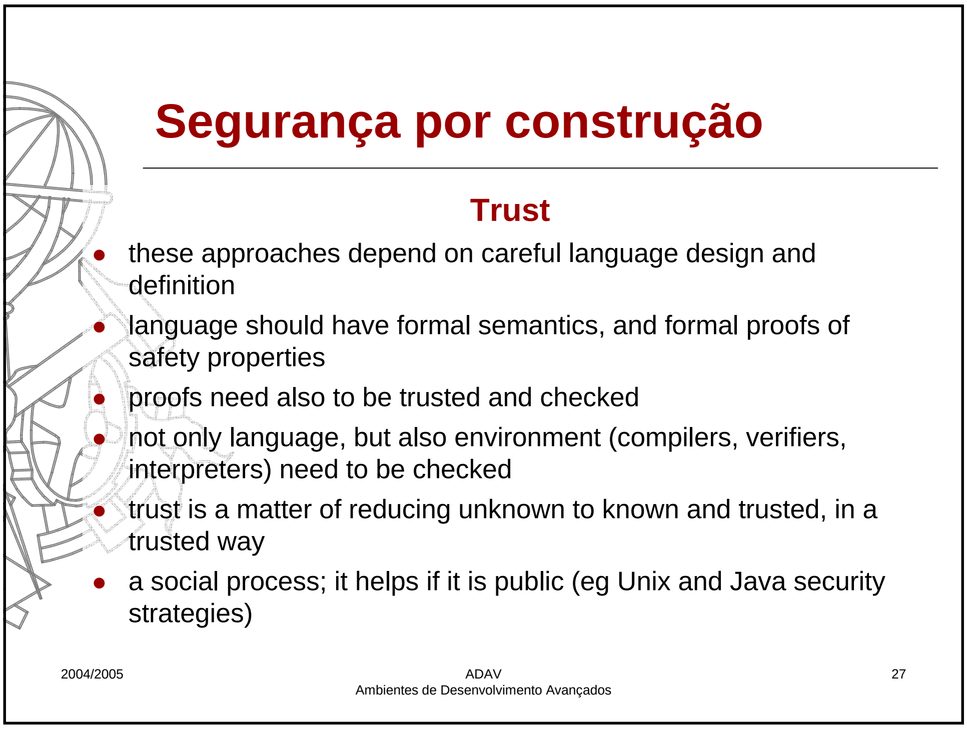
- careful language design can allow static safety checking
- eg Java is *type-safe*, in the sense that most memory references are statically checked (automatic garbage collection helps), and the others (eg array bounds) are checked at run-time
- memory errors cannot occur
- must also check for eligibility to access certain features
- Java also provides *module safety*: explicit statement of services needed; other services prohibited
- module safety under reflection and meta-programming also achievable
- Component Pascal is also type- and module safe



Segurança por construção

Multi-language environments

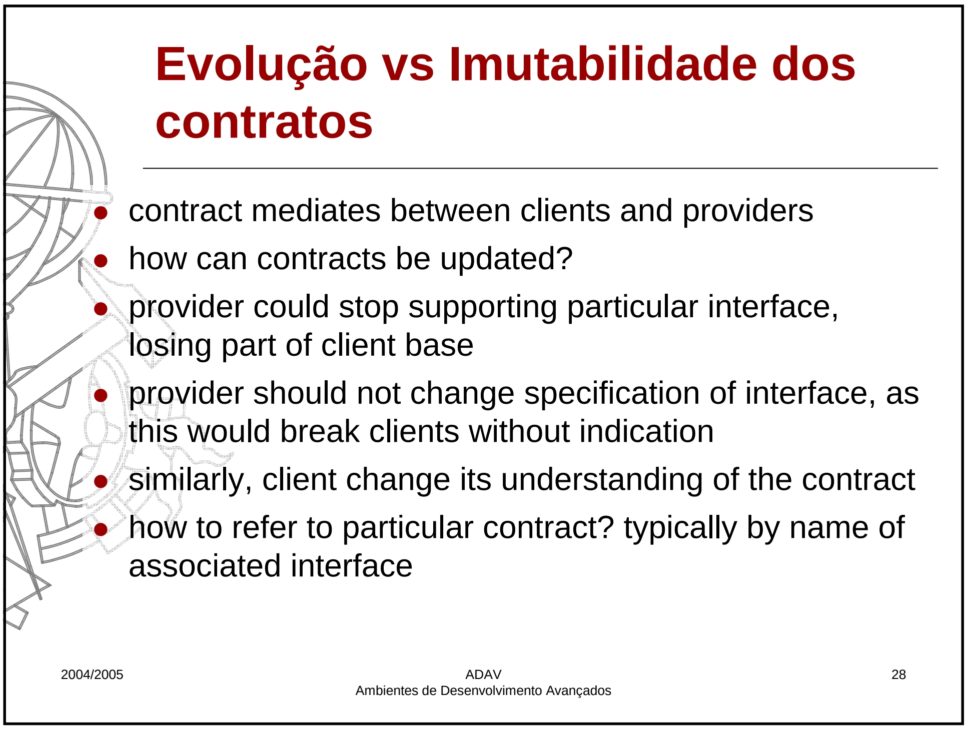
- component technology allows assembly of components implemented in different languages
- mutual safety then depends on safety of all languages involved
- requires sufficiently strong IDL (interface definition language)
- strength of whole is limited by strength of weakest link



Segurança por construção

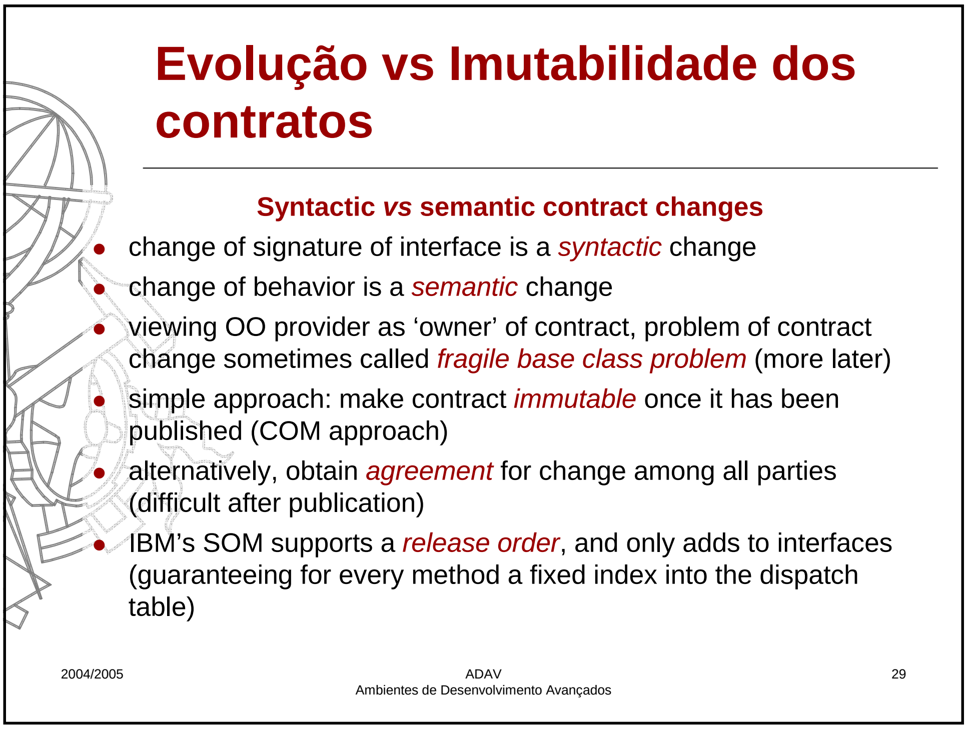
Trust

- these approaches depend on careful language design and definition
- language should have formal semantics, and formal proofs of safety properties
- proofs need also to be trusted and checked
- not only language, but also environment (compilers, verifiers, interpreters) need to be checked
- trust is a matter of reducing unknown to known and trusted, in a trusted way
- a social process; it helps if it is public (eg Unix and Java security strategies)



Evolução vs Imutabilidade dos contratos

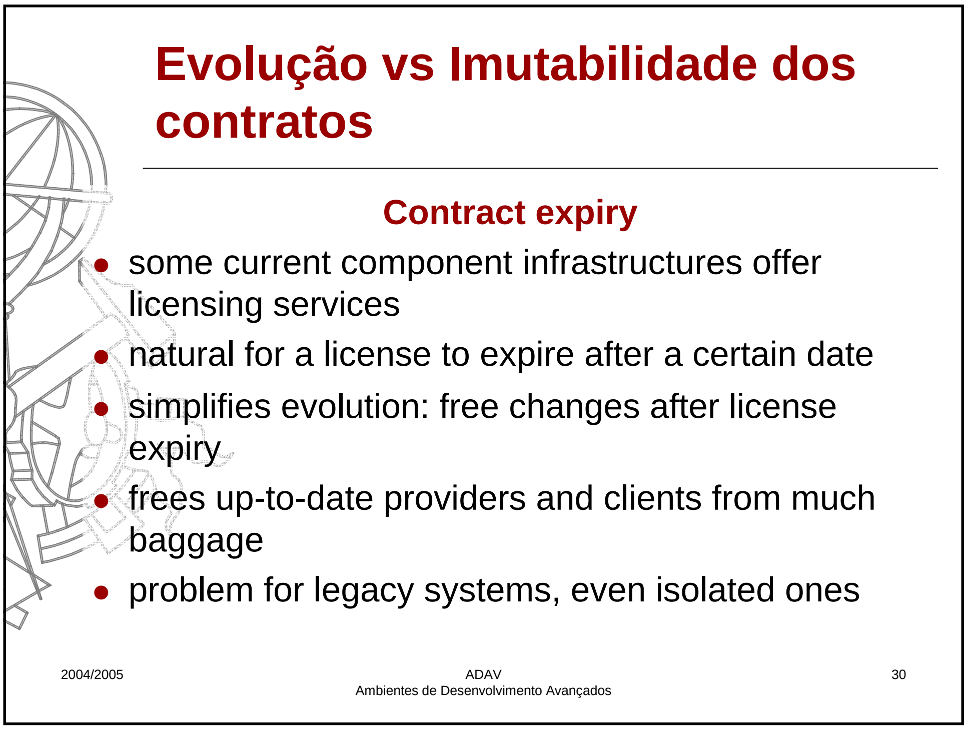
- contract mediates between clients and providers
- how can contracts be updated?
- provider could stop supporting particular interface, losing part of client base
- provider should not change specification of interface, as this would break clients without indication
- similarly, client change its understanding of the contract
- how to refer to particular contract? typically by name of associated interface



Evolução vs Imutabilidade dos contratos

Syntactic *vs* semantic contract changes

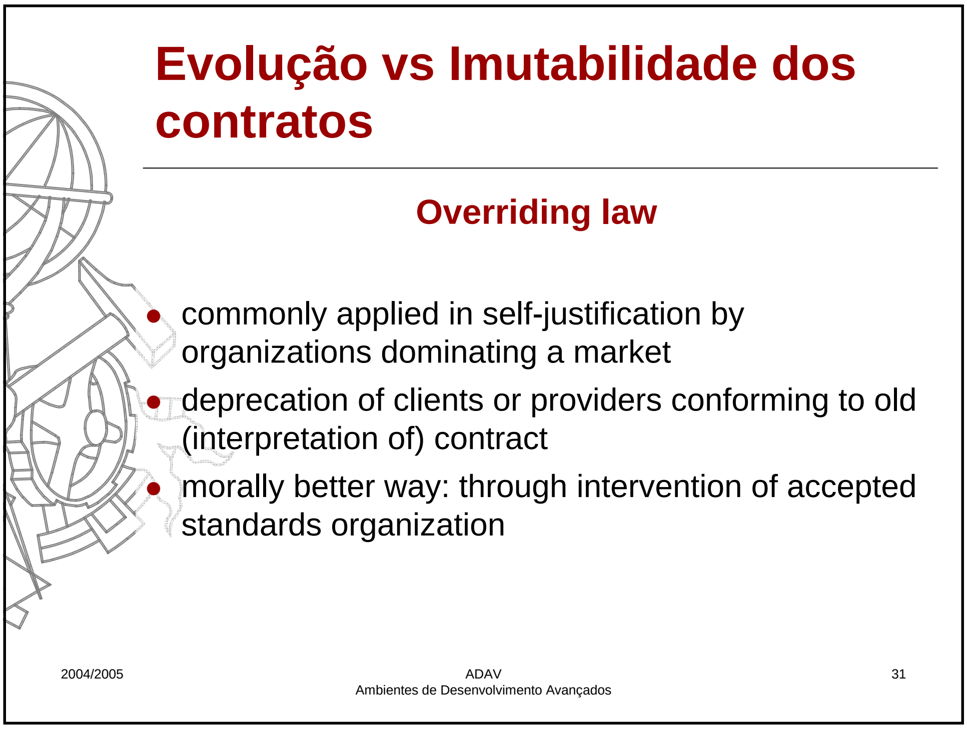
- change of signature of interface is a *syntactic* change
- change of behavior is a *semantic* change
- viewing OO provider as ‘owner’ of contract, problem of contract change sometimes called *fragile base class problem* (more later)
- simple approach: make contract *immutable* once it has been published (COM approach)
- alternatively, obtain *agreement* for change among all parties (difficult after publication)
- IBM’s SOM supports a *release order*, and only adds to interfaces (guaranteeing for every method a fixed index into the dispatch table)



Evolução vs Imutabilidade dos contratos

Contract expiry

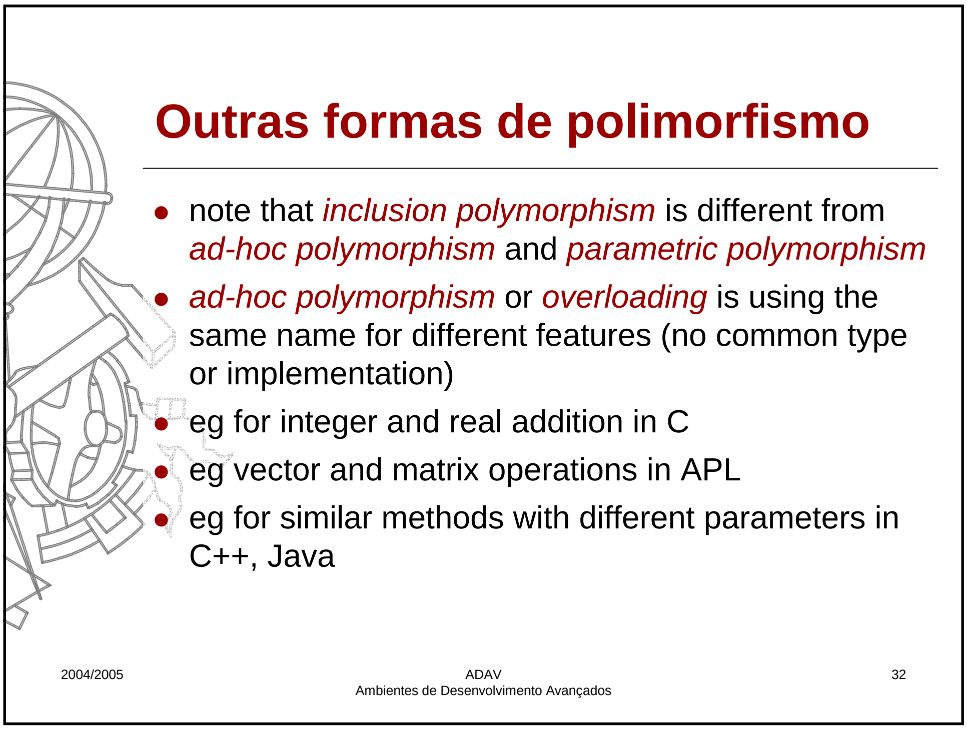
- some current component infrastructures offer licensing services
- natural for a license to expire after a certain date
- simplifies evolution: free changes after license expiry
- frees up-to-date providers and clients from much baggage
- problem for legacy systems, even isolated ones



Evolução vs Imutabilidade dos contratos

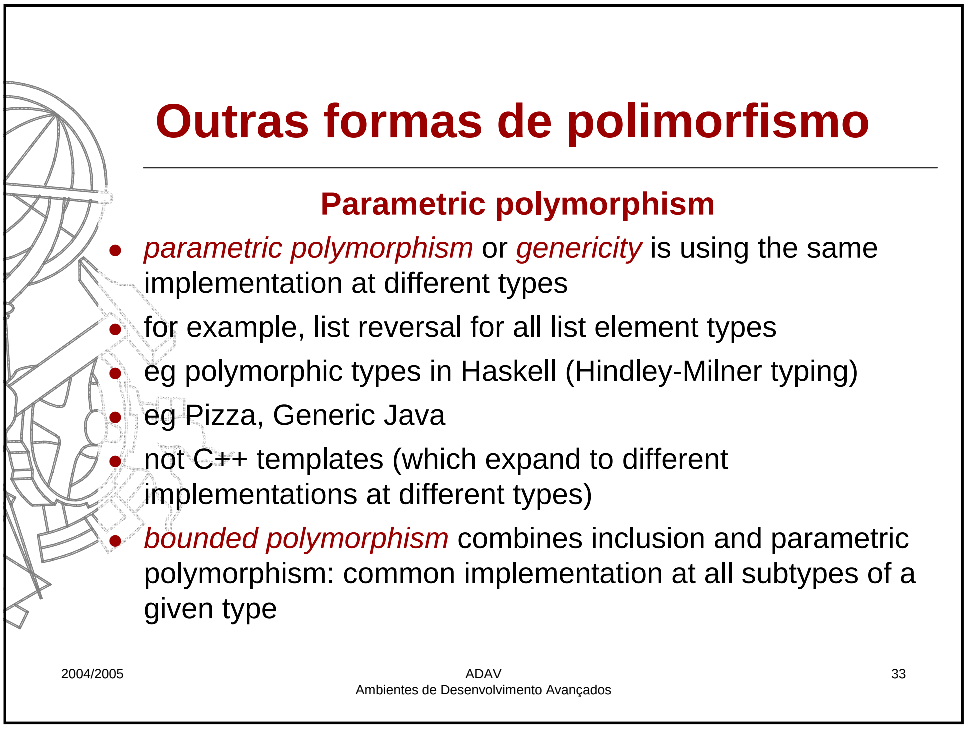
Overriding law

- commonly applied in self-justification by organizations dominating a market
- deprecation of clients or providers conforming to old (interpretation of) contract
- morally better way: through intervention of accepted standards organization



Outras formas de polimorfismo

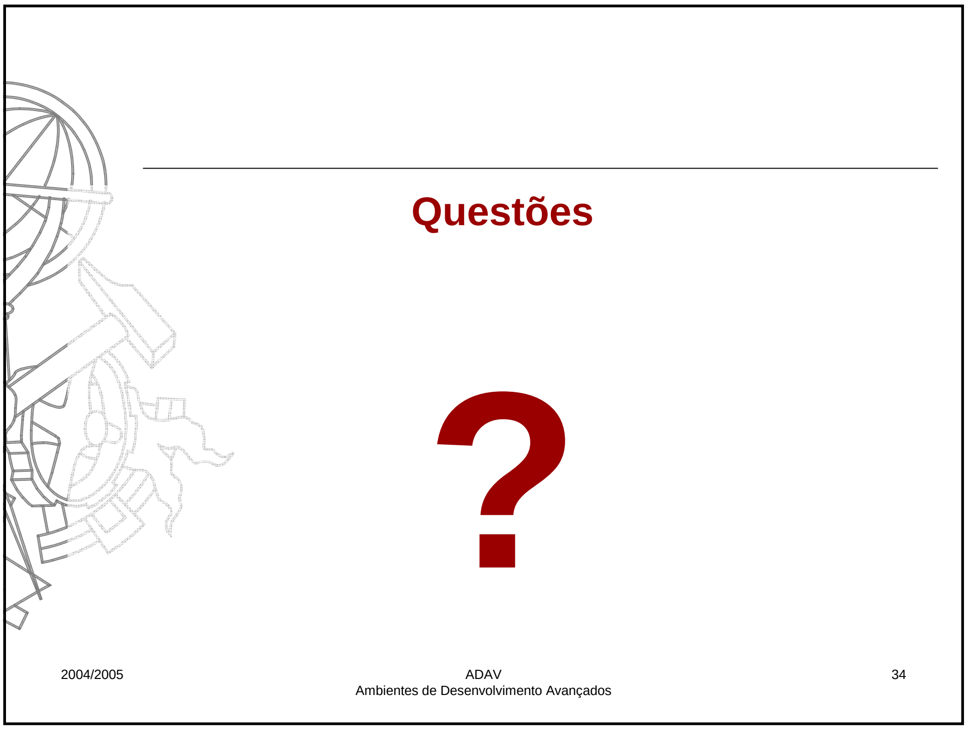
- note that *inclusion polymorphism* is different from *ad-hoc polymorphism* and *parametric polymorphism*
- *ad-hoc polymorphism* or *overloading* is using the same name for different features (no common type or implementation)
- eg for integer and real addition in C
- eg vector and matrix operations in APL
- eg for similar methods with different parameters in C++, Java



Outras formas de polimorfismo

Parametric polymorphism

- *parametric polymorphism* or *genericity* is using the same implementation at different types
- for example, list reversal for all list element types
- eg polymorphic types in Haskell (Hindley-Milner typing)
- eg Pizza, Generic Java
- not C++ templates (which expand to different implementations at different types)
- *bounded polymorphism* combines inclusion and parametric polymorphism: common implementation at all subtypes of a given type



Questões

?