Ambientes de Desenvolvimento Avançados

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

Aula 4

Engenharia Informática

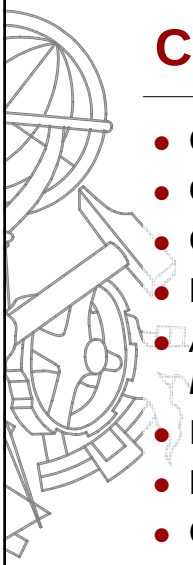
2004/2005

José António Tavares
jrt@isep.ipp.pt

O que é um componente e o que não é?

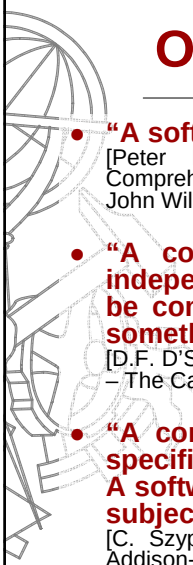
Capítulo 4 de:

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



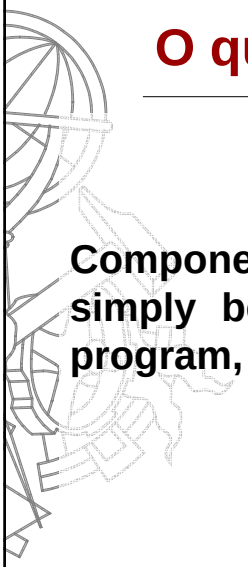
Conteúdo

- Componentes
- Objectos
- Componentes e Objectos
- Módulos
- Abstração e Reutilização : *WhiteBox* vs *BlackBox*
- Interfaces
- Dependências de Contexto Explícitas
- Componentes – “Peso”



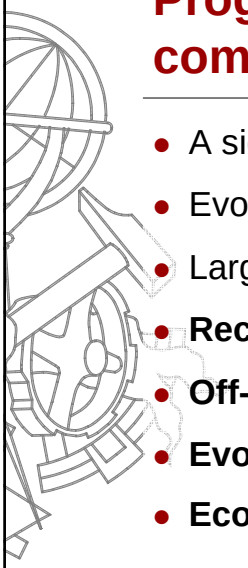
O que é um componente?

- **“A software package which offers *service* through *interfaces*”**
  [Peter Herzum and Oliver Sims, “Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise”, John Wiley & Sons, Incorporated, 1999].
- **“A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger”**
  [D.F. D’Souza and A.C. Wills, “Objects, Components, And Frameworks with UML – The Catalysis Approach” Addison-Wesley, 1998].
- **“A component is a unit of composition with contractually specified interfaces and *explicit context dependencies* only. A software component can be deployed independently and is subject to composition by third parties.”**
  [C. Szyperski, “Component Software: Beyond Object-Oriented Programming” Addison-Wesley, 1998].



O que não é um componente?

Component isn't an object, not in sense of simply being an object in a Java or C++ program, although it is true at runtime.



Programação orientada a componentes

- A sign of **maturity**
- Evolved from **Object-Oriented**
- Large scale **reuse**
- **Reconfigurable** capabilities
- **Off-the-shelf** components
- **Evolutionary** refinement
- **Economically** scaling

Termos e Conceitos

Componentes

- pre-built binary units
- characteristic properties of a component are:
  - **unit of independent deployment** – it needs to be well separated from its environment and other components;
  - **unit of a thirtd-party composition** – it need to be sufficient self-contained and needs to come with clear specification of what it requires and provides;
  - **has no (external) observable state** – it is required that the component cannot be distinguished from copies of its own.

Termos e Conceitos

Objectos

- objects are not sold, bought or deployed, the unit of deployment is something more static such as a class or rather a library or framework of classes
- characteristic properties of an object are:
  - a unit of instantiation with a unique identity;
  - may have state and this can be externally observable;
  - encapsulates its state and behavior,
  - instances of classes or clones of *prototype* objects;
  - Intitialization: *constructor* (static procedure) or *object factory* (separate object)

Termos e Conceitos

Componentes e Objectos

- a component is likely to act through objects – normally consists of one or more classes;
- however, there is no need for a component to contain classes only, or even contain classes at all – might contain global procedures or static variables; might be implemented in functional or assembly language;
- state maintained by an object is abstracted by that object's reference – a component that does not maintain observable state cannot (observably) maintain references even to the objects it creates;
- just as classes can depend on other classes using inheritance, components can depend on other components – the superclasses of a class do not necessarily need to reside in the same component as the class itself;

Exemplos de Componentes:

- Potential Examples:
  - Procedures ?
  - Classes ?
  - Modules ?
  - entire application ?
- Are not components:
  - C macros
  - C++ templates
  - Smalltalk blocks

Termos e Conceitos

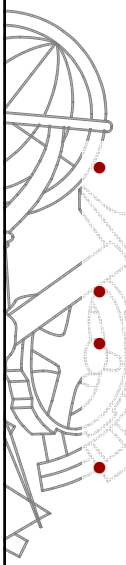
Módulos

- Modular languages – Molula-2 (Wirth, 1982) and Ada;
- With Eiffel, it was claimed that a class is a better module (Meyer, 1988);
- In more recent languages designs – such as Oberon, Modula-3, Component Pascal and C# – the notions of modules and classes are kept separated;
- Modules as opposed to classes can be seen as **minimal components**: they package multiple classes, can be compiled separately and deployed independently;
- There are cases where modules do not qualify as components – modules can be build to use global (static) variables to expose observable state.

Termos e Conceitos

Abstração e Reutilização : *WhiteBox* vs *BlackBox*

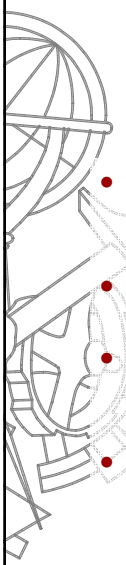
- In ideal **blackbox** abstraction a client cannot see beyond the interface, implementations are reused without building on anything else than their interface
- In **whitebox** abstraction the interface may still enforce encapsulation and limit what clients can do but the implementation is fully available and implementation inheritance allows for substantial interference
- **Glassbox** reuse allows inspection of the implementation but not interference
- **Grayboxes** are those that reveal a controlled part of their implementation



Termos e Conceitos

Interfaces

- Define component's access points – allow clients of a component, usually components themselves, to access services provided;
- Interface specifies signature and behavior;
- Normally, multiple interfaces are provided corresponding to different access points, each representing a service that component offers;
- Emphasizing the contractual nature of the interface specifications is important because the component and its clients are developed in mutual ignorance.



Termos e Conceitos

Dependências de Contexto Explícitas

- Components have to specify their needs – specification of what the deployment environment will need to provide so that the component can function;
- These needs are called context dependencies, referring to the context of composition and deployment;
- If there were only one software component world, it would suffice to enumerate requires interfaces of the other components to specify all context dependencies;
- In reality, there are several component worlds that partial coexist, partial compete and partially conflict with each other – OMG's CORBA, Sun's Java and Microsoft's COM and CLR.

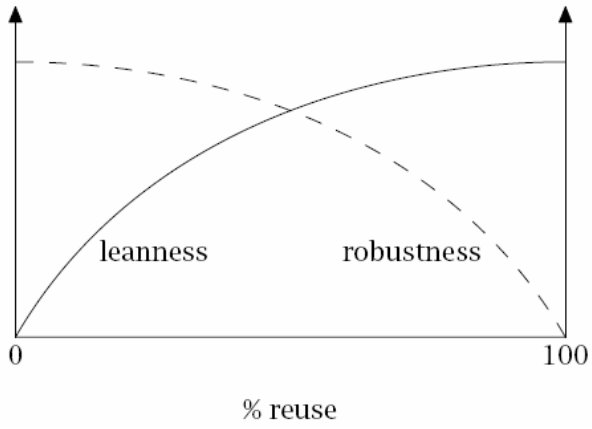
Termos e Conceitos

"Peso" de Componentes

- Fat **Components**
  - The component is self-contained and can function under weak environmental guarantees
  - The context dependencies are reduced making the component more robust over time
  - But a component with everything bundled in is not a component anymore
- Lean **Components**
  - Other components are (re)-used to achieve the component's services
  - The context dependencies increase making the component more vulnerable in case of context evolution
  - Re-use is maximized, use is compromised

Termos e Conceitos

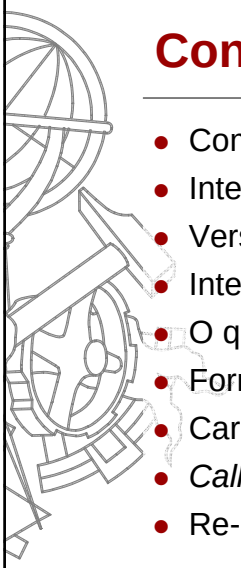
Compromisso entre 'ágil' (*leanness*) e robustez



Componentes, Interfaces e re-entrada.

Capítulo 5 de:

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

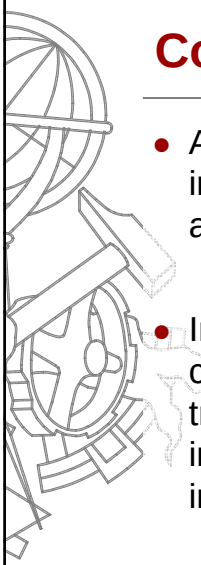


Conteúdo

- Componentes e interfaces
- Interfaces directas e indirectas
- Versões
- Interfaces como contrato
- O que pertence a um contrato?
- Formalidade ou informalidade?
- Características não documentadas
- *Callbacks* e contractos
- Re-entrada nos objectos

Componentes e interfaces

- Interfaces are the means by which components connect. Technically, an interface is a set of named operations that can be invoked by clients.
- Each operation's semantics is specified, and this specification plays a dual role as it serves both providers implementing the interface and clients using the interface.



Componentes e interfaces

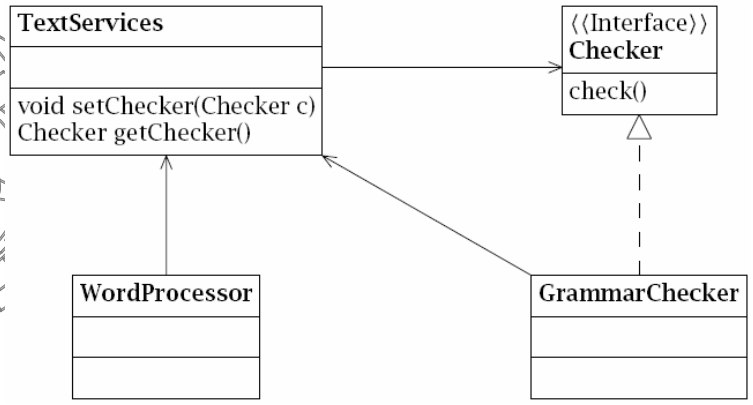
- A component may either directly provide an interface or implement objects that, if made available to clients, provide interfaces.
- Interfaces **directly** provided by a component correspond to procedural interfaces of traditional libraries. Such **indirectly** implemented interfaces correspond to object interfaces.

Interfaces Directas e indirectas

- A procedural (direct) interface to a component is modeled as an object interface of a static object within the component.
- An object (indirect) interface introduces an indirection called method dispatch or, sometimes, dynamic method lookup.

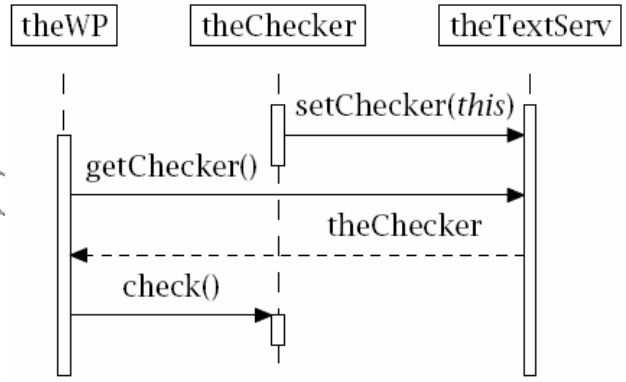
Interfaces Directas e indirectas

Example of indirection: classes



Interfaces Directas e indirectas

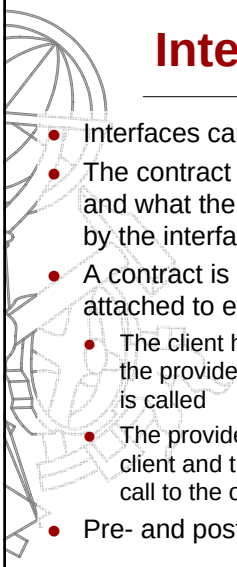
Example of indirection: messages



Versões

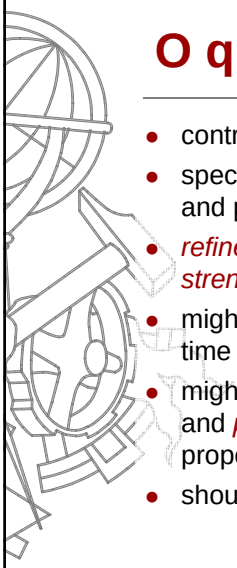
- Traditional version management assumes that the versions of a component involve at a single source. In a free market, the evolution of versions is more complex and management of version numbers can become a problem in its own right.

- With direct interfaces it suffices to check versions at bind time, which is when a service is first requested.
- In indirect interfaces couple arbitrary third party.
- In a versioned system, care must be taken to avoid indirect coupling of parties that are of incompatible versions.
- The goal is to ensure that older and newer components are either compatible or clearly detected as incompatible.



Interfaces como contrato

- Interfaces can be viewed as contracts between provider and consumer;
- The contract states what the client needs to do to use the interfaces and what the provider has to implement to meet the services promised by the interface;
- A contract is an appropriate approach, with **pre-** and **post-conditions** attached to every operation
  - The client has to establish the pre-condition before calling the operation and the provider can rely on the precondition being met whenever the operation is called
  - The provider has to establish the post-condition before returning to the client and the client can rely on the post-condition being met whenever the call to the operation returns
- Pre- and post-conditions are not the only way to form contracts.



O que pertence a um contrato?

- contract = signature + behavioral specification;
- specifies *requirements* and *guarantees*, perhaps using pre- and post-conditions;
- *refinements* (eg revisions) may *weaken preconditions* and/or *strengthen post conditions*
- might also specify *non-functional requirements* (eg speed, time complexity, space)
- might also specify *safety* (“this bad thing will never happen”) and *progress* (“this good thing will eventually happen”) properties
- should be *rigorous*; may be *formal*

Formalidade ou informalidade?

- None of the real-world laws are formal. New “interpretations” are found every day and tested in court.
- Interface contracts should be as formal as possible to derive all necessary information and to enable formal verification – this is complex and, therefore, rarely used in practice;
- Different parts of a system can be specified using different degrees of formality – the preciseness of the specification have to be balanced against the critically of the target part.

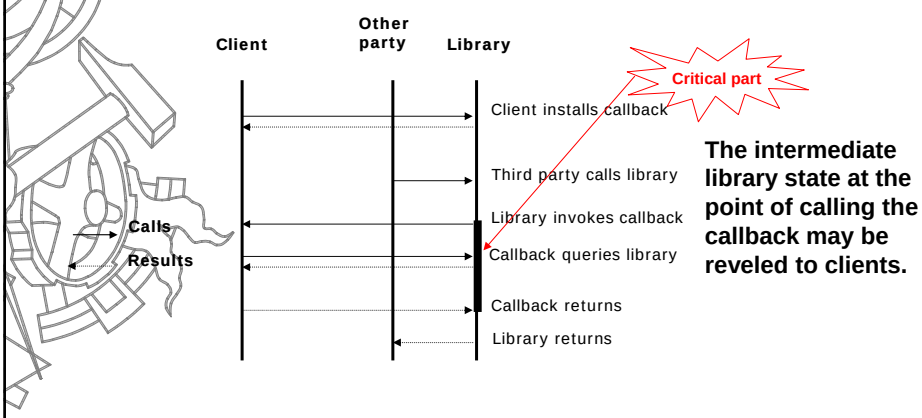
Características não documentadas

- always possible to observe behavior of implementation (eg testing, debugging, espionage)
- may provide more information than specification
- depending on such information is dangerous
- no guarantee that later versions will behave the same
- no guarantee even that this version always behaves the same

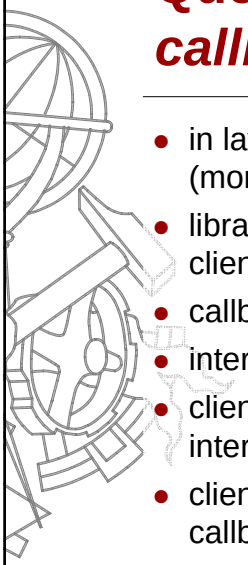
Callbacks e contractos

- *Callback* or *up-call* is procedure *registered with* and subsequently called by a library
- *Callbacks* are a common feature in procedural libraries that have to handle asynchronous events.
- A callback usually reverses the direction of the flow of controls, so a lower layer calls a procedure in a higher layer.
- The resulting contract are far less manageable than simple pre- and post-conditions.
- Validity of the library state is specified as part of a contract.

Callbacks e contractos

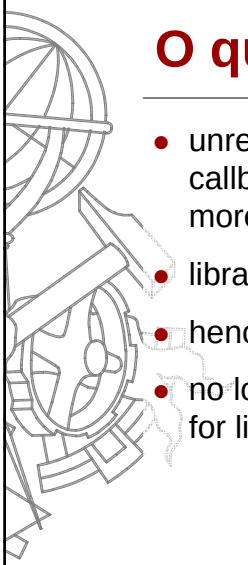


The intermediate library state at the point of calling the callback may be reveled to clients.



Que é têm de especial os *callbacks*?

- in layered architecture, calls originate in higher (more abstract) layer and move downwards
- library operations complete before returning to client, who cannot observe intermediate states
- callback usually reverses this flow
- intermediate state of library becomes visible
- client may observe, or even modify, library's intermediate state
- client certainly observes identity and ordering of callbacks



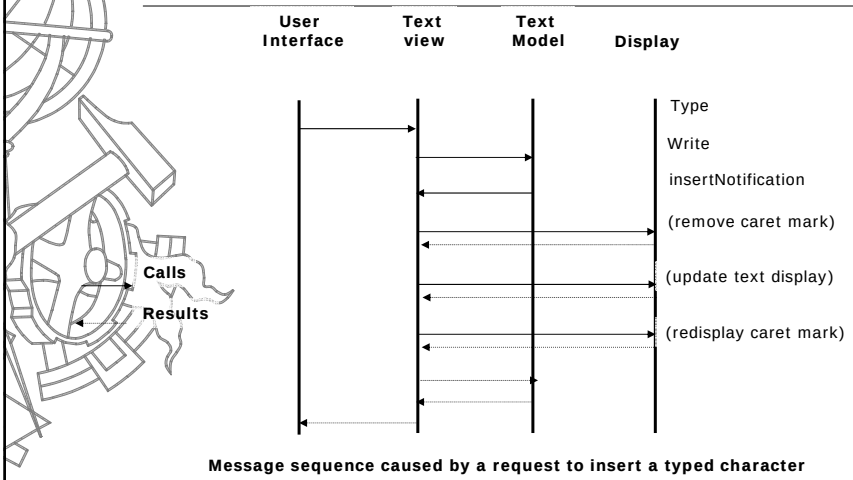
O que é que se pode fazer

- unrealistic to restrict behavior of client during callback (most non-trivial callbacks query library for more information before taking appropriate action)
- library state must remain valid while observable
- hence must remain valid during callbacks
- no longer sufficient to give pre- and postconditions for library

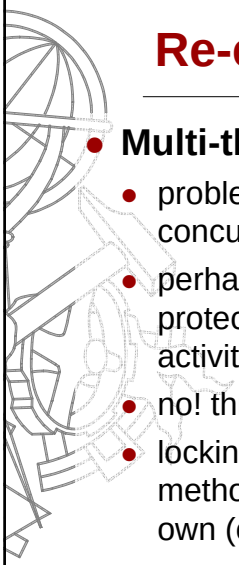
Re-entrada nos objectos
(*re-entrance*)

- The object re-entrance is the situation in which an object´s method is invoked while another method is still executing.
- The real problem is observation of an object undergoing a state transition with inconsistent intermediate states becoming visible. Considering object re-entrance, the problem is when an object´s method is invoked while another method is still executing.
- Recursion and re-entrances become even more pressing problem when crossing the boundaries of components.

Re-entrada nos objectos
(*re-entrance*)

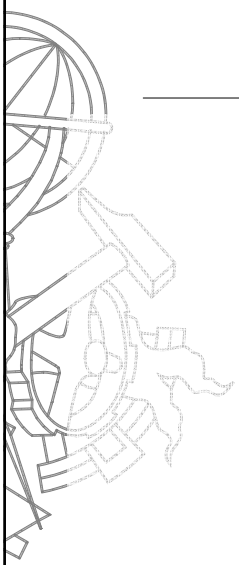


Message sequence caused by a request to insert a typed character



Re-entrada nos objectos

- **Multi-threading**
  - problems of recursive re-entrance similar to those of concurrent interaction
  - perhaps helps to make objects *thread-safe*? (ie protected from unwanted interference from concurrent activities)
  - no! thread safety addresses only external re-entrance
  - locking prevents other objects from invoking our methods, but cannot prevent us from invoking our own (or self-inflicted deadlocks would result)



Questões

?