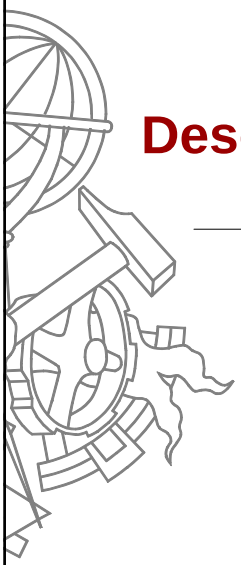


Ambientes de Desenvolvimento Avançados

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

Aula 5

Engenharia Informática

2004/2005

José António Tavares
jrt@isep.ipp.pt

Components and Interfaces
Part 2

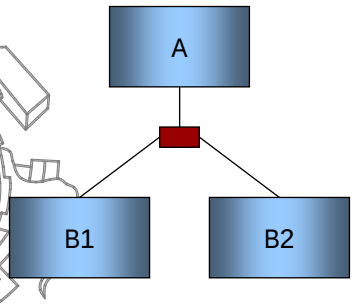
Complemento ao Capítulo 5 de:

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

Conteúdo

- Interfaces as Contracts
- Interfaces vs. Abstract Classes
- Specifying Pre- and Post-conditions
- Using assertions to check Conditions

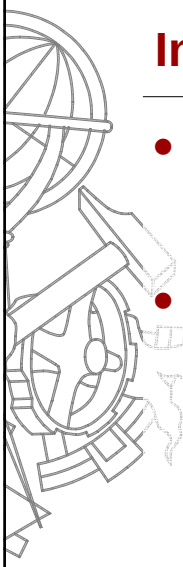
Components and Interfaces



A *requires* certain functionality to fulfil its obligations.

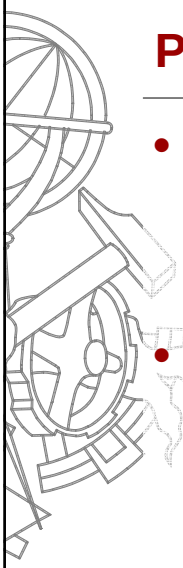
B1 and/or B2 *provide* that functionality.

An interface *mediates* between clients and providers.



Interfaces as contracts

- Can view an interface specification as a “contract” between the client and a provider
- So the interface specification must state:
  - What the client needs to do
  - What a provider can rely on
  - What a provider must promise in return
  - What the client can rely on



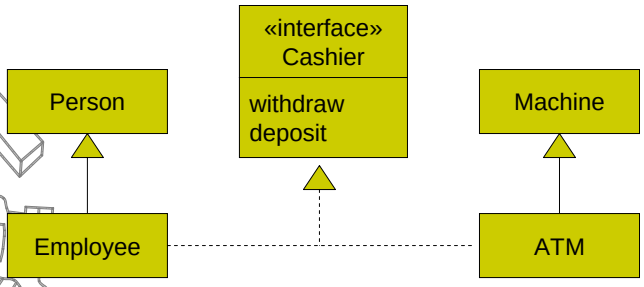
Pre- and Post-Conditions

- **Pre-conditions**:
  - What the client must establish before calling the operation
  - The provider can rely on this condition being true whenever the operation is called
- **Post-conditions**:
  - What the provider must establish before returning to the client
  - The client can rely on this condition being true whenever the call to the operation returns

Interfaces

- Sometimes we may find two or more different subclasses share some common behaviour
- In this case, they are not strictly “kinds of” some common parent
- Rather, they “behave like” some common pattern (under certain circumstances)
- We say that they both implement a common interface

A “Cashier” Interface



Java interfaces

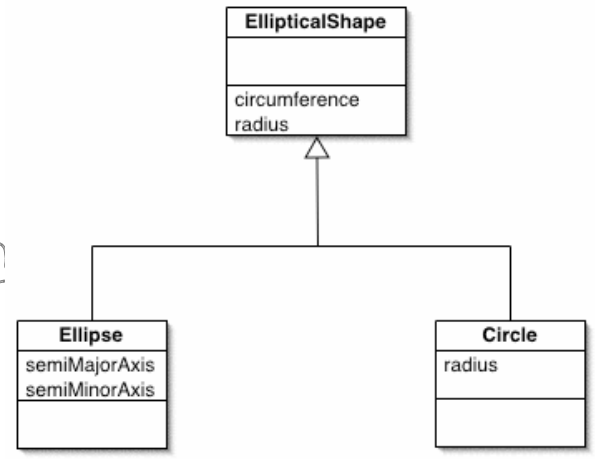
```
public interface Cashier
{
    public void deposit(int id, double amount);
    public boolean withdraw(int id, double amount);
}
```

- Note:
  - An interface is pure specification - it contains no implementation
  - Identical in C#

Conteúdo

- Interfaces as Contracts
- Interfaces vs. Abstract Classes
- Specifying Pre- and Post-conditions
- Using assertions to check Conditions

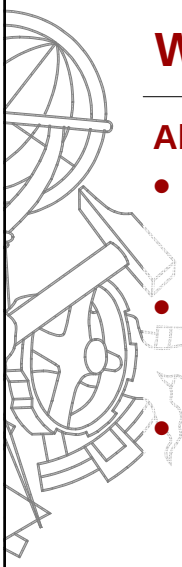
Abstract Classes



Abstract Class vs. Interface

```
public abstract class EllipticalShape
{
  public abstract double area( );
  public abstract double circumference( );
}


public interface Cashier
{
  public void deposit(int id, double amount);
  public boolean withdraw(int id, double amount);
}
```



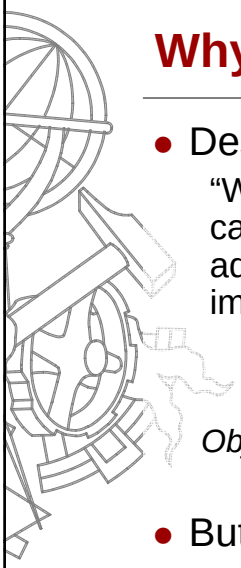
What are the differences?

Abstract Class

- Can include class and instance fields or properties
- May include concrete implementations of methods
- A concrete class can only extend a *single* abstract class

Interface

- Can declare properties
- All methods must be abstract declarations - no implementation
- A class can implement *multiple* interfaces



Why use interfaces?

- Design Guideline:

  "When the functionality supported by a class can be implemented in different ways, it is advisable to separate the interface from the implementation"

  Xiaoping Jia

  Object-Oriented Software Development Using Java

- But with components …?

Conteúdo

- Interfaces as Contracts
- Interfaces vs. Abstract Classes
- Specifying Pre- and Post-conditions
- Using assertions to check Conditions

Back to interfaces?

- Design Guideline:

  "When the functionality supported by a class can be implemented in different ways, it is advisable to separate the interface from the implementation"

  Xiaoping Jia

- And we can cast an object into its interface type if we only want to access its services

List as an example

```
public interface List {
  public int size( );
  public boolean isEmpty( );
  public Object element(int i);
  public Object head( );
  public Object last( );
  public void insert(Object item, int i);
  public void insertHead(Object item);
  public void insertLast(Object item);  //
  more…
}
```

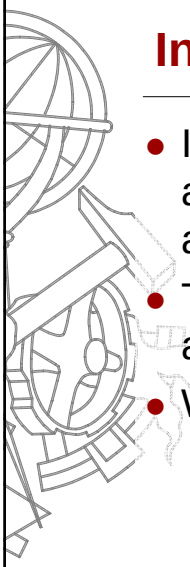

Implementations of List

- We can have several different implementations of the List interface:

```
public class LinkedList implements List
{
  <body of Linked List here>
}
```

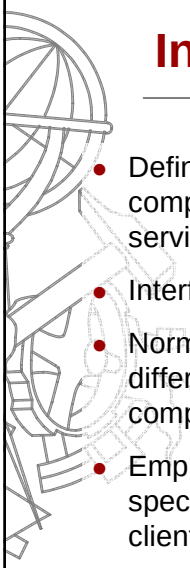
- or:

```
public class DynamicArray implements List
{
  <body of DynamicArray here>
}
```



Interfaces as Contracts

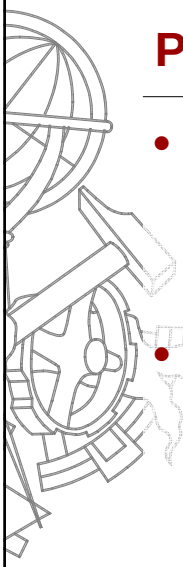
- Interfaces specify the types of their associated methods, but say nothing about their behaviour
- This is true in languages such as Java and C#
- We can improve on that!?



Interfaces as Contracts

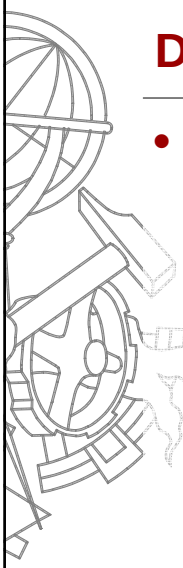
Interfaces

- 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.



Pre- and Post-Conditions

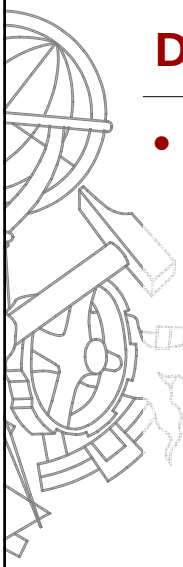
- Pre-conditions:
  - What the client must establish before calling the operation
  - The provider can rely on this condition being true whenever the operation is called
- Post-conditions:
  - What the provider must establish before returning to the client
  - The client can rely on this condition being true whenever the call to the operation returns



Documentation - JAVA

- Use “@pre” and “@post” as special tags for pre-conditions and post-conditions:

```
/**
*   @pre precondition
*   @post postcondition
**/
public void method1(...)
{
   // …
}
```



Documentation – CLR (.NET)

- Use “<pre>” and “<post>” as special tags for pre-conditions and post-conditions if XML comments are used:

```
/// <pre> pre-condition </pre>
/// <post> post-condition </post>
///
public void method1(...)
{
   // …
}
```


A boring example - JAVA

```
/**
* Returns the number of
* elements in a list
*
* @pre true
* @post @nochange
*/
public int size( );
```

A more interesting example - JAVA

```
/**
* Returns the i-th element in the list
*
* @pre i >= 0 && i < size( )
* @post @nochange
*/
public Object element(int i);
```


The same example - CLR (.NET)

```
/// <summary>
/// Returns the i-th element in the list
/// </summary>
/// <pre> i >= 0 && i < size( ) </pre>
/// <post> @nochange </post>
///
public Object element(int i);
```

Caracteres <, > e & - CLR (.NET)

Cuidado quando é necessário processar os comentários em XML

- < <
- > >
- & &

Other logical expressions

==>	logical implication
<==>	logical equivalence
@forall	*x* : *Range* @ *Expression*
	Universally quantified expression
@exists	*x* : *Range* @ *Expression*
[*m* … *n*]	Integer range from *m* to *n*

A nasty example

```
/**
* Inserts a new element into a list
* at the i-th position
*
* @pre item != null && i >= 0 && i <= size( )
* @post size( ) == size( )@pre + 1
* @post @forall k : [0 .. size( ) - 1] @
*    (k < i ==> element(k)@pre == element(k)) &&
*    (k == i ==> item@pre == element(k)) &&
*    (k > i ==> element(k-1)@pre == element(k)
*/
public void insert(Object item, int i);
```


An exercise

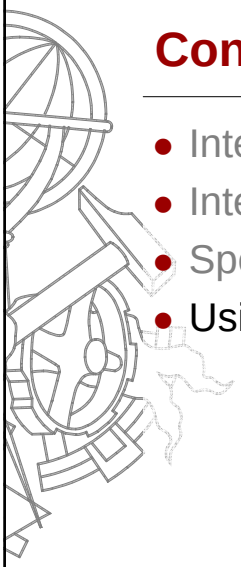
Use this method to specify the interface to a method that inserts a new element to the head of a list

Solution

```
/**
* Inserts a new element at the head of a list
*
* @pre item != null
* @post size( ) == size( )@pre + 1
* @post item@pre == element(0)
* @post @forall k : [1 .. size( ) - 1]
        @ element(k-1)@pre == element(k)
*/
public void insertHead(Object item);
```

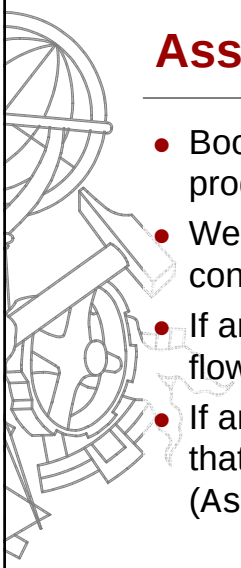

How does this help?

- As this stands, the comments provide guidance only
- They prescribe what should be done, but are agnostic about whether a specific implementation satisfies them
- We can, however, provide run-time checks through the use of *Assertions*



Conteúdo

- Interfaces as Contracts
- Interfaces vs. Abstract Classes
- Specifying Pre- and Post-conditions
- Using assertions to check Conditions



Assertions are?

- Boolean conditions that are inserted into a program at strategic points
- We expect them to be true when the flow of control reaches each respective point
- If an assertion is *true*, it has no effect and the flow of control continues
- If an assertion is *false*, then execution stops at that point and an `AssertionError` – Java (`AssertionException` - C#) is thrown

Where do you put them?

- Assertions on the pre-conditions of a method should be placed immediately on entry to the method
- Assertions on post-conditions should be placed just prior to the return statement
- But remember that any references to the pre-state of variables must be instantiated upon entry to the method

LinkedList implements List – *Java* Code

```
/**
 * @pre item != null && i >= 0 && i <= size( )
 * @post size( ) == size( )@pre + 1
 */
public void insert(Object item, int i) {
  assert item != null && i >= 0 && i <= size( );
  int size_pre = size( );
  //
  // < body of method here ... >
  //
  int size_post = size( );
  assert size_post == size_pre + 1;
}
```

Assertions in CLR (.NET)

- Debug Mode – Checks only when in debug configuration

  `Debug.Assert(...)`

- Trace Mode – Checks always, debug and release configurations

  `Trace.Assert(...)`

Assertions in CLR (.NET)

```
[Visual Basic]
Public Shared Sub MyMethod(type As Type, baseType As Type)
   Trace.Assert( Not (type Is Nothing), "Type parameter is null", _
      "Can't get object for null type") ' Perform some processing.
End Sub

[C#]
public static void MyMethod(Type type, Type baseType)
{
   Trace.Assert(type != null, "Type parameter is null",
      "Can't get object for null type"); // Perform some processing.
}

[C++]
public: static void MyMethod(Type* type, Type* baseType)
{
   Trace::Assert(type != 0, S"Type parameter is 0",
      S"Can't get Object* for 0 type"); // Perform some processing.
}
```

Eiffel – Pre- Post-Conditions

http://www.eiffel.com/

Expressing assertions

Eiffel provides syntax for expressing pre-conditions (**require**), post-conditions (**ensure**) and class invariants (**invariant**), as well as other assertion constructs such as loop invariants and variants, check instructions.

- **no** : assertions have no run-time effect.
- **require** : monitor pre-conditions only, on routine entry.
- **ensure** : pre-conditions on entry, post-conditions on exit.
- **invariant** : like **ensure**, plus class invariant on both entry and exit for qualified calls.
- **all** : like **invariant**, plus **check** instructions, loop invariants and loop variants

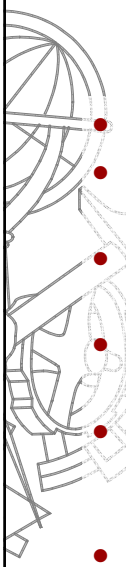
Eiffel – Pre- Post-Conditions

http://www.eiffel.com/

```
indexing
   description: "Simple bank accounts“

class ACCOUNT
   feature -- Access
         balance: INTEGER -- Current balance
         deposit_count: INTEGER is -- Number of deposits made since opening
         do ... code ... end

   feature -- Element change
         deposit (sum: INTEGER) is -- Add `sum' to account.
         require non_negative: sum >= 0
         do
                  ... code ...
         ensure
                  one_more_deposit: deposit_count = old deposit_count + 1
                  updated: balance = old balance + sum
         end
   ...
end -- class ACCOUNT
```



Voltando aos contratos?

- 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*

O que pertence a um contrato?

- How to specify non-functional requirements?
  - `@speed`
  - `@complexity`
    - `@time`
    - `@space`
  - …
- What about explicit context dependencies?

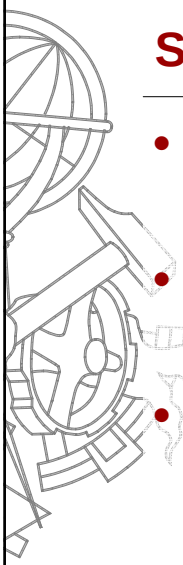
O que pertence a um contrato?

- What about XML ?

- `<speed> ... </speed>`
- `<complexity> ... </complexity>`
  - `<time> ... </time>`
  - `<space> ... </space>`
- …

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.



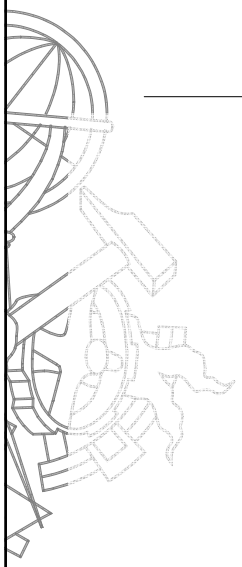
Sumário

- We have discussed the use of interfaces when there may be multiple ways of implementing certain functionality
- Including statements of **pre-** and **post-** conditions in the comment lines for each method declaration makes a richer specification
- Including the pre- and post-conditions in the implementation of each interface adds a further level of rigour (but has limited support in Java and C# at present)

Exercício

```
public interface ISale
{
    /// <pre> ... </pre>
    /// <post> ... </post>
    ///
    DataSet CreateDetails(string user, string pass);

    /// <pre> ... </pre>
    /// <post> ... </post>
    ///
    ShopStatusEnum Add(string user, string pass,
                       long customerID, DateTime date,
                       DataSet dsDetails);
}
```



Questões

?