Modelação e Padrões de Desenho

Capítulo 3
Desenho de Classes

Fernando Miguel Carvalho
DEETC
Instituto Superior de Engenharia de Lisboa,
Centro de Cálculo
mcarvalho@cc.isel.ipl.pt

Design Guidelines

Tipos de classes

Públicas:

- Reside num ficheiro com o nome da classe e extensão .java;
- Para utilização geral.

Auxiliares (*helper*):

- **Não devem ser declaradas públicas**;
- Utilizadas na construção de outras classes;
- Opções de desenho:
  1. Se forem usadas apenas por uma classe, então devem ser declaradas dentro do mesmo ficheiro da classe que suporta;
  2. Se forem usadas por várias classes do mesmo *package*, então devem ser declaradas em ficheiros separados.

➔ Em qualquer dos casos se não é pública, nem privada só está acessível às classes do mesmo *package*.

Exemplo: Implementação de uma **lista de nós ligados**

```
public class DoubleLinkedList implements IList {
  protected Node head;
  protected int count;
  ... // implementação de Double Linked List
}
class Node {
  Object element;
  Node next, prev;
}
```

A classe Node não é acessível em nenhuma classe derivada de DoubleLinkedList, que não esteja no mesmo *package*

- Uma especialização da classe **`DoubleLinkedList`** poderia ter interesse em modificar a implementação de **`Node`**.

➔ **No entanto, a visibilidade de `Node` não permite modificá-la.**

➔ **Solução?**

Exemplo: Implementação de uma **lista de nós ligados...**

```
public class DoubleLinkedList implements IList {
  protected Node head;
  protected int count;
  ... // implementação de Double Linked List
  protected static class Node {
    Object element;
    Node next, prev;
  }
}
```

A classe Node já é acessível em qualquer classe derivada de DoubleLinkedList

Característica:

- Se **não** for `static` as instâncias da ***inner class*** terão mais um campo, com a referência para a instância da ***enclosing class*** que instanciou a *inner class*.
  Nesse caso o acesso à instância da enclosing class usa a sintaxe:
  <NomeEnclosingClass>.this.<NomeMembro>
  Ex: `DoubleLinkedList.this.head`

- Se **não** for `static` as instâncias da *inner class* só poderão ser criadas no interior da *enclosing class*.

Membros

Em Java, a ordem dos membros é insignificante, no entanto **são boas práticas**:

- Ordenar os campos consoante as acessibilidades e papeis;
- Organizar os métodos por grupos com a seguinte ordem:
  - Construtores públicos;
  - Métodos públicos de acesso ou de selecção (não mudam o estado dos objectos);
  - Métodos públicos de modificação (modificam o estado dos objectos);
  - Construtores não públicos;
  - Métodos auxiliares.

```
public class AClass {
   <constantes públicas>
   <construtores públicos>
   <métodos de acesso públicos>
   <métodos de modificação públicos>
   <campos não públicos>
   <construtores não públicos>
   <métodos auxiliares não públicos>
   <classes internas>
}
```

Organização recomendada dos membros de uma classe

Membros… Exemplo

```
public class DoubleLinkedList implements List {
   // Constructor
   public DoubleLinkedList() { ... }

   // Acessors
   public int size() { ... }
   public boolean isEmpty() { ... }
   public Object element(int idx) { ... }
   public Object head() { ... }
   public Object last() { ... }

   // Mutators
   public void insert(Object item, int i) { ... }
   public void insertHead(Object item) { ... }
   public void insertTail(Object item) { ... }
   public void remove(int i) { ... }
   public void removeHead() { ... }
   public void removeTail() { ... }

   // Fields
   protected Node head;
   protected int size;

   // Auxiliary Nested class
   protected static class Node { … }
}
```

Desenho de classes - Boas práticas

- **Evitar campos públicos**.
- **A interface pública deve ser completa**.
- **Separação da interface da implementação**:
  - Sempre que uma funcionalidade pode ser implementada de diversas formas (Exº List);
  - Vantagens:
    - **Detalhes de implementação ocultos dos clientes**;
    - **Mudanças na implementação não afectam clientes**.

Documentação do código fonte

Cada comentário:

- precede imediatamente a característica que descreve:
  - Classe, campo, método, construtor, classe interna.
- consiste na descrição textual da característica seguida de um conjunto de tags.

Tag	**Descrição**
@author	Autor. Suporta múltiplas tags.
@version	Versão corrente
@since	Versão onde apareceu a primeira vez
@param	Significado e valores aceites para um parâmetro. Um método pode ter múltiplas tags
@return	Significado e valores aceites para o retorno de um método
@see	Link para a documentação de uma classe ou membro relacionado
@throws	Excepção que pode ser lançada pelo método. Suporta várias tags

Classes Date na *Java Class Library*

java.util.Date

Implementa uma abstracção de um ponto no tempo medido em milissegundos a contar de 01-01-1970 00:00:000 GMT ("*Unix Epoch*")

Métodos	Descrição
`boolean after(Date when)`	• Testa se esta data ocorre após a data especificada.
`boolean before(Date when)`	• Testa se esta data ocorre antes da data especificada.
`int compareTo(Date anotherDate)`	• Compara a ordenação entre dois objectos Date.
`long getTime()`	• Retorna o número de milissegundos que decorrem desde 01-01-1970 00:00:000 GMT (negativo ou positivo, consoante seja antes ou após)
`void setTime(long time)`	• Afecta este objecto Date para representar um ponto no tempo correspondente ao número de milissegundos decorridos desde 01-01-1970 00:00:000 GMT

- Noção de ordem consistente.
  Sejam `d` e `e`, referências para duas instâncias de `java.util.Date`, então:
  `d.after(e)` ⇔ `d.getTime() > e.getTime()`

`java.util.Calendar`

- **`java.util.Date`** não tem conhecimento do dia, mês e ano associado a um ponto no tempo.
- ATENÇÃO: os métodos de **`java.util.Date`** que forneciam essa funcionalidade foram descontinuados e não deverão ser usados.

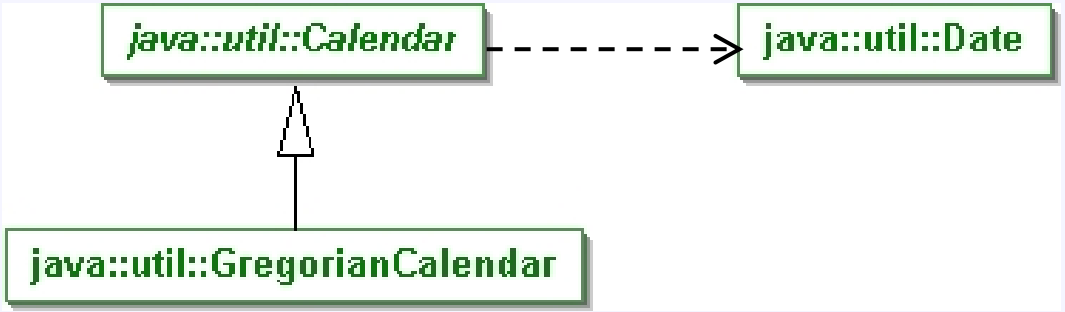
- A responsabilidade de saber:
  - O número de dias de um mês (Janeiro 31 dias, Fevereiro 28 ou 29 dias, etc)
  - Anos bissextos;
  - Primeiro e último mês do ano;
  - etc,

➔ pertence ao tipo **`java.util.Calendar`**.

java.util.Calendar...

- As subclasses de **`Calendar`** implementam as especificidades de cada sistema de calendário, como sejam o calendário Gregoriano, *Julian*, Japonês, etc.

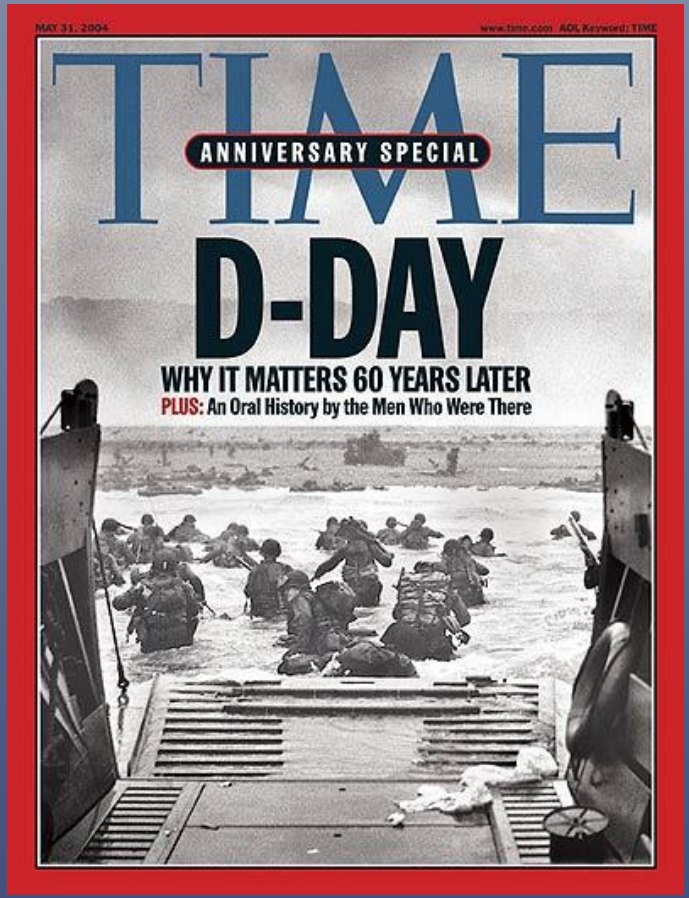
- Usa internamente uma instância de **`java.util.Date`** para a realizar os cálculos temporais.



java.util.Calendar...

Métodos	Descrição
`int get(int field)`	• `field` é uma constante da classe `Calendar`, tal como: `YEAR, MONTH, DATE, HOUR`, etc;
`void set(int field, int value)`	• Afecta um correspondente valor de `field`.
`int add(int field, int increment)`	• Adiciona o `increment` ao correspondente valor de `field`.
`Date getTime()`	• Converte este calendário numa instância de Date.
`Date setTime(Date d)`	• Afecta este calendário ao ponto no tempo representado por `d`.

Caso de estudo: classe `Day`



Classe `Day`

Uma instância de `Day` representa um dia, ignorando a hora associada a esse momento no tempo.

Ao contrário da classe `java.util.Date` os dias não são referenciados a 1 de Janeiro de 1970.

Pretende-se ainda responder a questões como:

- Quantos dias faltam até ao final deste ano?
- Qual a data daqui a 100 dias?

Métodos	Descrição
`int daysFrom(Day other)`	• Calcula o número de dias entre esta data e outra recebida por parâmetro.
`Day addDays(int n)`	• Retorna uma **nova** instância de `Day` correspondente a `n` dias passados desta data.

Classe `Day...`

As operações `addDays` e `daysFrom` são inversas uma da outra.

- `d.addDays(n).daysFrom(d)` ⇔ `n`

  equivalente a escrever: (d + n) – d = n

- `d1.addDays(d2.daysFrom(d1))` ⇔ `d2`

  equivalente a escrever: d1 + (d2 – d1) ⇔ d2

Característica:

- A representação matemática do comportamento de uma classe permite fazer uma **definição formal** das suas propriedades, garantindo uma representação precisa e não ambígua.

Sobrecarga de operadores

Algumas linguagens como o C++ e o C#, permitem a definição de sobrecarga de operadores.

Este mecanismo é implementado pelos compiladores através da tradução dos operadores em métodos com nomes especiais tais como: **`operator+`**, entre outros.

Desta forma seria possível obter outra expressividade na representação de operações entre instâncias de `Day`, como por exemplo:

```
int nr = today - bday;
```

Característica:

- A sobrecarga de operadores pode melhorar a legibilidade, sobretudo em cálculos matemáticos.
  Sendo x, y e z instâncias de `int`, imagine-se o que seria representar z + y * z:
  `x.add(y.multiply(z));`

Encapsulamento

1. *Helper methods*
2. Tipos imutáveis
3. Separar modificadores e assessores
4. *Side effects*
5. A lei de Deméter

1. *Helper methods*

Métodos auxiliares deverão ser privados.

Razões para que métodos auxiliares (*helper methods*) sejam privados:

- Trazem maior entropia à interface pública tornando mais difícil a compreensão da classe;
- Por vezes os métodos auxiliares requerem um protocolo e uma ordem de chamada própria.

  Nestes casos poderá não haver interesse em documentar detalhadamente os métodos, ou até mesmo confiança que os utilizadores os compreendam devidamente.
- Criam uma dependência com os clientes, obrigando à manutenção do protótipo e comportamento desses métodos.

Dica: deverão ser privados métodos auxiliares:

- **não directamente necessárias** à utilização da classe a partir do exterior;
- **não facilmente suportados** em caso de alteração da implementação da classe.

2. Tipos imutáveis

A classe `Day` não disponibiliza modificadores (*mutators*).

A classe `Day` é imutável, tal como a classe `String` do Java.

Deveria a classe `Day` oferecer estes métodos?

- Imagine-se o seguinte cenário:

```
Date deadLine = new Day(2006 , 1, 31);
deadeLine.setMonth(2);
```

Ops!!! Ou será que deveria ficar com 3 de Março?

Dica:

- Não fornecer automaticamente modificadores para os campos de instância, a não ser que exista mesmo essa necessidade.

2. Tipos imutáveis…

Além disso existe ainda outra vantagem:

Característica:
- As instância de tipos imutáveis podem ser livremente partilhadas.

Exemplo do perigo de partilha de instâncias:

```
class CalendarioEscolar{
  ...
  List<Test> mapaDeTestes;

  // Atribui uma data a cada teste do mapaDeTestes;
  void allocateTestDate(int year, int month){
    Day d = new Day(year, month, 1);
    foreach(Test t in mapaDeTestes){
      d.setDate(nextAvailableDayInMonth(month))
      t.setTestDate(d);
    }
  }
  // Devolve o próxima dia livre no mês especificado.
  int nextAvailableDayInMonth(int month){
  ...
  }
...
```

Todos os testes ficariam com a mesma data!

2. Tipos imutáveis…

Dica:
- Campos de instância de classes imutáveis deverão ser marcados como **final**.
- Além disso garantem **melhor desempenho e eficiência** no seu acesso.

Atenção:

```
class Test{
  final Day dataExame;
  ...
}
```

Não protege contra alterações ao conteúdo da instância referida por dataExame.

3. Separar modificadores e assessores

- **Assessores**: não devem fazer modificações sobre o estado do objecto;
- **Modificadores**: devem retornar **`void`**.

Característica:
➔ Ajuda a clarificar o entendimento do utilizador sobre o comportamento dos métodos

Exemplo: não é expectável que o método **`getBalance`** de uma classe **`Account`** modifique o seu saldo.

No entanto existem cenários nas bibliotecas do Java que violam esta regra.

Exemplo:

- método `next()` da classe `Scanner`.
- A alternativa seria ter dois métodos:
  - **`String getCurrent();`**
  - **`void next();`**

3. Separar modificadores e assessores…

Contudo podem existir cenários em que faz sentido violar esta regra.

Exemplo:

- Método `E remove(int index)` da interface `List`

➔ é um modificador porque altera o estado da lista sobre a qual é removido o elemento que ocupa a posição `index`, mas no entanto não retorna `void`.

Faz sentido?

Dica:

- Sempre que possível separar os assessores dos modificadores (idealmente estes últimos retornam `void`).
- Em cenários que faça sentido um modificador retornar um valor, fará sentido também existir um método que retorne esse valor sem fazer a alteração de estado do objecto.

4. *Side Effects*

➔ Um efeito secundário (*side effect*) é uma modificação de dados que é observável quando um método é chamado.

- Um método sem efeitos secundários pode ser chamado diversas vezes, obtendo sempre a mesma resposta (desde que não exista a chamada a um outro método que provoque efeitos secundários sobre o primeiro).

- Em OOP é expectável que um modificador tenha efeitos secundários, nomeadamente sobre o objecto chamado – **parâmetro implícito**.
- Além deste um modificador pode ter efeitos sobre:
  - Campos estáticos acessíveis;
  - **Parâmetros explícitos** (argumentos do método).

- Na maioria das utilizações de OOP os utilizadores não esperam que um método modifique os parâmetros explícitos.

4. *Side Effects…*

Exemplo:

- Sejam `a` e `b` referências para duas colecções.

  A chamada `a.addAll(b)`, deverá adicionar todos os elementos de `b` à colecção referida por a.

➔ O parâmetro implícito, `a`, terá mudado de estado, resultante da adição dos elementos referidos por `b`.

➔ Não é expectável que o conteúdo de `b` se tenha modificado.

Isso seria um **efeito** secundário **indesejado**.

Contudo existem situações em que pode ser desejável uma actualização sobre um parâmetro explicito.

Exemplo:

- Um dos métodos **`parse(text, pos)`** de **`SimpleDateFormat`**, recebe um índice como parâmetro, que indica a próxima posição de **`text`** a ser processada;
- **`pos`** refere uma instância de **`ParsePosition`** que tem esta responsabilidade (*keep track*);

➔ É expectável em caso de sucesso o **`parse`** actualizar a instância **`ParsePosition`**.

4. *Side Effects…*

Outro tipo de efeitos secundários evitável é a apresentação de mensagens de erro através de **`System.out`**:

Exemplo:

```
public void addMessage(Message aMessage){
  if(newMessages.isFull())
    System.out.println("Sorry, no space available!"); // DON'T DO THAT
  ...
}
```

Um ambiente de execução como o Java disponibilizam um mecanismo próprio para notificação de erros.

As **excepções** oferecem uma boa flexibilidade em OOP permitindo que o utilizador decida qual a opção a tomar na ocorrência de um erro.

Dica:

- Minimize os efeitos secundários dos métodos que vão além da mutação do parâmetro implícito.

5. Lei de Deméter

Um método apenas deve usar

- Variáveis de instância;
- Paramêtros;
- Objectos construídos por si.

➔ Não deve operar sobre objectos globais ou objectos que façam parte do estado de outro objecto.

Exemplo:

O método **`findMailbox`** de **`MailSystem`** devolve um objecto **`MailBox`** que será posteriormente manipulado por uma **`Connection`** adicionando-lhe novas messagens.

➔ Modificar a interface de **`Mailbox`** obriga a actualizar **`MailSystem`** e **`Connection`**.

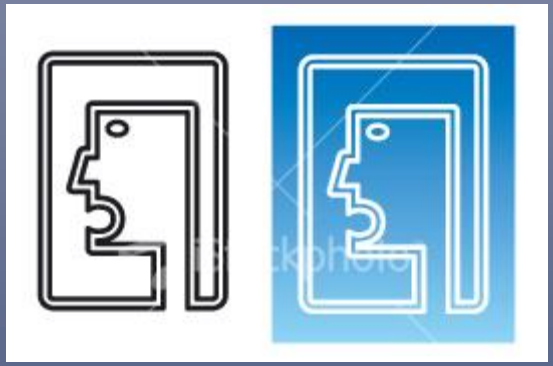
A lei de Deméter implica que uma classe:

- Não deve retornar referências para objectos que façam parte da sua implementação interna.
- Deve assumir a responsabilidade de interactuar com os seus objectos internos.

➔ Exemplo: **`MailSystem`** devia assumir o papel de adicionar e remover mensagens das suas **`Mailbox`**'s.

➔ Seguir a lei de Deméter permite reorganizar a estrutura interna das classes conservando as suas interfaces.

Desenho por contrato



Motivação

```
package aula09;

/**
 * Implementação de um contentor sequenci
 * custa de uma lista circular duplamente
 *
 * @see aula09.List
 * @author Fernando Miguel Carvalho - CCI
 **/

public class LinkedList extends
   implements Cloneable {
 /**
    * Obter o elemento, índice
    * @param idx Índice do elemen
    *    (entre 0 e size()-1)
    * @return  O elemento na posição
    */
  public Object element(int idx) {
    //...
  }
  //...
...
```

Class LinkedList

```
java.lang.Object
  └ aula09.AbstractList
      └ aula09.LinkedList
```

All Implemented Interfaces:

aula09.List, java.lang.Cloneable

```
public cl      inkedList
xtends      .AbstractList
le          neable
```

...sequencial à custa de uma lista circular duplamente ligada

...Object **element**(int idx)

...ter o e...o, índice idx, da sequência

...pecified by:

element in interface aula09.List

Parameters:

idx - - Índice do elemento pretendido (entre 0 e size()-1)

Returns:

O elemento na posição idx

A descrição puramente textual pode ser:

- **ambígua**
- **incompleta**
- **contraditória**

Programação por contrato…

```
public class MessageQueue{
  public Message remove() {...}
  public void add(Message aMessage) {...}
  public int size() {...}
  public boolean isFull() {...}
  public Message peek(){...}
}
```

O que deve acontecer se o "cliente" remover uma mensagem de uma fila vazia?

Duas hipóteses:

- O designer do componente declara no contracto esse comportamento como errado, não tomando responsabilidades nas consequências.
- O designer do componente decide tolerar esse potencial abuso e construir um mecanismo robusto, como seja retornar uma referência **`null`**.

Dica:

➔ ATENÇÃO: mecanismos para controlo de falhas, podem ter custos significativos na redução de desempenho do componente/aplicação.

Mais importante:

➔ Existir um **acordo formal** (**contracto**) entre a classe que fornece o serviço e o seu "cliente" (aplicação/ componente), sobre os requisitos da prestação do serviço.

Pré condições

Princípio:

- A terminologia de pré e pós condição serve para **formalizar** o **contracto** entre o serviço e o seu chamador.

- **Uma pré condição** é uma expressão lógica que tem de ser válida **antes** da chamada ao método.
- Se a pré condição não for válida o fornecedor do serviço não garante o bom comportamento, sendo expectável qualquer acção.

```
/**
     Remove message at head.
     @return the message that has been removed from the queue
     @precondition size() > 0
 */
public Message remove()
{ ... }
```

Nota:

- Por omissão o *javadoc* não reconhece as anotações `@precondition`. Para incluir esta anotação nos documentos é necessário usar a opção:

  `-tag precondition:cm:Precondition` (-tag <name>:<locations>:<header>)

Pré condições…

Regra:

- As pré condições devem permitir a validação pelo do utilizador do serviço.

Exemplo:

- Uma vez que **`elements`** é um campo privado, não está acessível ao utilizador:

```
/**
    Append a message at tail.
    @param aMessage the message to be appended
    @precondition size() < elements.length;
*/
public void add(Message aMessage)
{...}
```

```
!isFull();
```

Pós condições

Conjunto de expressões lógicas que têm de ser válidas **depois** da chamada ao método.

Neste caso é o prestador do serviço que tem a responsabilidade de cumprir a pós condição.

```
/**
  * Append a message at tail.
  * @param aMessage the message to be appended
  * @precondition !isFull();
  * @post size() > 0
  */
  public void add(Message aMessage) {...}
```

Dica:

- Não tem utilidade repetir o que já foi expresso na *tag* `@return`.

Programação defensiva: *assert*

Que acção tomar caso determinada pré-condição não se verifique?

1. Não fazer nada é legítimo. O chamador sabe que pode "sofrer" consequências de não cumprir as pré condições.
2. No entanto, pode facilitar a localização de erros, o método "alertar" a quebra de uma pré-condição.

   ➔ Mecanismo de **asserções**

Conceito:

- Uma asserção é um ponto de teste num programa.

➔ Caso a expressão lógica que compõe a asserção seja falsa é lançada uma excepção do tipo **`AssertionError`**.

```
/**
   @precondition size() > 0
 */
public Message remove() {
  assert count>0 : "violated precondition size()>0";
  ...
}
```

Dica:

- Asserções podem ser usadas para verificar, em tempo de execução, as **pré-condições**, **pós-condições** e **invariantes**.

Programação defensiva: *assert* ...

Porquê usar asserções em vez de retornar valores “inofensivos”?

Qual a desvantagem de fazer:

```
public Message remove() {
  if (count <= 0) return null;
  ...
}
```

➔ A utilização do valor **null** pode vir a produzir erros cuja a causa não é trivial de encontrar.

Porquê usar asserções em vez de lançar uma excepção?

Qual a desvantagem de fazer:

```
public Message remove() {
  if (count <= 0) throw new IllegalStateException();
  ...
}
```

➔ Desempenho.

Programação defensiva: *assert* ...

- As asserções podem ser activadas ou desactivadas.
  - ➔ Permitem a sua utilização para efeitos de *debug*.
  - ➔ Podem ser desactivadas para não penalizar o desempenho.

Exemplo:

```
java –enableassertions App
java –ea App
```

Contractos com excepções

As excepções fazem em muitos casos parte do contracto

```
/**
  * @param fileName name of the file to read from
  * @throws FileNotFoundException if the file
  * does not exist, is a directory, or cannot
  * be open for reading
  */
 public FileReader(String fileName){...}
```

Este construtor não tem qualquer pré-condição

- As classes cliente podem depender da promessa de ser lançado uma excepção.

- Será correcto/possível exigir que o ficheiro exista?

R: entre o momento da verificação de existência e a instanciação de **`FileReader`** o ficheiro pode ter sido adquirido por outro processo.

Invariante

Princípio:

- Se todos os **construtores** iniciarem os objectos de uma classe com **estados válidos** e todos os **métodos modificadores** preservarem os objectos num estado válido ➔ **então nunca existirão objectos inválidos**.
- **Excepto durante a execução de um método.**

Definição:

- **Invariante** de uma classe é uma propriedade válida para qualquer objecto dessa classe num **estado válido** (ou seja, após a construção e execução de qualquer método).

Exemplo:

- Para cada nó (`p`) de uma lista duplamente ligada, tem de se verificar:
  - `p.prev.next = p`
  - `p.next.prev = p`

Invariante…

Um invariante da implementação de **`MessageQueue`** é:

`head >= 0 && head <= elements.length`

```
/**
 * A first-in, first-out bounded collection of messages.
 * @invariant wellformed();
 */
public class MessageQueue {
   /**
    * Invariant of a bounded collection
    */
   protected boolean wellformed(){
     return head >= 0 && head < elements.length;
   }
   /**
    * Remove message at head.
    * @return the message that has been removed from the queue
    * @precondition size() > 0
    */
   public Message remove() {
      assert count > 0 : "violated precondition size()>0";
       Message r = elements[head];
       head = (head + 1) % elements.length;
       count--;
       assert wellformed() : "violated MessageQueue invariant.";
       return r;
   }
```

Testes Unitários

Testes Unitários

Verificam o correcto funcionamento de uma unidade do sistema (classe no contexto da OOP)

Os testes podem ser escritos com base em:

- Descrição funcional;
- Pré, pós condições e invariantes;
- Aspectos de implementação.

Ter um conjunto completo de testes aumenta a confiança no código produzido.

JUnit

JUnit é uma ferramenta popular para a realização de testes unitários

- Cada conjunto de testes é designado *test suite* e representado numa classe que estende de **TestCase.**

```
import junit.framwork.*;
public class DayTest extends TestCase
{
   public void testAdd() { ... }
   public void testDaysBetween() { ... }
   ...
}
```

```
public void testAdd()
{
   Day d1 = new Day(1970, 1, 1);
   Day d2 = d1.addDays(20);
   assertTrue(d2.daysFrom(d1) == 20);
}
```

Cobertura de testes unitários

Esta técnica permite quantificar o grau de cobertura dos testes realizados

As métricas usadas dependem da ferramenta

- Na ferramenta EMMA
  - Classes
  - Métodos
  - Linhas de código
  - Blocos básicos (conjunto de instruções sem *jumps* ou *calls*)

EclEmma 1.3.1 Java Code Coverage for Eclipse:

http://www.eclemma.org/

ANEXO:
Notas spbre polimorfismo e *dynamic binding*

Override – Redefinição de métodos

As subclasses podem redefinir (*overriding*) os métodos de instância (não estáticos), dando outra implementação.

A redefinição tem que ter:

- A mesma assinatura (nome e parâmetros);
- O mesmo tipo de retorno;
- Uma lista igual ou mais restrita de excepções;

Override != Overload

A **redefinição** (herança) e a **sobrecarga** (na mesma classe) são mecanismos totalmente distintos.

- A resolução da chamada a **métodos em sobrecarga** é feita na **compilação**;
- A resolução da chamada a **métodos redefinidos** é feita em ***runtime*** (***dinamic binding***)
- Redefinir com mesma assinatura e tipo de retorno diferentes gera **erro de compilação**;
- Redefinir métodos estáticos ou campos nas subclasses apenas **esconde** (***hiding***) a definição da superclasse.

Override – Redefinição de métodos...

Protecção contra redefinição:

- Os métodos classificados com **`final`** não podem ser redefinidos nas subclasses;
- As chamadas são optimizadas pela JVM.

Chamar o método da superclasse:

- Na subclasse a implementação de um método na superclasse pode ser invocada com **`super.m().`**

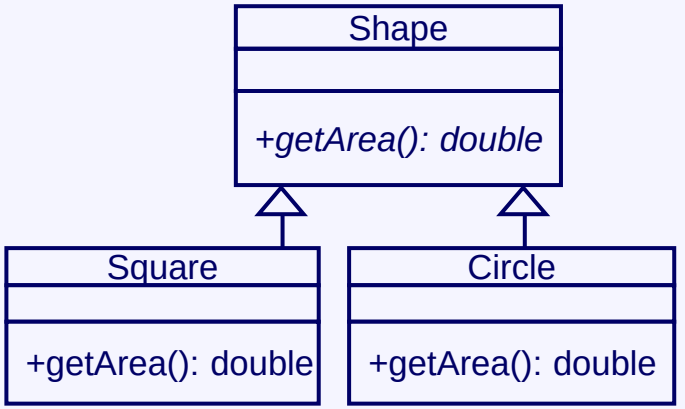
Restrição na herança:

- Se na subclasse não faz sentido a funcionalidade de determinado método, então deve ser redefinido para lançar excepção (ex: **`MethodNotSupported`**).

Chamada polimórfica

As chamadas aos métodos de instância (virtuais) são decididas em tempo de execução, dependendo do tipo de objecto referenciado (***dynamic binding***)

```
double totalArea(Shape[] sa) {
  double total = 0.0;
  for(int i=0; i < sa.lenght ; ++i)
    total += sa[i].getArea();
}
```



A chamada polimórfica **`ref.m()`** é processada assim:

Passo 1: **`cl`** ← classe do objecto referenciado por **`ref`**

Passo 2: Se **`m()`** implementado por **`cl`**
- ➔**então** chamar implementação de **`m()`** em **`cl`**
- ➔**senão** **`cl`** ← superclasse de **`cl`**, e repetir passo 2