Computação Distribuída

Introdução ao Java

António Rui Borges

Sumário

- *Java*
  - *Nota histórica*
  - *Características principais*
- *Técnicas gerais de descrição da solução de um problema*
- *Metodologias de programação*
  - *Programação procedimental ou imperativa*
  - *Programação modular*
  - *Programação orientada por objectos*
  - *Programação concorrente*
  - *Programação distribuída*

Nota histórica - 1

O *Java*, designado originalmente de *Oak*, foi concebido em meados da década de 90 por um grupo de trabalho da Sun Microsystems, liderado por James Gosling, no âmbito do projecto *Green*

- a sua estrutura sintáctica baseia-se nas linguagens de programação C e C++;
- o objectivo inicial era criar uma linguagem que servisse de suporte à construção de ambientes de controlo e de comunicação de sistemas embebidos ao nível de equipamento electrónico de consumo;
- a sua popularidade surge, porém, no contexto da *internet*: primeiro, com os *applets*, pequenos programas escritos em Java que podiam ser executados no interior de um *browser*, e, mais recentemente, com os *serviços web*;
- numa perspectiva mais genérica, tem vindo a posicionar-se ao longo da última década como a linguagem de eleição para o desenvolvimento de aplicações distribuídas, fornecendo um ambiente integrado e uniforme para comunicação entre processos sobre redes de computadores (Java in action).

Nota histórica - 2

Os objectivos iniciais do Java, enquanto linguagem de programação, incluíam

- *ser totalmente independente da plataforma hardware subjacente*, o que conduziu a que fosse concebida simultaneamente como uma *linguagem* e um *ambiente de execução*: os programas podem, por conseguinte, ser transferidos e executados dinamicamente em qualquer nó da malha de processamento;
- *ser inerentemente robusta*, minimizando os erros de programação: é, por isso, uma linguagem fortemente semântica em que algumas características do C++, como os mecanismos de herança múltipla e de *overloading* de operadores, foram eliminadas por serem julgadas fontes potenciais de erro; foi também introduzido um mecanismo de gestão implícita da memória dinâmica (*garbage collection*) e impedida qualquer possibilidade de definição de estruturas de dados fora do âmbito do construtor `class`; nesta perspectiva, há quem diga que ***Java is C++ done right***;
- *promover a segurança* em ambientes que estão continuamente a trocar informação e a partilhar código: os *ponteiros* foram por esta razão eliminados, procurando-se impedir que programas não autorizados acedam a estruturas de dados residentes em memória.

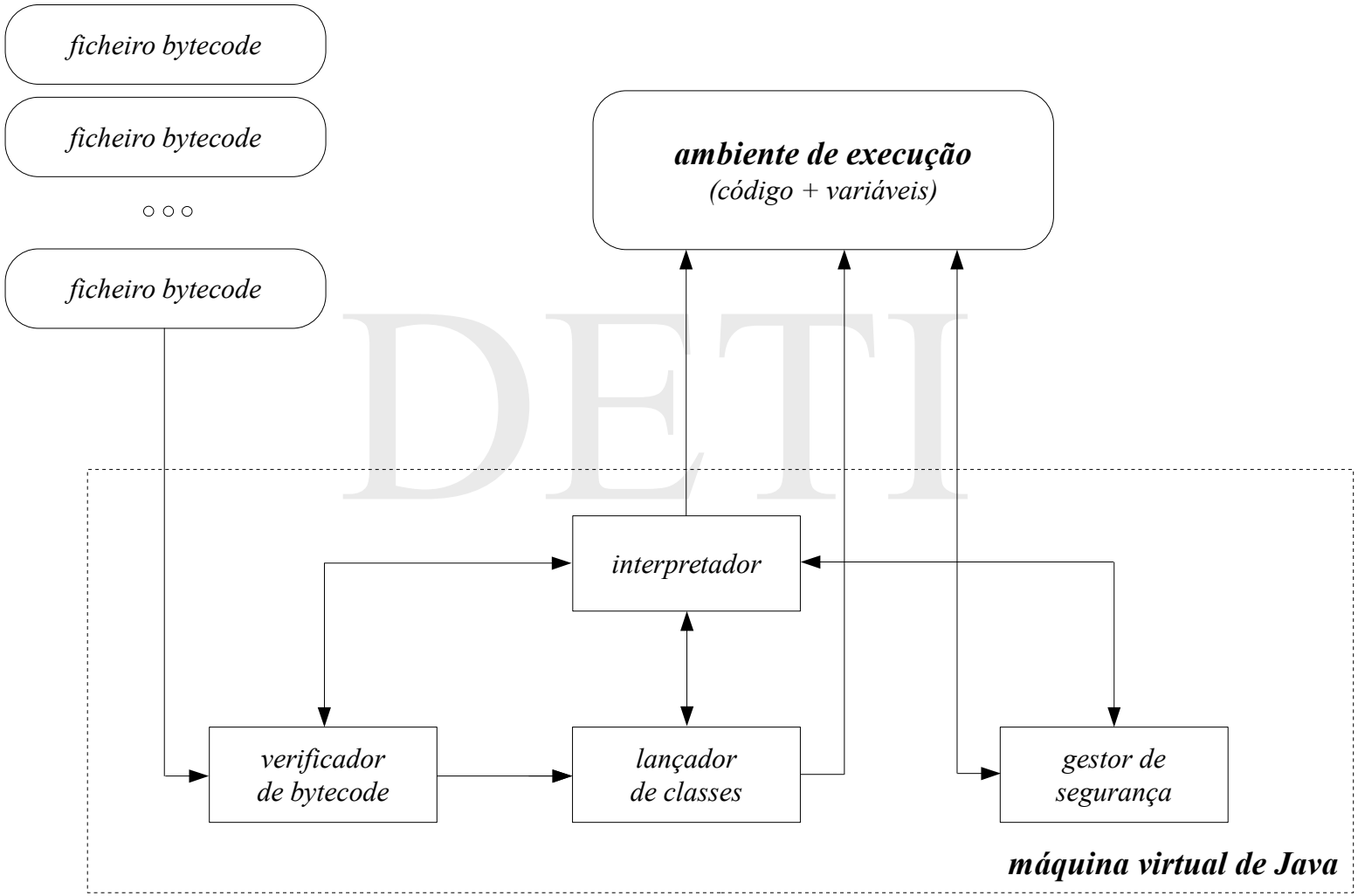
Características principais - 1

- linguagem organizada segundo os princípios do *paradigma orientado por objectos*
  - número mínimo de construções semânticas, potenciando uma descrição compacta e inerentemente segura;
  - mecanismo de herança estritamente linear;
  - introdução do construtor `interface` para separação precisa entre especificação e implementação e como elemento de ligação entre variáveis de tipos de dados disjuntos;
- inclusão de mecanismos de concorrência
  - apoio expresso à construção de ambientes *multithreading*;
  - transformação implícita de *objectos* em *monitores* [de tipo Lampson / Redell] por mera sincronização dos seus métodos públicos;
- suporte à programação distribuída através da disponibilização de bibliotecas que implementam os dois grandes paradigmas de comunicação
  - passagem de mensagens: *sockets*;
  - variáveis partilhadas: *remote method of invocation*;

Características principais - 2

- o ambiente de execução em Java, designado de *máquina virtual de Java*, constitui uma camada *middleware* que torna as aplicações totalmente independentes da plataforma hardware e do sistema de operação onde estão residentes, implementando um modelo de segurança de três camadas para proteger o sistema contra código não confiável
  - o *verificador de bytecode* analisa o bytecode presente para execução e certifica-se que as regras básicas da gramática de Java são obedecidas;
  - o *lançador de classes* fornece os tipos de dados Java ao interpretador de código;
  - o *gestor de segurança* lida com os problemas que põem potencialmente em causa a segurança do sistema ao nível da aplicação, controlando as condições de acesso, por parte de um programa em execução, ao sistema de ficheiros, à rede, a processos externos e ao sistema de janelas;

Características principais - 3



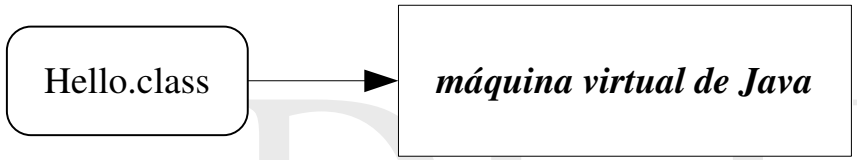
Características principais - 4

- suporte à internacionalização das aplicações
  - substituição do código ASCII pelo Unicode na representação interna de caracteres para permitir descrever alfabetos e símbolos ideográficos das diferentes línguas mundiais;
  - separação da informação de localização do código executável, o que permite, conjuntamente com o armazenamento dos elementos textuais fora do código fonte e o consequente acesso dinâmico, que as aplicações se conformem à língua e a outros traços culturais específicos do utilizador final;
- inclusão de ferramentas de apoio à produção de documentação
  - o recurso aos *comentários de documentação* nos ficheiros fonte para descrever os tipos de dados, as suas estruturas de dados internas e os seus métodos de acesso, possibilita a produção automática de documentação em formato *html* por aplicação de `javadoc`.

Exemplo simples - 1

```
/**
 *    Descrição geral:
 *      neste caso será, por exemplo, o programa imprime a frase
 *      "Olá < nome de pessoa >, como estás?" no écran do monitor vídeo.
 *
 *    @author António Rui Borges
 *    @version 1.0 – 5/8/2004
 */

 public class Hello

 {
   /**
    *    Programa principal,
    *      é implementado pelo método main do tipo de dados, qualquer aplicação
    *      tem que conter um método estático de nome main no seu tipo de arranque.
    *
    *    @param args nome da pessoa a saúdar
    */

    public static void main (String [] args)

    {
       System.out.println ("Olá " + args[0] + ", como estás?");
    }
 }
```

Exemplo simples - 2



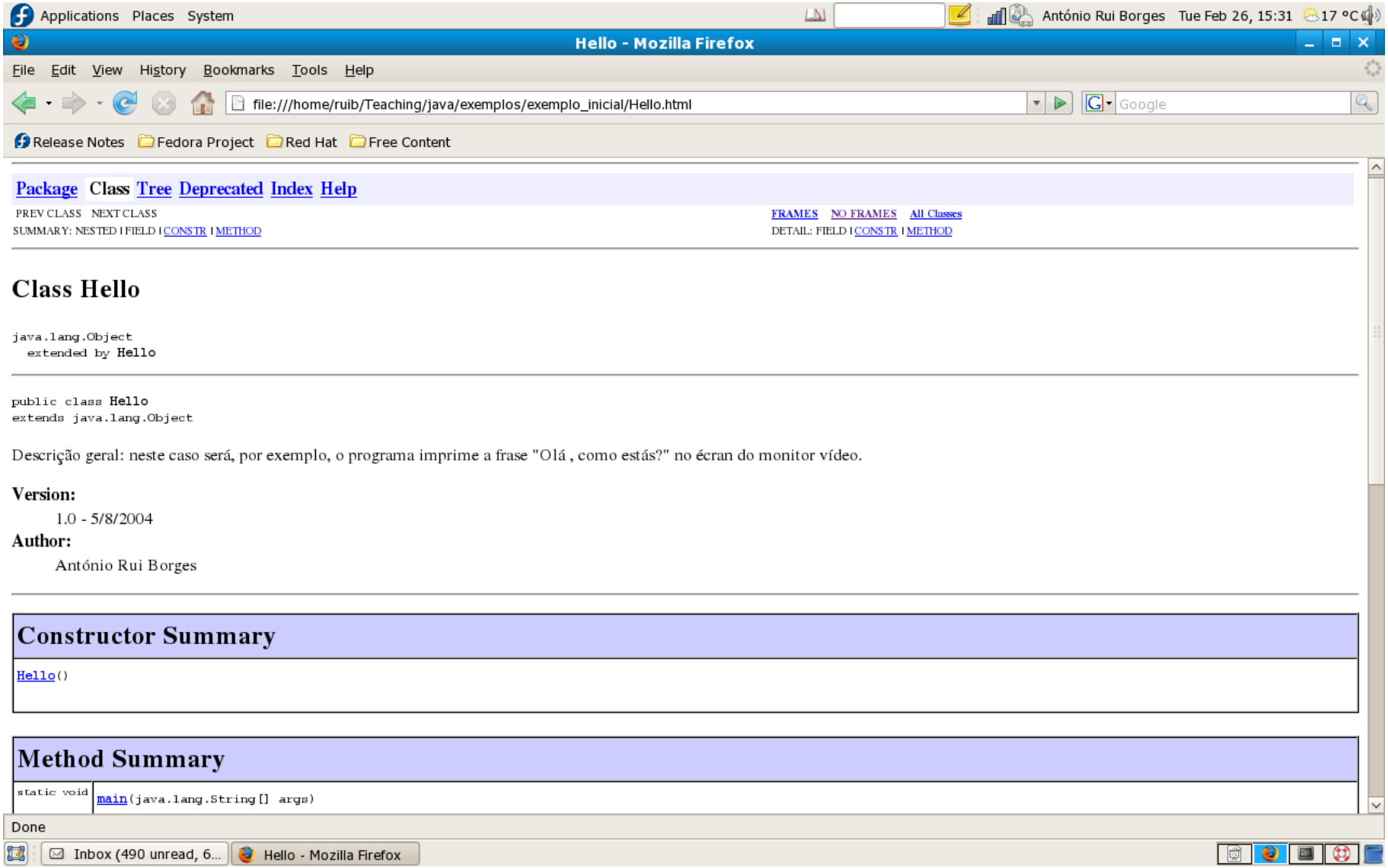

```
[ruib@ruib-laptop exemplo_inicial]$ javac Hello.java
[ruib@ruib-laptop exemplo_inicial]$ java Hello Pedro

Olá Pedro, como estás?

[ruib@ruib-laptop exemplo_inicial]$ ll
total 60
-rw-r--r-- 1 ruib ruib  598 2008-02-26 15:04 Hello.class
lrwxrwxrwx 1 ruib ruib   26 2008-02-26 15:10 Hello.html -> Hello_html_desc/Hello.html
drwxrwxr-x 3 ruib ruib 4096 2008-02-26 15:10 Hello_html_desc
-rw-r--r-- 1 ruib ruib  657 2008-02-26 15:03 Hello.java
-rwxr-xr-x 1 ruib ruib  131 2008-02-26 15:10 Hello.javadoc
-rw-r--r-- 1 ruib ruib  378 2008-02-26 15:07 HelloWorldApp.class
-rw-r--r-- 1 ruib ruib  149 2008-02-26 15:09 HelloWorldApp.html
-rw-r--r-- 1 ruib ruib  349 2008-02-26 15:07 HelloWorldApp.java
[ruib@ruib-laptop2 exemplo_inicial]$
```

Exemplo simples – 3

Produção de documentação

```
[ruib@ruib-laptop2 exemplo_inicial]$ cat Hello.javadoc
javadoc -d Hello_html_desc -author -version -breakiterator -charset "utf 8" Hello.java
ln -s Hello_html_desc/Hello.html Hello.html

[ruib@ruib-laptop2 exemplo_inicial]$ Hello.javadoc
Creating destination directory: "Hello_html_desc/"
Loading source file Hello.java...
Constructing Javadoc information...
Standard Doclet version 1.6.0_04
Building tree for all the packages and classes...
Generating Hello_html_desc/Hello.html...
Generating Hello_html_desc/package-frame.html...
Generating Hello_html_desc/package-summary.html...
Generating Hello_html_desc/package-tree.html...
Generating Hello_html_desc/constant-values.html...
Building index for all the packages and classes...
Generating Hello_html_desc/overview-tree.html...
Generating Hello_html_desc/index-all.html...
Generating Hello_html_desc/deprecated-list.html...
Building index for all classes...
Generating Hello_html_desc/allclasses-frame.html...
Generating Hello_html_desc/allclasses-noframe.html...
Generating Hello_html_desc/index.html...
Generating Hello_html_desc/help-doc.html...
Generating Hello_html_desc/stylesheet.css...
[ruib@ruib-laptop2 exemplo_inicial]$
```

Exemplo simples – 4



HelloWorld applet – 1

HelloWorldApp.html

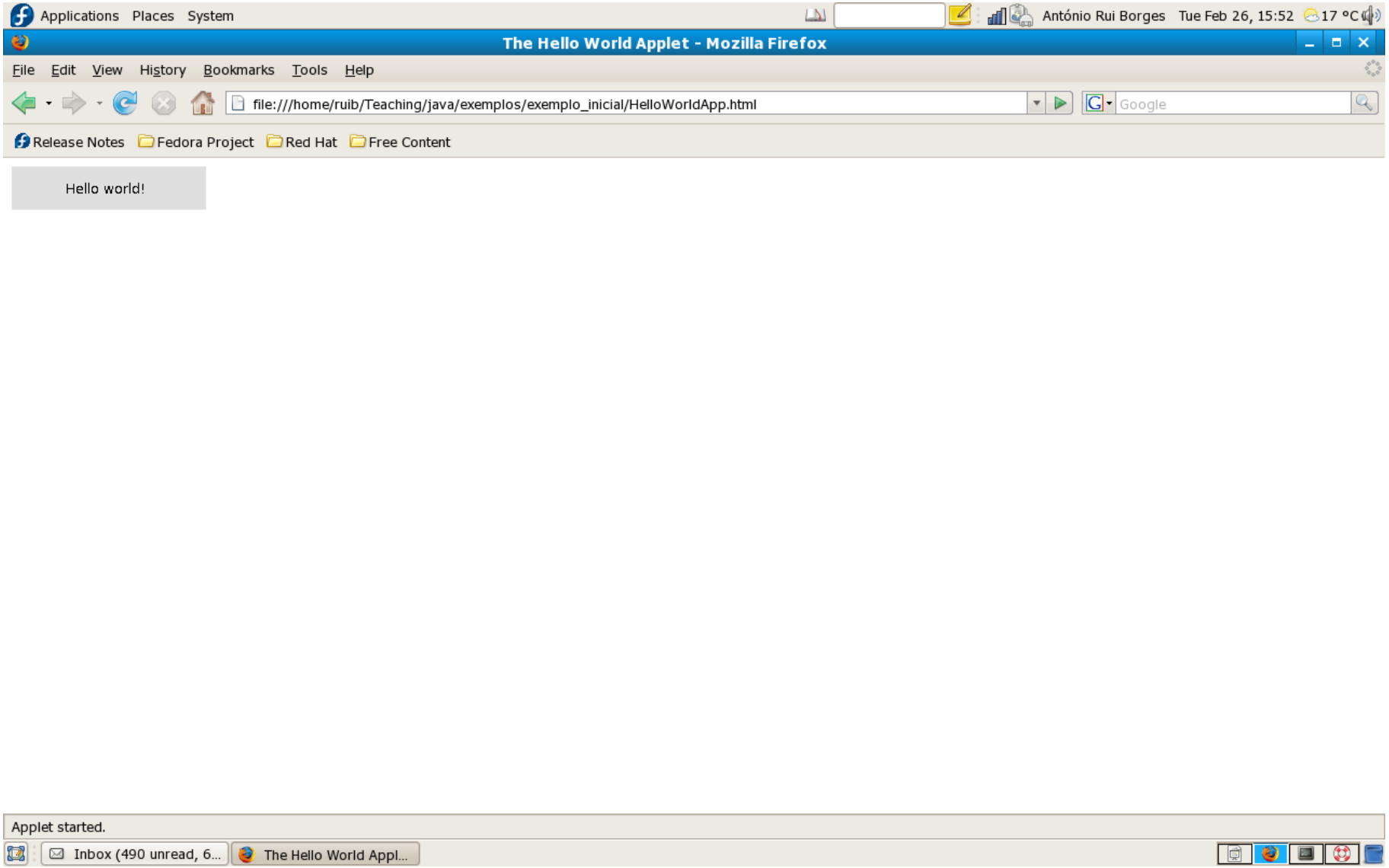
```
<HTML>
<HEAD>
<TITLE>The Hello World Applet</TITLE>
</HEAD>
<BODY>
<APPLET CODE="HelloWorldApp.class" WIDTH=180 HEIGHT=40>
</APPLET>
</BODY>
</HTML>
```

HelloWorld applet – 2

HelloWorldApp.java

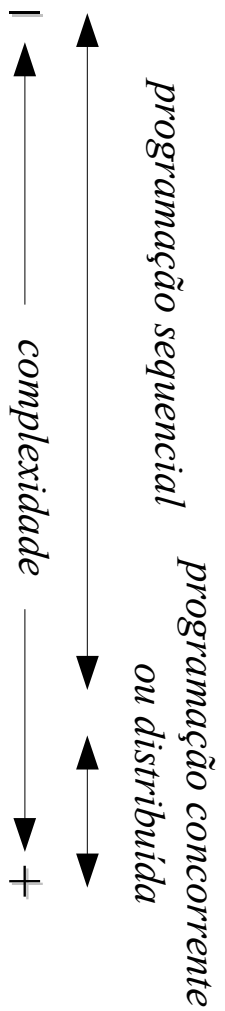
```
/**
 *  Descrição geral:
 *      applet que imprime a saudação "Hello World!" numa janela de um browser onde
 *      a execução de Java está activada.
 */

import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorldApp extends Applet
{
   public void init ()
   {
   }

   public void paint(Graphics g)
   {
     g.drawString("Hello world!", 50, 25);
   }
}
```

HelloWorld applet – 3



Técnicas gerais de descrição da solução de um problema

- *decomposição hierárquica*
  - recurso a uma metalinguagem de descrição, preferencialmente semelhante à linguagem de programação que vai ser usada;
  - encapsulamento de informação através de
    - construção de tipos de dados adequados às características do problema;
    - definição de novas operações no âmbito da linguagem;
  - estabelecimento estrito das dependências de informação;
- *decomposição em estruturas autónomas interactivas*
  - especificação precisa de um mecanismo de interacção;
  - separação clara entre *interface* e *implementação*;
    - abstracção dos dados e sua protecção;
    - virtualização das operações de acesso;
- *decomposição em entidades autónomas interactivas*
  - especificação precisa de um modelo de comunicação;
  - definição e implementação de mecanismos de sincronização.



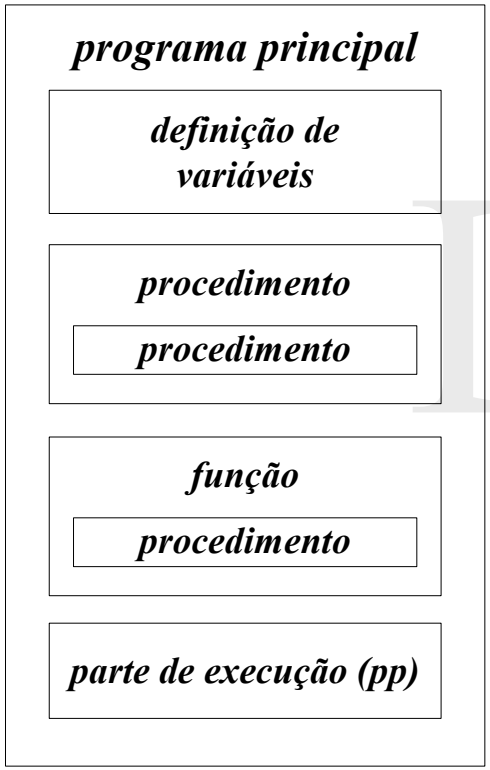
Metodologias de programação - 1

Programação procedimental ou imperativa

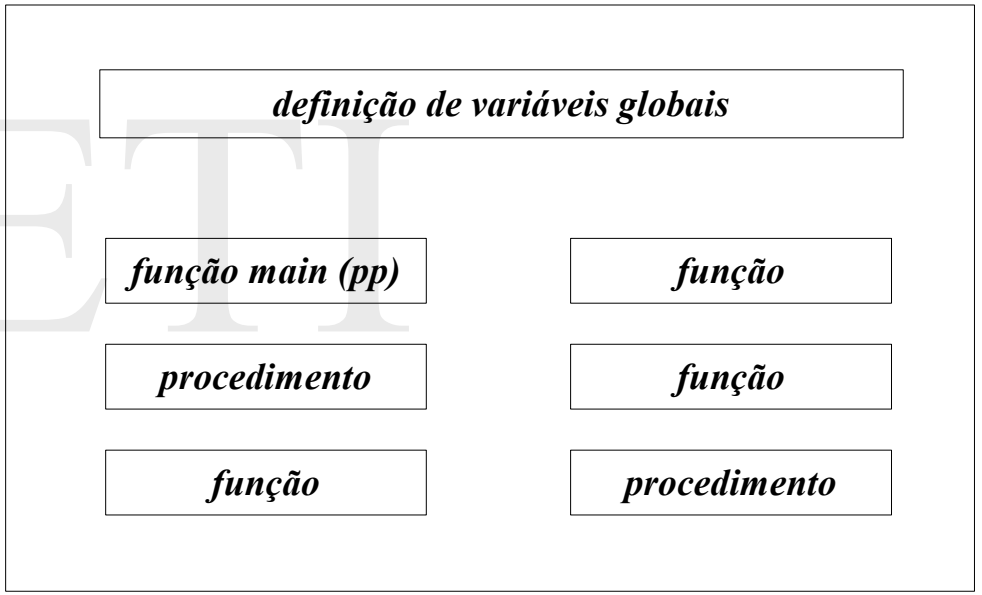
- o ênfase é colocado numa implementação mais ou menos fidedigna da decomposição hierárquica da solução;
- a complexidade é controlada pelo uso intensivo
  - *de funções* e *procedimentos*, como meio de descrever as operações a diferentes níveis de abstracção;
  - *de construtores de tipos de dados*, para organizar a informação associada do modo mais adequado às características do problema;
- o espaço de variáveis é *concentrado*
  - toda a informação relevante à solução do problema é definida tipicamente ao nível do programa principal, ou é global; as variáveis locais às restantes funções e procedimentos representam armazenamento meramente temporário (só existem durante a sua invocação);
  - o acesso das diferentes operações aos dados é concebido numa obediência estrita ao princípio "*daquilo que elas precisam de saber*", usando um modelo de comunicação baseado na passagem de parâmetros;

Metodologias de programação - 2

Programação procedimental ou imperativa



organização hierárquica do ficheiro fonte típica do Pascal



organização horizontal do ficheiro fonte típica da linguagem C (estrutura em mar de funções)

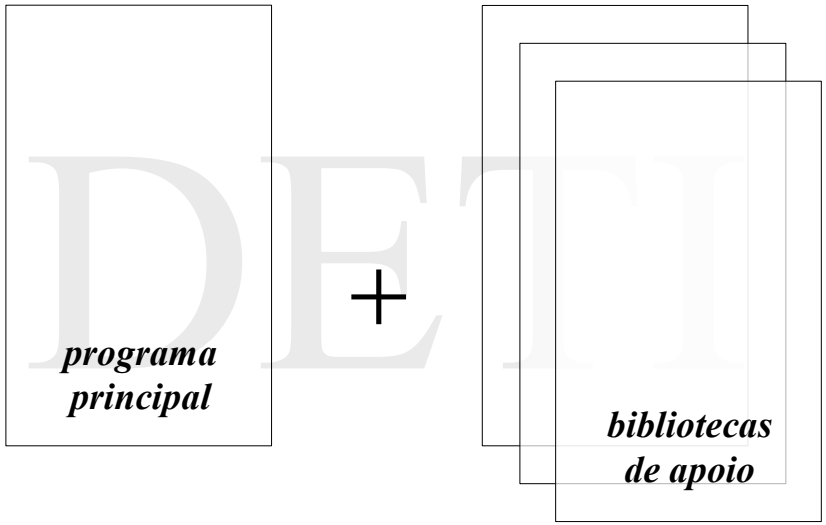
Metodologias de programação - 3

Programação procedimental ou imperativa

- uma questão que rapidamente se coloca é o da *reutilização de código*;
- diferentes operações, implementadas por funções e procedimentos no âmbito de uma aplicação específica, são muitas vezes úteis para outras aplicações;
- a sua inclusão automática no código destas novas aplicações faz-se dividindo o ficheiro fonte em múltiplos ficheiros, cada um deles objecto de compilação separada e reunidos num ficheiro executável único na fase de linkagem;
- assim, o código de uma aplicação particular passa a estar disperso por
  - um ficheiro fonte que contém o programa principal;
  - múltiplos ficheiros fonte que contêm diferentes funcionalidades, expressas por grupos de funções e procedimentos que foram previamente desenvolvidos e que vão ser usados no contexto da aplicação presente, as chamadas *bibliotecas de apoio*;

Metodologias de programação - 4

Programação procedimental ou imperativa



- neste tipo de organização, torna-se necessário inserir nos ficheiros fonte parcelares, sempre que o código aí existente referenciar operações externas, o nome do ficheiro onde elas estão descritas (*ficheiro de interface*) para garantir consistência nas compilações parcelares que terão que ter lugar.

Metodologias de programação - 5

Programação modular

- à medida que a complexidade da descrição da solução aumenta, a gestão centralizada do espaço de variáveis torna-se progressivamente mais difícil e o ênfase desloca-se da concepção das operações para uma organização mais fina da informação;
- o espaço de variáveis torna-se *distribuído* e surge o conceito de *módulo* para efectuar a sua gestão
  - um *módulo* é, por definição, uma estrutura autónoma, implementada num ficheiro fonte separado, que encapsula uma funcionalidade bem definida, incluindo genericamente um espaço de variáveis próprio e um conjunto de operações que as manipulam;
  - o espaço de variáveis é em princípio *interno*, o que significa que o acesso a ele é realizado apenas pelas operações especificadas, as chamadas *primitivas de acesso*;
  - note-se que, quando o espaço próprio de variáveis não tem significado externo, um *módulo* constitui aquilo que se designou por *biblioteca* no contexto do modelo anterior;

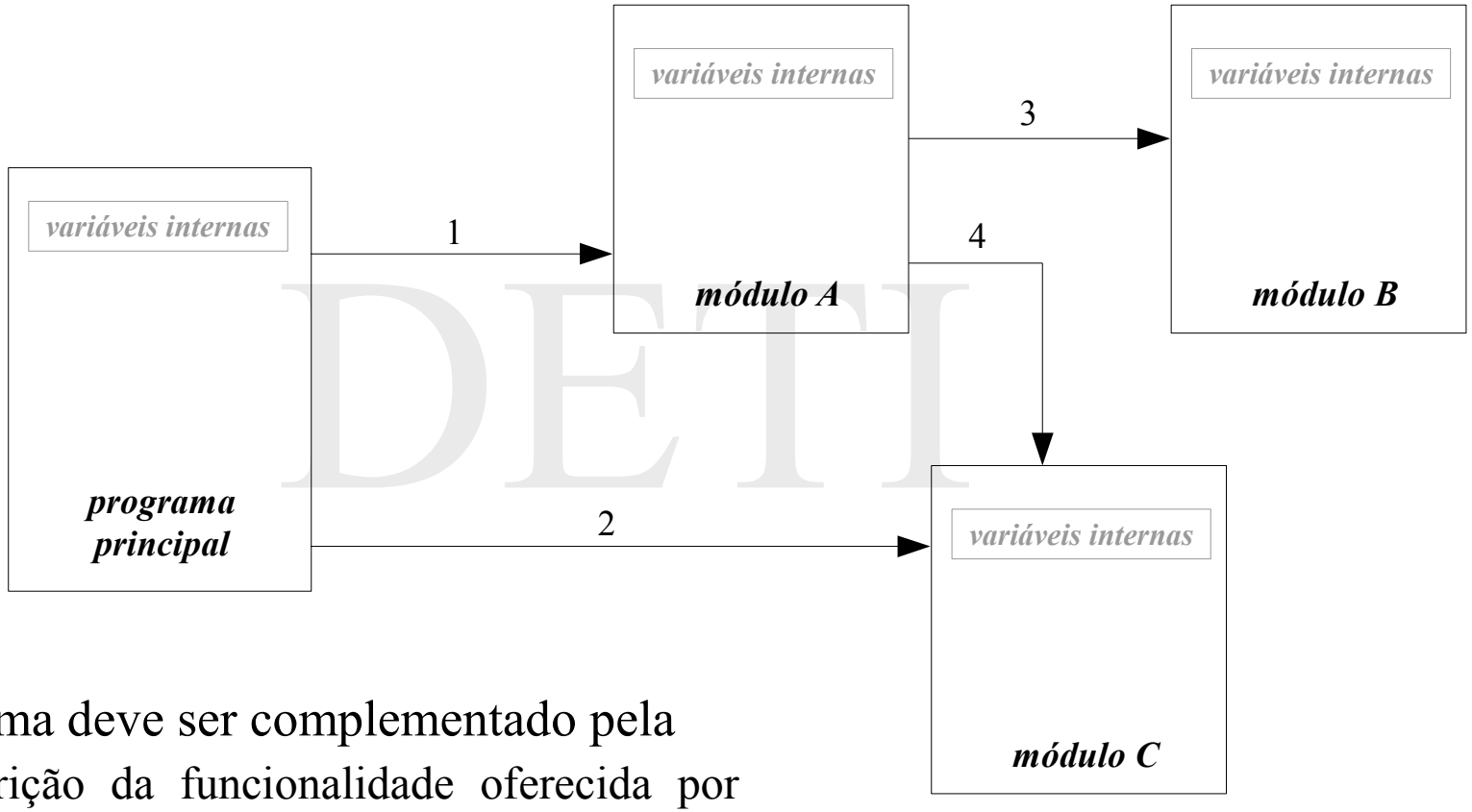
Metodologias de programação - 6

Programação modular

- a aplicação de uma abordagem de carácter funcional à decomposição da solução origina a especificação de um conjunto de estruturas autónomas que interactuam e permite lidar com problemas mais complexos, criando uma divisão clara de tarefas e promovendo o trabalho cooperativo
  - a descrição passa a ser feita pela especificação precisa do conjunto de estruturas que interactuam, sendo estabelecido para cada uma delas o seu papel na interacção e o modo de acesso à operacionalidade disponibilizada;
  - a separação clara entre a especificação de acesso, o *interface*, e a implementação respectiva possibilita o seu uso na construção de outros módulos, mesmo antes do módulo original estar concluído e validado;

Metodologias de programação - 7

Programação modular



- o diagrama deve ser complementado pela
  - descrição da funcionalidade oferecida por cada um dos módulos;
  - indicação das operações envolvidas em cada interacção.

Metodologias de programação - 8

Programação modular

- a aplicação consistente de uma estratégia de tipo *dividir para reinar*, subjacente a qualquer metodologia de descrição e, particularmente, quando se faz uma decomposição funcional da solução, potencia ainda o recurso a práticas de desenvolvimento de software *robusto*, corporizadas por
  - *desenho para o teste*: sendo o módulo uma estrutura autónoma, a sua funcionalidade pode e deve ser validada de acordo com as especificações previamente estabelecidas antes da sua incorporação na aplicação de que é parte;
  - *programação por contrato*: o programador responsável pela implementação do módulo pode estabelecer mecanismos de garantia em *runtime* que só é disponibilizada a operacionalidade descrita
    - todas as primitivas de acesso devolvem uma informação de *status* de operação: *operação executada com sucesso* ou *ocorrência de erro*, neste caso com indicação do erro ocorrido;
    - a consistência da estrutura de dados interna deve ser garantida antes da realização da primeira operação (*pré-invariante*);
    - aquando da realização de uma operação, a gama de valores de cada variável da lista de parâmetros de entrada deve ser validada antes da sua execução;
    - a consistência da estrutura de dados interna deve ser garantida após a realização da operação actual (*pós-invariante*).

Metodologias de programação - 9

Programação orientada por objectos

- a implementação modular de uma decomposição funcional pode tornar-se bastante inflexível em termos de *reutilização de código* quando se verifica que uma nova aplicação exige relativamente a um módulo previamente construído
  - a sua instanciação múltipla;
  - a introdução de pequenas alterações;
- a atitude tomada pelo programador neste contexto é inevitavelmente a adequação dos módulos pré-existentes à situação presente, conduzindo a uma pulverização progressiva do seu reportório de funcionalidades, com o consequente aumento de complexidade e perda de eficiência na gestão do software de apoio ao desenvolvimento de aplicações futuras;
- uma solução possível para este problema consiste no recurso a uma metodologia de aumento da abstracção;

Metodologias de programação - 10

Programação orientada por objectos

- a primeira situação apontada pode ser resolvida por uma *extensão* ao conceito de *tipo de dados*;
- com efeito, se for possível associar a cada instanciação do módulo uma variável diferente, a ambiguidade desaparece e a instanciação múltipla torna-se trivial;
- neste sentido, a noção de *tipo de dados* como um conjunto de regras que permitem armazenar em memória valores com determinadas características, é agora estendida de modo a contemplar não só o armazenamento de grupos de valores, mas também a incluir as operações que podem ser realizadas sobre eles;
- embora impropriamente, estes tipos de dados são conhecidos pelo nome de *tipos de dados abstractos*; *tipos de referência*, ou *tipos de dados definidos pelo utilizador* são designações mais correctas que lhes são por vezes também atribuídas;

Metodologias de programação - 11

Programação orientada por objectos

- as linguagens de programação que os suportam, incluem um construtor, cujo nome tradicional é `class`, que permite no fundo associar a definição de uma estrutura de tipo módulo a um identificador;
- a partir daí, torna-se possível declarar variáveis desse tipo;
- um pormenor importante é que a declaração simples de variáveis deste tipo não reserva espaço em memória para o seu armazenamento, reserva apenas espaço para *armazenamento de um ponteiro*;
- a reserva de espaço propriamente dita é ditada pela sua *instanciação* em *runtime*; esta operação consiste na reserva de espaço na zona de definição dinâmica e consequente inicialização da estrutura de dados interna; de igual modo, quando a variável já não é necessária o espaço de armazenamento em memória dinâmica deve ser libertado;
- os valores instanciados são designados de *objectos*, donde advém o nome do paradigma;

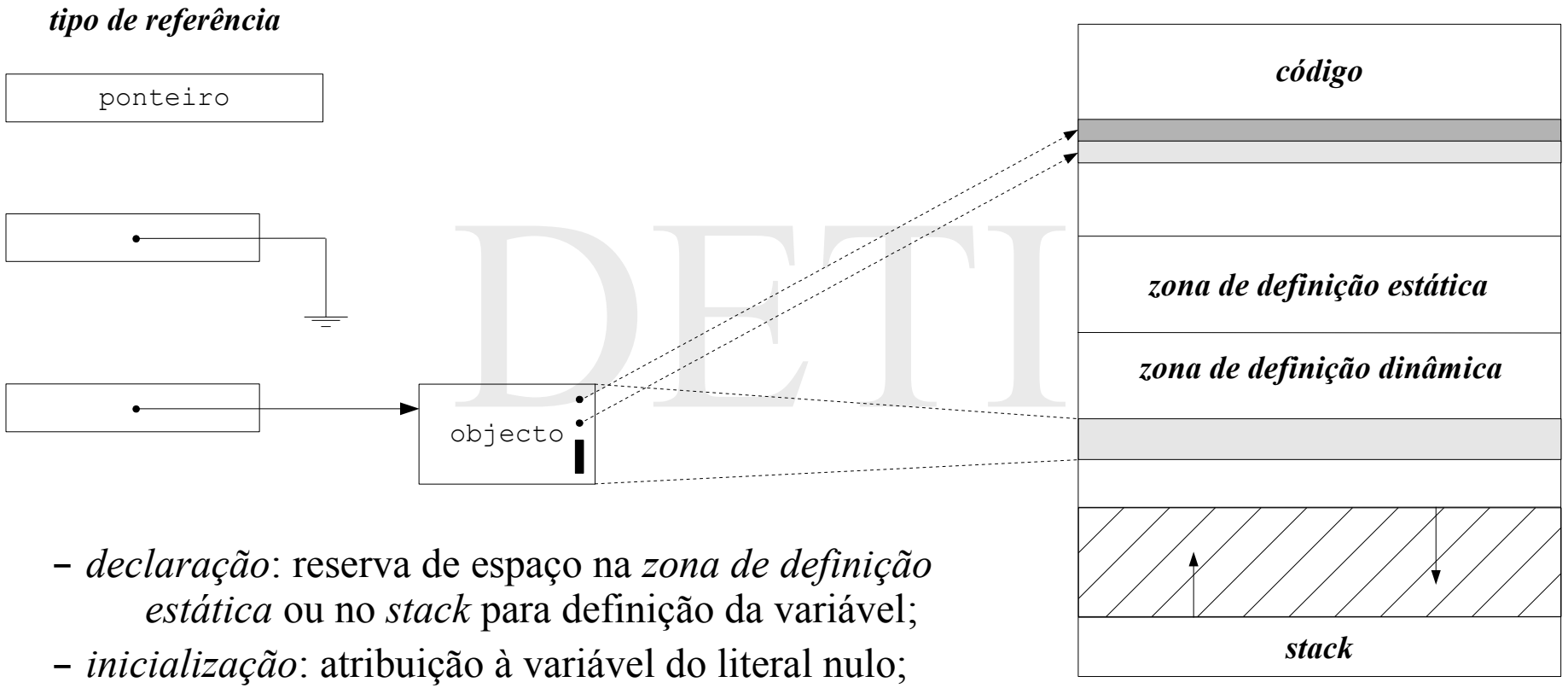
Metodologias de programação - 12

Programação orientada por objectos

- na terminologia *orientada por objectos*, designa-se de *campos* as variáveis da estrutura de dados interna de um tipo de referência, e de *métodos* as primitivas de acesso; diz-se então que um *objecto* é determinado pelo seu *estado* (conjunto dos valores actuais dos diferentes campos) e pelo seu *comportamento* (leque de métodos de acesso providenciados);
- as linguagens de programação que suportam este paradigma, permitem normalmente a *abstracção de dados* na especificação da estrutura de dados interna, originando tipos de dados genéricos com uma gama de aplicação mais alargada
  - em vez de se criar um tipo de dados que represente uma memória de tipo stack para armazenamento de caracteres, por exemplo, pode criar-se um tipo de dados que represente uma memória de tipo stack que armazene qualquer tipo de valores;
- além disso, pode ainda garantir-se, através da sua *parametrização*, que só um tipo particular de valores é armazenado em *runtime*
  - usando o mesmo exemplo, cria-se um tipo de dados que represente uma memória de tipo stack para armazenamento de valores de um tipo genérico T, não especificado, e adia-se para a declaração e instanciação de variáveis desse tipo a indicação do que o tipo T realmente significa;

Metodologias de programação - 13

Programação orientada por objectos



- *declaração*: reserva de espaço na *zona de definição estática* ou no *stack* para definição da variável;
- *inicialização*: atribuição à variável do literal nulo;
- *instanciação*: reserva de espaço na *zona de definição dinâmica* para criação de um *objecto* (feita em *runtime*);

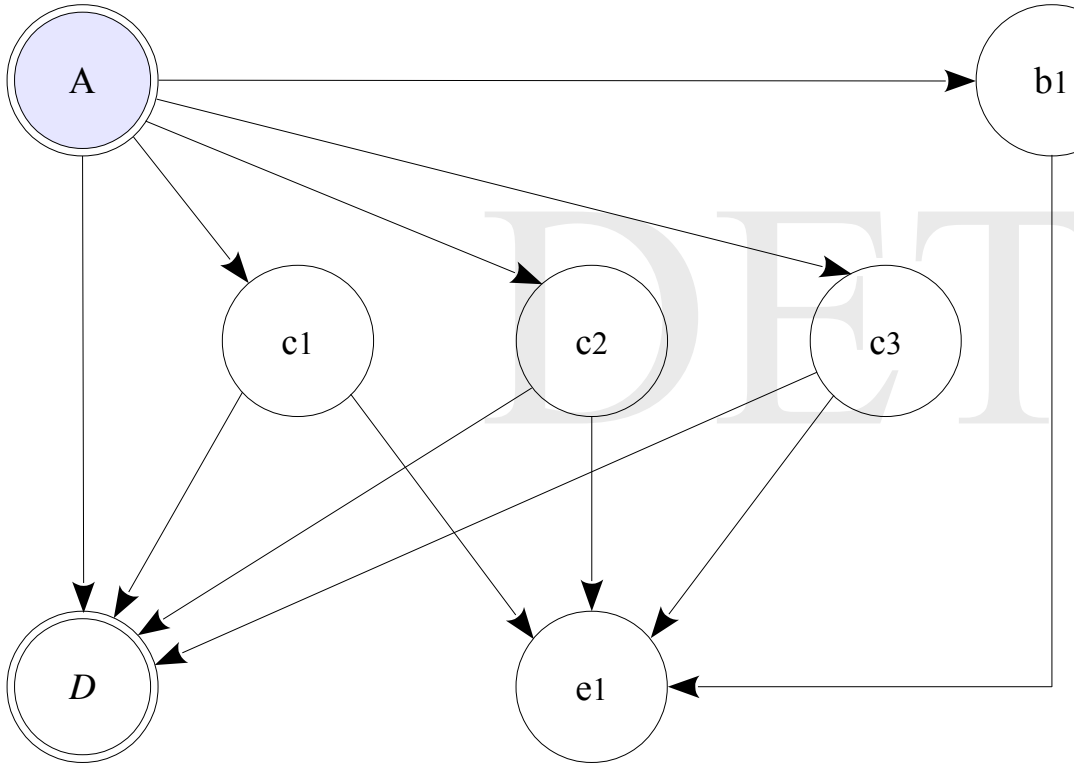
Metodologias de programação - 14

Programação orientada por objectos

- um programa organizado segundo esta metodologia é formado por múltiplos ficheiros fonte, cada um descrevendo um tipo de dados particular;
- tal como para a programação modular, a descrição da interacção é feita por um diagrama que inclui agora os tipos de dados não instanciados e os tipos de dados instanciados, os primeiros nomeados pelos identificadores de tipo e os segundos pelos identificadores das respectivas variáveis; o modo como eles interagem é especificado através de grafos orientados que exprimem o tipo de acesso materializado em cada caso; o diagrama deve ser complementado com uma descrição da funcionalidade disponibilizada por cada tipo e pela listagem das operações envolvidas;
- genericamente, os *tipos de dados não instanciados* correspondem ao programa principal e a grupos de operações genéricas enquadradas em *bibliotecas*; todos os tipos restantes são em princípio *instanciados*;

Metodologias de programação - 15

Programação orientada por objectos



A – tipo que contém o programa principal
b1 – instanciação do tipo **B**
c1 – instanciação do tipo **C**
c2 – instanciação do tipo **C**
c3 – instanciação do tipo **C**
D – tipo que contém métodos gerais
e1 – instanciação do tipo **E**
→ – instanciação, acesso a campo ou invocação de método

organização em mar de tipos de dados não-instanciados e instanciados (classes e objectos)

Metodologias de programação - 16

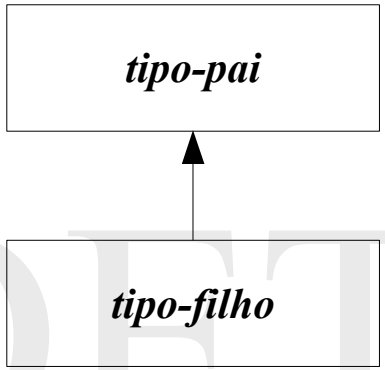
Programação orientada por objectos

- a segunda situação apontada pode ser resolvida pela aplicação do conceito de *reutilização* à construção de tipos dados;
- com efeito, se for possível definir novos tipos de dados a partir de tipos de dados pré-existentes, pode conseguir-se a sua diferenciação para os adequar a novas situações com um mínimo de esforço e de dispersão;
- este mecanismo designa-se de *herança* e possibilita estender numa perspectiva hierárquica ao novo tipo de dados, *tipo-filho* ou *subtipo*, as propriedades do tipo em que a definição se baseia, *tipo-pai* ou *supertipo*;
- concretamente, isto significa que
  - os campos *protegidos* da estrutura de dados interna
  - os métodos públicos

  do tipo base (tipo-pai) são directamente acessíveis e, no último caso, modificáveis no tipo-filho;

Metodologias de programação - 17

Programação orientada por objectos



- o processo de criação do novo tipo de dados, designado de mecanismo de *derivação*, permite
  - introduzir novos campos na estrutura de dados interna;
  - introduzir novos métodos públicos;
  - proceder ao *overriding* de métodos: redefinição de métodos públicos [anteriormente implementados] do tipo base;
  - proceder à *implementação* de métodos *virtuais*: definição de métodos públicos cujo interface é especificado no tipo base;

Metodologias de programação - 18

Programação orientada por objectos

- em termos de compatibilidade, o subtipo é *compatível* com o supertipo de que deriva, mas o inverso já não é sempre verdadeiro;

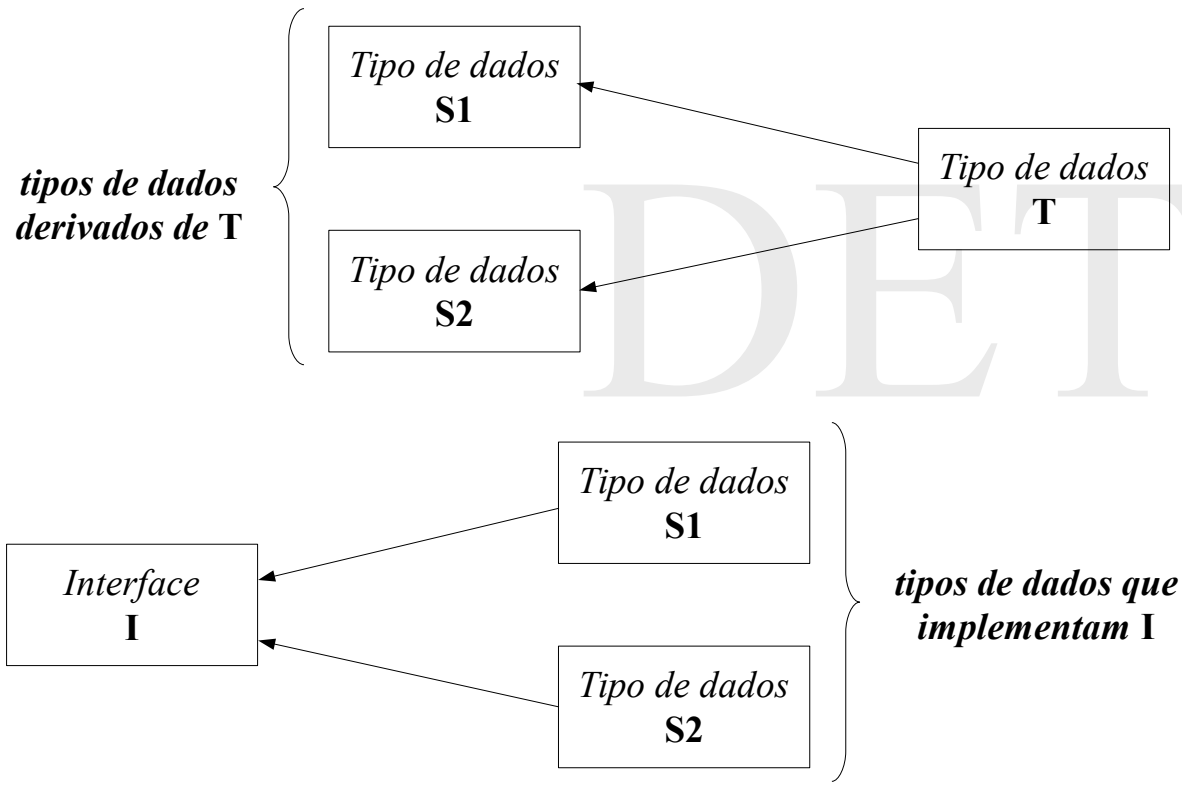
Metodologias de programação - 19

Programação orientada por objectos

- a relação de compatibilidade pode ser usada para se estabelecer um princípio muito poderoso do paradigma orientado por objectos, o *polimorfismo*, que consiste na possibilidade de uma mesma variável referenciar objectos de tipos diferentes, derivados do mesmo tipo base;
- sejam T um tipo de dados de referência, S1 e S2 dois tipos de dados distintos derivados do tipo T e t, s1 e s2 variáveis de cada um destes tipos em que as duas últimas foram instanciadas; considere-se agora que se atribui qualquer uma destas variáveis à variável t (operação sempre possível porque s1 e s2 são compatíveis com t, embora não sejam compatíveis entre si) e se invoca sobre ela um método definido em T;
- a implementação do método adequada ao objecto que está a ser referenciado é automaticamente escolhida e é realizada
  - na fase de compilação, *associação estática*, quando o método em questão não foi modificado na definição de qualquer dos tipos derivados em jogo;
  - em *runtime*, *associação dinâmica*, se tal tiver acontecido.

Metodologias de programação - 20

Programação orientada por objectos




```
T t;
I i;
S1 s1 = new S1 (...);
S2 s2 = new S2 (...);

if (condição)
   t = s1;
   else t = s2;
   ...
t.m (...);
```

invocação do método m sobre a variável t

```
   ...

if (condição)
   i = s1;
   else i = s2;
   ...
i.m (...);
```

invocação do método m sobre a variável i

Metodologias de programação - 21

Programação concorrente

- a decomposição funcional de soluções de problemas complexos conduz rapidamente à necessidade de se conceber um leque de actividades que se vão desenrolar de uma maneira mais ou menos autónoma;
- a manutenção de um fio de execução único impõe que se integre no código um *scheduler* mais ou menos elaborado que permita comutar entre a realização das diferentes actividades;
- esta abordagem, além de ser complexa e exigente, é também completamente inútil, já que os ambientes de execução actuais promovem a multiprogramação e torna-se, portanto, natural conceber a solução como um conjunto de processos que cooperam entre si;
- adicionalmente, com a popularização dos processadores *multicore*, os sistemas de operação têm vindo a implementar a gestão de *threads* ao nível do *kernel*, o que origina um acréscimo de eficiência na execução de aplicações concorrentes;

Metodologias de programação - 22

Programação concorrente

- são comuns duas abordagens na organização da solução
  - *abordagem event-driven*: os processos que implementam as diversas actividades estão bloqueados aguardando a ocorrência de um acontecimento que despoleta a sua execução; trata-se de processos mais ou menos independentes em que geralmente um e um só está activo de cada vez; a comunicação é efectuada através do acesso a uma estrutura de dados global que é mantida centralmente; os *gestores de espaços de visualização*, que interagem com o utilizador usando o rato, constituem talvez o exemplo mais divulgado de aplicação desta abordagem;
  - *abordagem interpar*: os diferentes processos que constituem a aplicação cooperam entre si de um modo mais ou menos específico; cada um deles é concebido como executando um ciclo de vida composto por operações independentes e operações de interacção, estas últimas comunicam ou recolhem informação, bloqueiam o processo até que determinadas condições estão reunidas, ou acordam outros processos quando determinadas condições acabaram de ser reunidas;
- qualquer que seja, porém, a abordagem seguida, um elemento sempre presente é que, ao contrário do que se passa na programação sequencial, não há garantia de reprodutibilidade de operações e, portanto, o *debugging* é muito mais sensível e muito menos eficaz;

Metodologias de programação - 23

Programação concorrente

- existem dois modelos base para partilha e comunicação de informação
  - *modelo de variáveis partilhadas*: constitui o modelo habitual em ambientes *multi-threading* onde os processos intervenientes escrevem e lêem valores armazenados numa estrutura de dados definida centralmente; para se evitar condições de corrida que conduzam à inconsistência da informação armazenada, o código de acesso à região partilhada (*região crítica*) tem que ser executado em regime de exclusão mútua; há ainda a necessidade de se prever dispositivos que possibilitem a sincronização;
  - *modelo de passagem de mensagens*: constitui um modelo de aplicação universal já que não exige a partilha de espaço de endereçamento; a comunicação é efectuada por troca de mensagens entre pares de processos (*unicast*), entre um processo e todos os outros (*broadcast*), ou entre um processo e todos os outros que pertencem ao mesmo grupo (*multicast*); assume-se a existência de uma infra-estrutura de canais de comunicação que interliga todos os processos intervenientes e que é gerida externamente à aplicação, garantindo o acesso com exclusão mútua ao(s) canal(is) e disponibilizando mecanismos de sincronização;

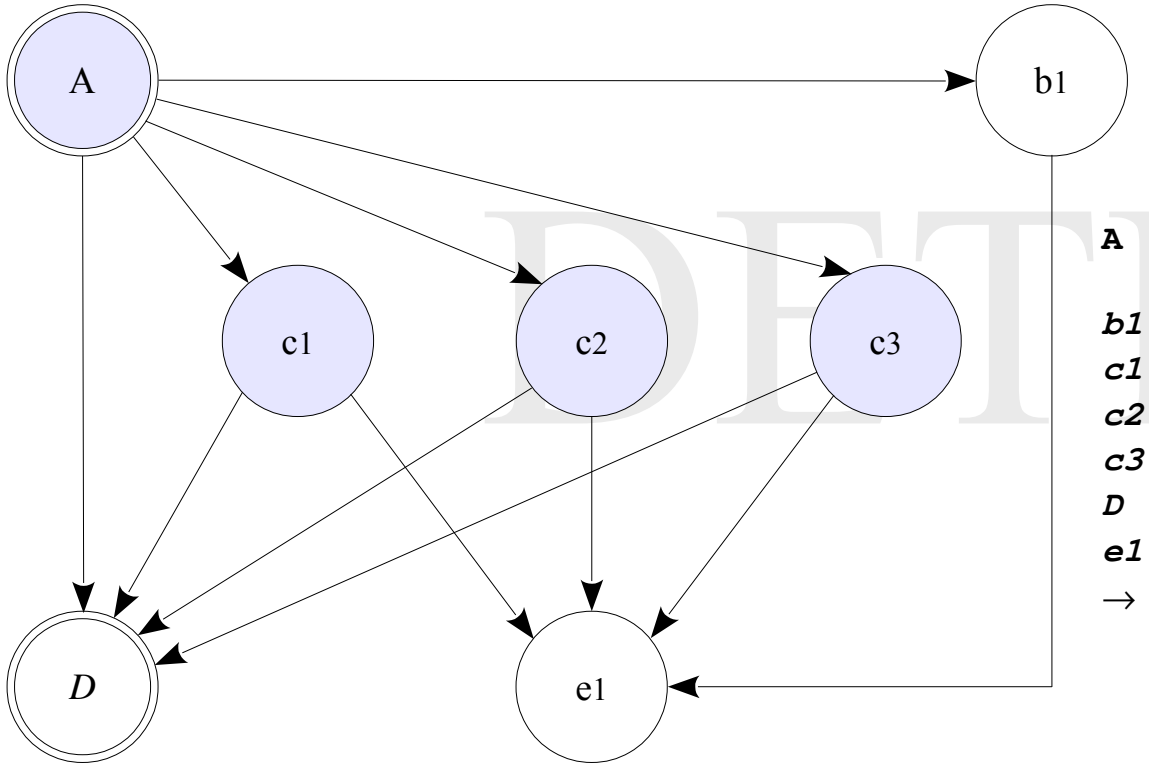
Metodologias de programação - 24

Programação concorrente

- um programa organizado segundo esta metodologia conforma-se aos princípios inerentes a uma decomposição modular, ou orientada por objectos, para descrição da interacção e é, por isso, também constituído por múltiplos ficheiros fonte;
- o que há de novo aqui é que, existindo mais do que um fio de execução, torna-se necessário distinguir claramente na descrição da interacção entre estruturas *activas* e estruturas *passivas*;
- as estruturas (ou entidades) *activas* correspondem aos diversos processos intervenientes, enquanto as entidades passivas correspondem à explicitação de diferentes tipos de funcionalidade.

Metodologias de programação - 25

Programação concorrente



`A` – tipo que contém o programa principal (entidade activa)
`b1` – instanciação do tipo **`B`**
`c1` – instanciação do tipo **`C`** (entidade activa)
`c2` – instanciação do tipo **`C`** (entidade activa)
`c3` – instanciação do tipo **`C`** (entidade activa)
`D` – tipo que contém métodos gerais
`e1` – instanciação do tipo **`E`**
$\rightarrow$ – instanciação, acesso a campo ou invocação de método

organização em mar de tipos de dados não-instanciados e instanciados e activos e passivos

Metodologias de programação - 26

Programação distribuída

- há duas razões principais que levam à passagem da *programação concorrente* para a *programação distribuída*
  - *paralelização*: tirar partido dos múltiplos processadores e outros componentes *hardware* existentes num sistema computacional paralelo para se obter uma execução mais rápida e eficiente da aplicação;
  - *disponibilização de um serviço*: fornecer de forma consistente, autónoma e segura uma funcionalidade bem definida a um grupo alargado de aplicações;
- a mudança de metodologia supõe, portanto, que se efectue de algum modo um mapeamento sobre sistemas computacionais distintos dos diferentes processos e centros de funcionalidade em que uma dada solução concorrente está dividida;

Programação distribuída

- o mapeamento, porém, não pode ser feito de um modo automático, existem diversos aspectos envolvidos no conceito de *plataforma de processamento paralelo* e que têm que ser avaliados cuidadosamente para que a migração seja possível;
- alguns deles são:
  - possíveis heterogeneidades entre os diferentes nós da plataforma de processamento;
  - ocorrência de falhas em um ou mais nós da plataforma de processamento ou na infra-estrutura de comunicações;
  - inexistência de um relógio global que permita efectuar a ordenação cronológica dos acontecimentos.