

# Aula 04

## Correcção

### Abordagens Sistemáticas à Programação

Programação II, 2019-2020

v1.6, 01-03-2020

DETI, Universidade de Aveiro

04.1

#### Objectivos:

- Tipos de Dados Abstratos.
- Correcção: Programação por Contrato;

## Conteúdo

<b>1</b>	<b>Tipos de Dados Abstratos</b>	<b>1</b>
<b>2</b>	<b>Abordagens Sistemáticas à Programação</b>	<b>3</b>
2.1	Testando o programa por fora . . . . .	3
2.2	Testando o programa por dentro . . . . .	3
2.3	Associando um significado aos programas . . . . .	3
2.4	Formalizando uma Especificação . . . . .	6
2.5	Programação por Contrato . . . . .	6
2.6	Programação por Contrato em Java . . . . .	8

04.2

## 1 Tipos de Dados Abstratos

**Tipo de Dados Abstrato (TDA):** É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e não pela forma como é implementado... (Definição mais completa adiante.)

- Um TDA descreve *o que* um tipo de dados representa sem ditar *como* o faz.
- Um TDA deve ter tudo o que precisamos de saber para *utilizar* o tipo de dados.
- Assim, o utilizador do tipo pode *abstrair-se* dos detalhes de implementação.
- Como veremos a seguir, o comportamento de um tipo de dados pode ser formalizado através de pré-condições, pós-condições e invariantes.
- A *abstracção* é a melhor forma de lidar com a complexidade.

04.3

### Exemplo: Mesma representação de dados, TDAs distintos

- Considere um tipo para representar frações.

---

```
public class Fraction {
    private int num;
    private int den;
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

---

- E um tipo para representar a posição de um pixel no ecrã.

---

```
public class Pixel {
    private int x;
    private int y;
    public Pixel move(int dx, int dy) { ... }
    public double distanceTo(Pixel) { ... }
}
```

---

- Ambos usam dois atributos inteiros para representar os *dados*.
- Mas têm *operações* com significado diferente.
- Logo, representam tipos de dados abstratos distintos.

04.4

### Exemplo: Dados diferentes, mesmo TDA

- Uma fração pode ser representada na forma  $\frac{N}{D}$ , por exemplo  $\frac{7}{3}$ .

---

```
public class Fraction {
    private int N; // Numerador
    private int D; // Denominador
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

---

- Mas também pode ser representada na forma mista  $I\frac{N}{D}$ , por exemplo  $2\frac{1}{3}$ .

---

```
public class Fraction {
    private int I; // Parte inteira
    private int N; // Numerador
    private int D; // Denominador (maior que N)
    public Fraction add(Fraction f) { ... }
    public Fraction multiply(Fraction f) { ... }
}
```

---

- São representações diferentes para os *dados*.
- Mas têm as mesmas *operações* com o mesmo significado.
- Logo, implementam o mesmo tipo de dados abstrato.

04.5

### Objectos - Abstracção de Dados

- As classes são uma forma de implementar tipos de dados abstratos.
- As operações do TDA são expostas pelas declarações de *membros públicos* de uma classe e constituem a sua *interface*.
- Os *membros privados* e os *corpos* dos métodos constituem a *implementação*.
- *Classe = Interface + Implementação*;

---

```
public class Data {
    public Data() { ... }
    public Data(int dia, int mes, int ano) { ... }
    public int dia() { ... }
    public int mes() { ... }
    public int ano() { ... }
    public boolean equals(Data outra) { ... }
    private ...
}
```

---

## 2 Abordagens Sistemáticas à Programação

Como foi referido na aula anterior, de todos os factores de qualidade dum programa, o mais importante é o da *correção*. Um programa que não resolve o problema para o qual foi feito é de pouca utilidade. Assim sendo, levanta-se uma questão pertinente: *como verificar se um programa está correcto*? Não vamos abordar a questão teórica formal de como *provar* que um programa está correcto, o que em geral é extremamente difícil, mas sim como testar ou verificar essa asserção.

### 2.1 Testando o programa por fora

Na prática, a forma como verificamos se um programa funciona é *testarmos* o programa em diferentes situações. Por exemplo, se temos um programa para determinar a raiz quadrada de um número real, então se testarmos o programa com o valor 4, estamos à espera de obter como resposta o valor 2. No entanto, um programa que divide por dois teria o mesmo resultado. Por isso é conveniente testar com outros valores. Na verdade, devemos testar o programa com um número elevado de valores para tentar confirmar que funciona bem.

Em bom rigor, se o programa falhar algum teste, podemos dizer que está errado, mas se passar todos os testes, não podemos dizer que está correto. Assim, devemos usar um conjunto de testes tão grande e variado quanto necessário por forma a tentarmos que o programa *falhe* em diferentes situações. No teste de programas usa-se um método análogo ao utilizado para testar hipóteses científicas: tentamos por todas as formas *refutar* (ou falsear) a hipótese de correção. Se não conseguirmos, então será mais provável que o programa não esteja errado e aumentará a nossa confiança de que está correto.

### 2.2 Testando o programa por dentro

O teste de programas pelo exterior pressupõe a existência de algo, ou alguém, que verifique se de facto o resultado do programa está correcto para determinados valores de entrada. Uma forma de concretizar isso será definir um conjunto conhecido de casos de teste e testar o programa sistematicamente para essas situações. No entanto, a aplicação desta técnica requer que um *árbitro exterior* ao próprio programa determine a correção desses testes. Esse árbitro poderá ser um segundo programa que se considere estar correto e contra o qual se comparam os resultados, por exemplo.

Será que podemos criar um “árbitro” automático dentro do próprio programa que estamos a desenvolver? Se o conseguirmos fazer, então teremos “dois em um”: teremos não só um programa que se testa a si próprio, como também um programa que *sabe* quando está errado, podendo agir em conformidade, se quiser.

Vamos ver como é que tal objectivo pode (e deve) ser concretizado.

### 2.3 Associando um significado aos programas

- Qualquer que seja o elemento de software em apreço – classe, função, bloco, instrução condicional, instrução repetitiva, atribuição de valor, etc. – existe sempre uma razão para a sua escolha e o seu uso tem um determinado significado (uma *semântica*).
- Não é boa ideia deixar esse significado apenas implícito no código, ou descrito apenas em documentação externa.

- O *significado deve ficar explícito* no próprio código fonte. Desse modo:

- Facilitamos a compreensão (perceber o significado) do software.
- Melhoramos a legibilidade.
- Potenciamos a correcção.

04.7

- Para procurar atingir esse objetivo devemos:

- Atribuir nomes sugestivos às classes, métodos, variáveis.
- Documentar adequadamente o código. Bons comentários devem conter significados que não sejam evidentes no próprio código.
- Anotar o elemento de software com *asserções*. Esta é uma abordagem ainda mais poderosa e eficaz que as anteriores.

04.8

- As *asserções* são expressões booleanas executáveis, que expressam condições esperadas sempre que o programa chega a esse ponto.
- Se a asserção preceder um elemento de software, diz-se que é uma *pré-condição* desse elemento.
- Se suceder ao elemento, então diz-se que é uma *pós-condição*.
- Cada asserção incluída no código fonte pode ser vista como
  - Uma *especificação* expressa de forma axiomática, que estipula o modo correto de utilizar o código nesse ponto do programa.
  - Uma *documentação* do funcionamento do código, sem o risco de ficar incoerente com o código.
  - Um *teste*, que pode ser verificado sistematicamente sempre que o programa é executado.

04.9

Note que a presença de asserções no código fonte possibilita que o programa detete automaticamente a sua própria incorrecção (sempre que a asserção é falsa), podendo agir em conformidade. Esta característica é aproveitada para melhorar o processo de depuração, mas também pode ser explorada para produzir programas tolerantes a falhas.

### Exemplo

- Este programa está correcto?

---

```
r = x;
q = 0;
while (r > y) {
    r = r - y;
    q = q + 1;
}
```

---

- Não sabemos! Depende do que é suposto ele fazer.
- Especificação:
  - Calcula o quociente  $q$  e o resto  $r$  como resultados da divisão inteira de  $x$  por  $y$ .

04.10

### Exemplo

- Este programa calcula o quociente  $q$  e o resto  $r$  como resultados da divisão inteira de  $x$  por  $y$ . Está correcto?

---

```
r = x;
q = 0;
while (r > y) {
    r = r - y;
    q = q + 1;
}
```

---

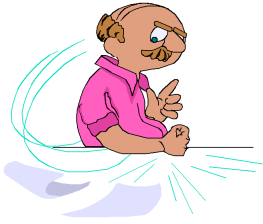
- *TALVEZ!* De acordo com a especificação podemos provar que no final:

$$x = y * q + r,$$

que é a *propriedade fundamental da divisão*.

04.11

## Algum tempo mais tarde



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- O programa não está correcto!
- Não termina quando  $y = 0$ !
- Obviamente que, por definição, não podemos dividir por zero.
- Valores negativos de  $x$  ou  $y$  também são problemáticos!

- Logo a especificação está incompleta.
- Devíamos ter “dito” que o programa só se aplica se  $y > 0 \wedge x \geq 0$ .

04.12

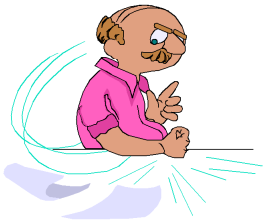
## Exemplo

```
Admitindo que:  $\{y > 0 \wedge x \geq 0\}$  ← pré-condição  
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}  
Podemos provar:  $\{x = y * q + r\}$  ← pós-condição
```

asserções

04.13

## Algum tempo mais tarde



```
r = x;  
q = 0;  
while (r > y) {  
    r = r - y;  
    q = q + 1;  
}
```

- O programa ainda não está correcto!
- Quando  $x = 6$  e  $y = 3$  o resultado é:

$$q = 1 \wedge r = 3$$

- em vez de:

$$q = 2 \wedge r = 0$$

- Oops! É um erro ... vamos ver...

04.14

## Exemplo

```
Admitindo que:  $\{y > 0 \wedge x \geq 0\}$   
r = x;  
q = 0;  
while ( $r \geq y$ ) {  
    r = r - y;  
    q = q + 1;  
}  
Podemos provar:  $\{x = y * q + r \wedge r < y\}$ 
```

rectificação

nova pós-condição

04.15

## 2.4 Formalizando uma Especificação

- Considere-se qualquer bloco de instruções  $A$ . A sua formulação em *lógica de Hoare* pode ser expressa como:

$$\{P\} A \{Q\}$$

- $P$  e  $Q$  são *asserções*:
  - $P$  é a *pré-condição* de  $A$ ;
  - $Q$  é a *pós-condição* de  $A$ .
- Significado:
  - Qualquer execução de  $A$ , começando num estado que satisfaça  $P$  deverá terminar num estado que satisfaça  $Q$ .
- Exemplo:

$$\{x \geq 9\} x = x + 5 \{x \geq 14\}$$

04.16

## 2.5 Programação por Contrato

- A *Programação por Contrato* (PpC) é uma abordagem à programação que acrescenta à programação modular a anotação sistemática dos programas com asserções.
- Segundo a PpC, cada função deve especificar as suas *pré-condições* e *pós-condições*.
- Cada tipo de dados, deve especificar as condições *invariantes*.
- A essa especificação, quando feita por asserções, dá-se o nome de *contrato do módulo*.
- Todas as asserções que definem as propriedades das operações públicas passam a ser parte integrante do tipo de dados abstrato.

04.17

### Contratos de Funções

- O contrato associado à especificação de funções é definido pelas *pré-condições* e *pós-condições* da função.
- Esse contrato faz parte da interface abstrata da função e deve manter-se mesmo que a implementação da função mude.
- Exemplo (raiz quadrada):

---

```
public static double sqrt(double x)
{
    assert x >= 0;
    double result;
    ...
    assert Math.abs(result*result-x) <= NEAR_ZERO;
    return result;
}
```

---

Diagrama de anotação de contrato para a função `sqrt`:

- A linha `assert x >= 0;` é destacada por uma caixa tracejada vermelha e rotulada como **pré-condição** por uma seta vermelha.
- A linha `assert Math.abs(result*result-x) <= NEAR_ZERO;` é destacada por uma caixa tracejada vermelha e rotulada como **pós-condição** por uma seta vermelha.

04.18

### Contratos de Objectos

- O contrato de um objecto é definido pelos contratos das suas funções públicas (ou seja, as suas **pré-condições** e **pós-condições**) conjuntamente com o **invariante** do objecto.
- As *pré-condições* e *pós-condições* descrevem propriedades à entrada e à saída de métodos.
- Os *invariantes* são condições que devem ser sempre respeitadas nos estados estáveis do objecto (ou seja quando estes são externamente utilizáveis).
- Por exemplo, a classe `Data` poderá ter o seguinte invariante:

---

```
valida(dia(),mes(),ano())
```

---

- Dessa forma simplificamos a concepção e a utilização do módulo `Data`, garantindo que os seus objectos representam **sempre** uma data válida.

04.19

A anterior definição de Tipo de Dados Abstrato está incompleta. A definição completa será:

**Tipo de Dados Abstrato (TDA):** É um *modelo* que descreve um tipo de dados apenas pelas *operações* que lhe são aplicáveis e pelo *contrato* dos seus objectos.

- Assim, são os contratos dos objectos que dão o significado ao respectivo Tipo de Dados Abstrato.
- Quando um contrato falha, normalmente o programa é interrompido e indica a linha onde o contrato falhou. (O erro estará sempre a montante dessa linha.)
- Para construir programas tolerantes a falhas, podemos recorrer ao mecanismo de excepções da linguagem e evitar que o programa termine, como veremos noutra aula.

04.20

### Distribuição de Responsabilidades

A PpC permite uma distribuição simples e clara de responsabilidades entre o módulo e os seus clientes:

	Obrigações	Benefícios
<b>Cliente</b>	Tem de garantir as pré-condições do módulo	Sabe que pós-condições e invariante são garantidos
<b>Módulo</b>	Tem de garantir o invariante e as pós-condições	Sabe que pré-condições são garantidas

04.21

Por exemplo, se a função `sqrt` especificada atrás for invocada com um argumento negativo, falhará a pré-condição. Ficamos a saber que o programa está errado, mas também podemos atribuir a culpa inequivocamente à parte do programa onde a invocação foi feita (a parte cliente), porque era ela a responsável por garantir a pré-condição. Por outro lado, se a invocação for feita com um valor não negativo, mas o quadrado do resultado não for próximo desse valor, então novamente o programa estará errado, mas agora a culpa recai na implementação da própria função (no módulo), porque é ela que tem de garantir a pós-condição.

### Escolha de Contratos

- Obviamente, a *escolha dos contratos* a associar a cada módulo (função, objecto) está nas mãos de quem o implementa.
- No entanto, como regra deve optar-se por *contratos* tão *fortes* quanto necessário para garantir implementações simples e para manter uma boa sensibilidade a falhas, mas sem restringir desnecessariamente o domínio de utilização nem complicar demasiado as condições.
- Por exemplo, no caso dos objectos do tipo `Data`, faz todo o sentido definir como invariante que as datas sejam válidas, já que torna bastante mais simples a compreensão e utilização destes objectos. Nunca será necessário lidar com datas absurdas como por exemplo 31 de Fevereiro de 2000.

04.22

## 2.6 Programação por Contrato em Java

### Asserções em Java

- Sintaxe:

---

```
assert booleanExpression [: expression ];
```

---

- Semântica:

- Se `booleanExpression` for `true`, a asserção passa.
- Se for `false`, a asserção falha e é gerado um *erro*, que normalmente provoca a terminação do programa e produz um relatório com o contexto que antecedeu a falha.
- `expression` é uma expressão opcional, geralmente uma `String`, que permite dar informação adicional sobre a falha.

04.23

### Asserções em Java (2)

- Por omissão, as asserções não são avaliadas.
- Para activar: (`-enableassertions` ou `-ea`):  
`java -ea Prog`
- Para desactivar (`-disableassertions` ou `-da`):  
`java Prog` ou `java -da Prog`
- O funcionamento do programa não deve depender da avaliação das asserções. Por isso, as expressões incluídas nas asserções nunca devem produzir efeitos secundários no estado do programa.
- A instrução `assert` só apareceu no Java versão 1.4.

04.24

O *erro* gerado quando uma asserção falha é na verdade uma *excepção* do tipo `AssertionError`. Por isso, as falhas de asserções podem ser interceptadas e tratadas usando o *mecanismo de excepções* da linguagem, como veremos na próxima aula.

### Asserções em Java: Exemplos

---

```
1 public class Assert1 {  
2     public static void main(String[] args) {  
3         assert false;  
4     }  
5 } Exception in thread "main" java.lang.AssertionError  
   at Assert1.main(Assert1.java:3)
```

---

---

```
1 public class Assert2 {  
2     public static void main(String[] args) {  
3         assert false: "disparate!";  
4     }  
5 } Exception in thread "main" java.lang.AssertionError: disparate!  
   at Assert2.main(Assert2.java:3)
```

---

04.25

### PpC em Java

A linguagem Java não suporta adequadamente a programação por contrato. Algumas das suas principais deficiências são as seguintes:

- Não distingue os diferentes tipos de asserções.
- Não tem suporte para a definição de invariantes de classe.
- As asserções não fazem parte da interface das classes.
- As aplicações de documentação (`javadoc`) não mostram os contratos de classe automaticamente.
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Apesar destas limitações (e outras, relacionadas com programação orientada por objectos), em Java nativo é possível fazer-se programação por contrato utilizando a instrução `assert`.

04.26