Aula 04

Correcção

Aproximações Sistemáticas à Programação

Programação II, 2014-2015

v1.1, 09-03-2015

DETI, Universidade de Aveiro

04.1

Objectivos:
- Tipos de Dados Abstractos.
- Correcção: Programação por Contrato;

Conteúdo

1 Tipos de Dados Abstractos **1**

2 Aproximações Sistemáticas à Programação **2**
2.1 Testando o programa por fora 2
2.2 Testando o programa por dentro 2
2.3 Associando um significado aos programas 3
2.4 Formalizando uma Especificação 5
2.5 Programação por Contrato 5
2.6 Programação por Contrato em `Java` 7

04.2

1 Tipos de Dados Abstractos

Tipos de Dados Abstracto (TDA): Tipo de dados definido apenas pelos serviços públicos que lhe são aplicáveis.

- Construção de modelos de objectos;
- As classes são uma forma de implementar tipos de dados abstractos.

```
public class Data {
   public Data() { ... }
   public Data(int dia, int mes, int ano) { ... }
   public int dia() { ... }
   public int mes() { ... }
   public int ano() { ... }
   public boolean igual(Data outra) { ... }
   public boolean anterior(Data outra) { ... }
   public boolean posterior(Data outra) { ... }
   ...
}
```

04.3

- De todos os passos abordados, é a melhor maneira de lidar com a complexidade;
- Permite separar a utilização do objecto da respectiva implementação;
- Algumas barreiras limitam a utilização dos dados a um pequeno conjunto de funções, ficando escondidos os dados e possivelmente outras funções;
- *Módulos = Interface + Implementação*;
- Como veremos a seguir, pode ser formalizado através de pré-condições, pós-condições e um invariante.

2 Aproximações Sistemáticas à Programação

Como foi referido na aula anterior, de todos os factores de qualidade dum programa, sem dúvida alguma que o mais importante a ter em conta é do da *correcção*. Um programa que não resolve o problema para o qual foi feito é de pouca utilidade.

Assim sendo, levanta-se a questão pertinente de *como verificar se um programa está correcto*! Não tanto a questão teórica formal de como *garantir* que um programa está correcto[1], 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[2] é *testarmos* o programa em diferentes situações. Assim, se é suposto, por exemplo, termos um programa que determina 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 como, por exemplo, um programa que divide por dois teria o mesmo resultado é conveniente testar com outros valores. Em bom rigor, devemos testar o programa com um número elevado de valores por forma a tentar confirmar que ele funciona.

Ou melhor, devemos testar o programa com um número tão grande quando necessário por forma a tentarmos que o mesmo *falhe* em diferentes situações. Na construção de programas faz-se uso do mesmo método utilizado em ciência: tentamos por todas as formas *falsificar* (ou seja: tornar errado) o funcionamento do programa. Se o não conseguirmos, então será mais provável que o programa não esteja errado (e aumentará a nossa confiança disso mesmo).

2.2 Testando o programa por dentro

O teste de programas do 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[3]. No entanto, a aplicação desta técnica requer que um *árbitro exterior* ao próprio programa determine a correcção desses testes[4].

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, se desejado, agir em conformidade).

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

[1] Objectivo em geral votado ao fracasso.

[2] Seria mais rigoroso afirmar: "a forma como verificamos se o programa não está errado".

[3] O servidor `SAAL` faz uso dessa técnica.

[4] No caso do servidor `SAAL`, esse árbitro é um programa que se considera ter resolvido o problema com correcção, pelo que basta comparar os resultados dos dois programas para ver se o novo programa está também "correcto".

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 um significado *(semântica)* na sua escolha e uso.
- Deixar esse significado implícito no próprio código, ou algures na especificação ou documentação externa, normalmente não é uma boa opção
- O significado deve ficar explícito no próprio código fonte.
  - Tornamos mais fácil a compreensão (tirar o significado) do software, aumentando a sua legibilidade e potenciando a sua correcção.
  - Uma aproximação (muito limitada) a esta opção consiste na boa prática de documentar adequadamente o código (como norma, os comentários devem conter significados que não sejam facilmente retirados do próprio código fonte executável).
  - Devemos atribuir nomes auto-explicativos às classes, métodos, variáveis, etc..

04.5

- Outra aproximação, de longe mais poderosa e eficaz, consiste em anotar o elemento de software desejado com *asserções* (expressões booleanas) executáveis, que expressem o que se espera que aconteça sempre que o programa chega a esse ponto.
- Dessa forma estamos a inserir no próprio código fonte (de uma forma axiomática) a especificação associada a esse ponto do programa. Se essa especificação preceder o elemento ela será uma sua pré-condição. Caso lhe suceda, então será uma pós-condição.
- Essas asserções passam a servir não só como uma especificação (correcção) como também servem para teste sistemático e documentação (sem o risco desta deixar de representar o estado actual do código, situação frequente quando há uma separação estrita entre os dois).

04.6

Note que a presença de asserções (principalmente, se executáveis) no código fonte permite que o próprio programa tenha condições para automaticamente determinar a sua própria incorrecção (o que acontecerá sempre que a asserção é falsa) podendo agir em conformidade (quer seja para melhorar substancialmente o processo de depuração, quer seja para se poder ter 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.

04.7

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

$$x = y * q + r$$

04.8

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 são também problemáticos!

- Logo a especificação está incompleta.
- Devíamos ter "dito" $y > 0$ *e* $x \geq 0$

04.9

Exemplo

```
Assumindo que:  {y > 0 and x ≥ 0}      ← pré-condição
   r = x;
   q = 0;
   while (r > y) {
        r = r - y;                      asserções
        q = q + 1;
   }
Podemos provar:  {x = y*q + r}         ← pós-condição
```

04.10

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:

$$q = 1 \ and \ r = 3$$

- em vez de:

$$q = 2 \ and \ r = 0$$

- Oops! É um erro . . . vamos ver. . .

04.11

Exemplo

```
Assumindo que:  {y > 0 and x ≥ 0}
   r = x;                              ← rectificação
   q = 0;
   while (r >= y) {
        r = r - y;
        q = q + 1;                     ← nova pós-condição
   }
Podemos provar:  {x = y*q + r and r < y}
```

04.12

2.4 Formalizando uma Especificação

- Considere-se qualquer bloco A. A sua *formulação* pode ser expressa como:

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

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

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

04.13

2.5 Programação por Contrato

- É da junção entre a programação modular e a anotação sistemática de programas com asserções que surge uma nova, e poderosa, metodologia: a *Programação por Contrato*.
- De facto, um módulo – seja ele resultante de abstracção algorítmica (função) ou de um Tipo de Dados Abstracto (objecto) – será sempre uma aproximação incompleta aos critérios de modularidade caso não lhe esteja fortemente associada a especificação que o define e lhe dá significado!
- A essa especificação, quando feita por asserções, dá-se o nome de *contrato do módulo*.

04.14

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 abstracta da função, mantendo-se em eventuais mudanças de implementação da função.
- Exemplo (raiz quadrada):

```
public static double sqrt(double x)
{
   assert x >= 0;          <-- pré-condição

   double result;
   ...

   assert Math.abs(result*result-x) <= NEAR_ZERO;
   return result;
}                          <-- pós-condição
```

04.15

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, e obrigando, que os seus objectos representam **sempre** uma data válida.

04.16

A definição de Tipo de Dados Abstracto dada na aula anterior está incompleta, a sua definição completa será:

Tipos de Dados Abstracto (TDA): Tipo de dados definido apenas pelos serviços públicos 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 Abstracto;
- Quando um contrato falha em tempo de execução, normalmente a execução do programa termina localizando o ponto no programa onde o contrato falhou (o erro estará sempre a montante desse ponto);
- Para a construção de programas tolerantes a falhas, pode-se evitar que o programa termine na presença de uma falha de contrato (como veremos, isso pode fazer-se recorrendo aos mecanismos de excepções existentes na linguagem `Java`).

04.17

Distribuição de Responsabilidades

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

	Obrigações	**Benefícios**
Cliente	Garantir as pré-condições do módulo	Garantia das pós-condições e invariante
Módulo	Garantir o invariante e as pós-condições	Garantia das pré-condições

04.18

Por exemplo, caso a função `sqrt` atrás especificada seja invocada com um argumento negativo, não só o programa estará errado, como também se pode inequivocamente atribuir a culpa à parte do programa onde a invocação foi feita. Por outro lado, caso a invocação seja feita com um valor não negativo, e o quadrado do resultado da função não seja próximo desse valor, então novamente o programa estará errado, mas agora a culpa recai na implementação do própria função.

Escolha de Contratos

- A *escolha dos contratos* a associar a cada módulo (função, objecto) está, é claro, nas mãos de quem os implementa;
- No entanto, como regra deve-se optar por *contratos* tão *fortes* (mais restritivos) quanto possível, já que quanto mais fortes estes forem, mais fácil será maximizar a correcção das respectivas implementações (já que estas tendem a ser mais simples);
- Por exemplo, no caso dos objectos do tipo `Data`, faz todo o sentido definir como invariante datas válidas já que torna bastante mais simples a compreensão destes objectos e bastante mais simples a sua utilização (nunca será necessário considerar a possibilidade de existirem datas com valores absurdos, como por exemplo: `31 de Fevereiro de 2000`).

04.19

2.6 Programação por Contrato em Java

Asserções em Java

- Sintaxe:

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

  - Se `booleanExpression` for `true`, a asserção passa;
  - Se for `false`, a asserção falha, havendo lugar, por omissão, à terminação do programa indicando o contexto completo envolvendo a falha;
  - `expression` é uma expressão opcional que permite passar informação adicional sobre o tipo de problema ocorrido.

- Como as asserções são usadas para testar condições que nunca devem falhar num programa correcto, as expressões nela existentes nunca devem produzir efeitos colaterais no estado do programa;
- As primeiras versões de `Java` não tinham Asserções (só a partir da versão 1.4);
- Por omissão, as asserções não são avaliadas;
- Activar (`-enableassertions` ou `-ea`):
  `java -ea Prog`
- Desactivar (`-disableassertions` ou `-da`):
  `java Prog` ou `java -da Prog`

04.20

Como veremos na próxima aula, uma descrição mais completa do que acontece num programa na presença de uma asserção falsa envolve um mecanismo de gestão de falhas em programas denominado *mecanismo de excepções*, sendo gerada uma excepção do tipo `AssertionError`.

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)
```

04.21

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 vão automaticamente buscar os contratos de classe;
- Não é possível activar e desactivar contratos por tipo de contrato e por objecto.

Embora com todas estas limitações (para além de outras, relacionadas com programação orientada por objectos), em `Java` nativo é possível fazer-se programação por contrato utilizando a instrução: `assert`.

04.22