Teste e depuração de programas

Mestrado Integrado em Engenharia Informática FCT UNL
http://ctp.di.fct.unl.pt/miei/ip/

Corpo Docente 2020/2021
António Ravara, Artur Miguel Dias, Bernardo Toninho,
Ema Vieira, Inês Fernandes, Margarida Mamede,
Miguel Monteiro, Rui Nóbrega

A importância dos testes na Programação

- Quando escrevemos um programa, por vezes ficamos erradamente convencidos que o programa está pronto e se comporta da maneira pretendida em todas as circunstâncias.
- Muitas vezes, as primeiras versões dos nossos programas têm defeitos (*bugs*) e exibem comportamentos errados para determinados valores de entrada.
- Nunca é boa ideia facilitar. É sempre salutar criarmos testes para os nossos programas, escolhendo os objetos e valores de entrada com maior probabilidade de expor problemas.
- Escolher os melhores cenários a montar em testes é algo que melhora com o treino, bem como ter “olho” para o cenários mais certeiros.

Testando um programa

- A abordagem mais básica consiste em testar um programa usando diversos valores de entrada.
  - Será que os resultados são os esperados e coincidem com as contas feitas à mão?
  - Não basta correr os exemplos do enunciado. Temos de inventar testes que cubram todas as possibilidades.
- Os **valores usados nos testes** não são escolhidos ao acaso. Os testes a um programa devem sempre incluir:
  - Alguns **casos normais**;
  - Todos os tipos de **casos especiais** que seja possível imaginar.
  - Todos os tipos de **casos limite** que seja possível imaginar.
- Os testes ajudam a descobrir as falhas do programa. Se o programa passar todos os testes, então ganhamos confiança no programa.

Testando um programa

- Exemplo: **CSI Portugal**
- O programa determina se um suspeito tem, ou não tem, um álibi.

  O input são dois intervalos de inteiros: o primeiro intervalo **[I, J]** indica o período de tempo em que o crime ocorreu; o segundo intervalo **[S, T]** indica um período de tempo em que o suspeito esteve longe do local do crime.
- **Casos normais**. Por vezes existe alguma arbitrariedade sobre o que se considera um caso normal. Aqui vamos considerar como casos normais duas situações: intervalos que se intersectam (mas sem que um intervalo esteja completamente contido dentro do outro) e ainda intervalos que não se intersectam:

```
[I, J]  [S, T]
[3, 7]  [4, 10]        // intersectam-se
[4, 10] [3, 7]
[3, 7]  [8, 10]        // não se intersectam
[8, 10] [3, 7]
```

Testando um programa

- **Casos especiais**. Nesta categoria, vamos considerar os casos em que um intervalo está completamente contido dentro do outro, incluindo o caso em que os dois intervalos são iguais:

```
[I, J]  [S, T]
[3, 7]  [0, 10]    // um contido no outro
[0, 10] [3, 7]
[0, 10] [0, 10]    // iguais
```

- **Casos limite 1**. Aqui vamos considerar todas as situações com intervalos mínimos:

```
[I, J]  [S, T]
[3, 3]  [0, 7]     // mínimo com não mínimo
[0, 7]  [3, 3]
[3, 3]  [4, 4]     // mínimo com mínimo
[4, 4]  [3, 3]
[3, 3]  [3, 3]     // iguais
```

- **Casos limite 2**. Aqui vamos considerar todas as situações com intervalos distintos não mínimos, mas em que alguma extremidade se toca:

```
[I, J]  [S, T]
[0, 3]  [3, 7]     // intervalos encostados
[3, 7]  [0, 3]
```

Depurando um programa

- Se, com testes manuais, descobrimos um erro, então relemos o programa, concentrando-nos nas partes relacionadas com o erro:
  - Idealmente descobrimos a origem do problema no código e fazemos a correção.
  - Caso contrário, colocamos mensagens em zonas críticas do código para confirmar se o que está a acontecer dentro do programa corresponde mesmo ao que pensamos.
- Exemplo: O seguinte programa pretende somar os primeiros 10 números naturais (i.e. a começar em 0) ou seja fazer a conta 0+1+2+3+4+5+6+7+8+9.

```
public class Main {
     private static int sum(int n) {   // metodo errado
          int i, sum=1;
          for( i = 0 ; i < n-1 ; i++ ); {
               sum += i;
          }
          return sum;
     }
     public static void main(String[] args) {
          System.out.println(sum(10));
     }
}
```

Depurando um programa

- O resultado esperado é 45 (porque 45 = 0+1+2+3+4+5+6+7+8+9). Mas o nosso programa está a produzir 10 - claramente errado!
- Vamos inserir um **println** dentro do ciclo para verificar se o ciclo está a funcionar bem. Vamos mostrar no ecrã os valores de **i** a evoluir dentro do ciclo.

```
public class Main {
     private static int sum(int n) {   // metodo errado
          int i, sum=1;
          for( i = 0 ; i < n-1 ; i++ ); {
               System.out.println(i);
               sum += i;
          }
          return sum;
     }
     public static void main(String[] args) {
          System.out.println(sum(10));
     }
}
```

- O ciclo devia mostrar os valores de i a variar de 0 até 9. Mas o programa mostra um único valor para i, o valor 9.
- Descobrimos então que há um ";" errado no final da linha do for. Retirando esse ";" e correndo novamente o programa já aparecem valores sucessivos para i. A situação melhorou!

Depurando um programa

- Contudo, os valores de i não alcançam o 9, como queríamos, mas param no 8!
- Descobrimos então que a condição do **for** está errada. É preciso trocar **i < n-1** por **i < n**. Correndo o programa corrigido, vemos que o ciclo já efetua todas as iterações pretendidas, com o i a variar de 0 até 9.
- Mesmo assim, o resultado final ainda não está certo. Esperávamos o resultado 45 e o programa está a produzir 46. Olhando melhor para o código e considerando que o desvio é de apenas uma unidade, se calhar o erro está na inicialização da variável **sum**.
- Está mesmo! Trocando o 1 por 0, já fica tudo bem. Podemos agora apagar o **println**, porque já não faz falta.

```
public class Main {
     private static int sum(int n) {   // metodo corrigido
          int i, sum=0;
          for( i = 0 ; i < n ; i++ ) {
               sum += i;
          }
          return sum;
     }
     public static void main(String[] args) {
          System.out.println(sum(10));
     }
}
```

Testando um programa

- O que se explicou nos slides anteriores é o **mínimo que se espera** de quem escreve um programa. Ou seja:
  - O programa final é testado com diversos valores de entrada.
  - Em caso de resultados errados, o programa é corrigido e novamente testado.
- No caso dum **programa pequeno**, poderá ser suficiente testar desta forma o programa final.
- Mas, no caso dum **programa grande**, com diversas classes, numerosos métodos, e ainda por cima com inputs muito variados, poderá não ser prático, ou então não dar suficiente confiança, testar o programa completo da forma que se descreveu.
- A melhor alternativa é fazer testes de pormenor durante o próprio desenvolvimento do programa, sem esperar pelo programa final. A ideia é ir testando separadamente cada unidade do programa, em particular as classes e os métodos mais importantes, à medida que são escritos.
- No desenvolvimento profissional de software, faz-se sempre dessa maneira e é essa a abordagem que se explica a seguir.

Programas que se testam a si próprios

Numa aproximação simplista, cada teste consistiria no envio de mensagens para a consola, que o programador teria de examinar, comparando as saídas obtidas com as que eram esperadas (e desejáveis).

Nessa abordagem, o programador terá de examinar um "relatório" na consola para cada teste. Tais tarefas rapidamente se tornam entediantes. Por causa disso, o programador evitará realizá-las frequentemente. Nem são escaláveis para muitos testes.

A abordagem dos **testes unitários** permite que sejam as próprias classes fazerem essas comparações.

Para isso, cada teste inclui um **valor esperado**, que é comparado com o valor obtido. Se os valores são idênticos (duma maneira geral, se a condição a testar é verdadeira) o programador não tem de examinar nada. Se todos os testes passarem, ao programador basta-lhe saber isso. Com esta abordagem, o programador pode executar muitos testes e a tarefa de cobrir um programa com muitos testes torna-se escalável.

Programas que se testam a si próprios

Nos próximos diapositivos, são mostrados dois exemplos ilustrativos de como montar diversos testes (i.e., cenários de teste) de modo a serem os próprios testes a dizerem se passaram ou não.

Os testes dizem-se **unitários**, porque destinam-se a testar unidades, i.e., os elementos mais simples de um programa. Essas unidades são tipicamente operações (funções ou métodos) ou objectos.

No primeiro exemplo, veremos como testar uma simples operação, na qual não é importante pensar no objecto de que faz parte.

No segundo exemplo, veremos como testar um objecto que disponibiliza diversas operações. Nesse exemplo, veremos como testar:

- Um método que devolve determinado valor.
- Um método **`void`**.
- Um construtor.

Exemplo ilustrativo 1

O teste 1 de IP2021 incluiu uma questão na qual se pediu o seguinte método:

```
/**
 * Determina o número de ordem duma data dentro dum ano.
 * Permite saber se a data representa o 1º dia do ano, o 2º dia, etc.
 * @param day representa o dia do mes
 * @param month representa o mes: inteiro [1..12]
 * @param isLeapYear  indicação se o ano e bissexto
 * @return numero de ordem da data
 */
public int dayOfYear(int dia, int mes, boolean leap year).
```

Exemplos de utilização:

```
dayOfYear(1, 1, false)  == 1    // 1 de janeiro dum ano comum
dayOfYear(1, 3, false)  == 60   // 1 de março dum ano comum
dayOfYear(1, 3, true)   == 61   // 1 de março dum ano bissexto
dayOfYear(31, 12, true) == 366  // 31 de dezembro dum ano bissexto
```

Cada um dos exemplos acima fornece um cenário de teste, que pode dar origem a um método de teste.

Exemplo ilustrativo 1

Cada cenário de teste deve incluir um **valor esperado**. Dependendo dos detalhes de cada cenário, poderá prever um ou vários **valores de entrada**.

```
private static boolean testDayOfYear(
        int expected, int d, int m, boolean ly){
    return expected == dayOfYear(d, m, ly);
}
```

O método de teste é do tipo **`boolean`**, para nos dizer se o teste passou (**`true`**) ou não (**`false`**).

Exemplo ilustrativo 1

Cada cenário de teste deve incluir um valor esperado. Dependendo dos detalhes de cada cenário, poderá prever um ou vários valores de entrada.

```
private static boolean testDayOfYear(
        int expected, int d, int m, boolean ly ){
    return expected == dayOfYear(d, m, ly);
}
```

Valor esperado

Valores de entrada (neste caso concreto)

Numa primeira aproximação, sugerimos que os métodos de teste sejam colocados na classe que contém o método a testar. Porém, se o método a testar for **`public`**, os testes podem ser colocados numa classes separada.

Exemplo ilustrativo 1

Cada cenário de teste deve incluir um valor esperado. Dependendo dos detalhes de cada cenário, poderá prever um ou vários valores de entrada.

```
private static boolean testDayOfYear(
        int expected, int d, int m, boolean ly ){
    return expected == dayOfYear(d, m, ly);
}
```

Os 4 cenários de teste atrás mencionados dão origem às seguintes chamadas, que colocamos num método separado, que poderá conter uma sequência com os cenários todos:

```
private static boolean allTests() {
    boolean result =   testDayOfYear(  1,  1,  1, false);
    result = result && testDayOfYear( 60,  1,  3, false);
    result = result && testDayOfYear( 61,  1,  3, true);
    result = result && testDayOfYear(366, 31, 12, true);
    return result;
}
```

Exemplo ilustrativo 1

Cada cenário de teste deve incluir um valor esperado. Dependendo dos detalhes de cada cenário, poderá prever um ou vários valores de entrada.

```
private static boolean testDayOfYear(
        int expected, int d, int m, boolean ly ){
    return expected == dayOfYear(d, m, ly);
}
```

Valores esperados

Os 4 cenários de teste atrás mencionados dão origem às seguintes chamadas, que colocamos num método separado, que poderá conter uma sequência com os cenários todos:

```
private static boolean allTests() {
    boolean result =   testDayOfYear(  1,  1,  1, false);
    result = result && testDayOfYear( 60,  1,  3, false);
    result = result && testDayOfYear( 61,  1,  3, true);
    result = result && testDayOfYear(366, 31, 12, true);
    return result;
}
```

Exemplo ilustrativo 1

Cada cenário de teste deve incluir um valor esperado. Dependendo dos detalhes de cada cenário, poderá prever um ou vários valores de entrada.

```
private static boolean testDayOfYear(
        int expected, int d, int m, boolean ly ){
    return expected == dayOfYear(d, m, ly);
}
```

Valores de entrada para os diversos testes

Os 4 cenários de teste atrás mencionados dão origem às seguintes chamadas, que colocamos num método separado, que poderá conter uma sequência com todos os nossos cenários de teste:

```
private static boolean allTests() {
    boolean result =   testDayOfYear(  1,  1,  1, false );
    result = result && testDayOfYear( 60,  1,  3, false );
    result = result && testDayOfYear( 61,  1,  3, true  );
    result = result && testDayOfYear(366, 31, 12, true  );
    return result;
}
```

Exemplo ilustrativo 1

```
private static boolean testDayOfYear(int expected, int d, int m, boolean ly){
    return expected == dayOfYear(d, m, ly);
}

private static boolean allTests() {
    boolean result =   testDayOfYear(  1,  1,  1, false);
    result = result && testDayOfYear( 60,  1,  3, false);
    result = result && testDayOfYear( 61,  1,  3, true);
    result = result && testDayOfYear(366, 31, 12, true);
    return result;
}
```

Por fim, incluímos um método que nos diz se todos os testes passaram… ou não

```
public static void main(String[] args) {
    if(allTests())
        System.out.println("All tests passed :-)");
    else System.err.println("At least one test failed :-(");
}
```

Exemplo ilustrativo 2

No segundo exemplo, examinamos um exemplo mais elaborado: em vez de uma simples “função”, queremos testar um objecto que oferece várias operações.

Vamos testar a classe `SafeBankAccount`.

O que testar?

1. Começamos com um cenário simples, em que testamos o estado interno do objecto acabado de criar - ou seja, testamos o próprio construtor - se este efectua a(s) inicialização(ões) esparadas.
2. Depois, podemos testar as operações mais usuais, e.g., **`deposit()` e `withdraw()`**
3. Depois, continuamos com outras operações, e.g., os métodos **`computeInterest()` e `applyInterest()`**

Exemplo ilustrativo 2 - testar o construtor

Considere que incluímos no projecto, uma classe `SafeBankAccounTester`, que terá o seu próprio método `main()`.

Como é típico destas classes, os métodos auxiliares são **`private static`**.

Repare que cada cenário de teste criará a sua própria instância da classe a testar. Isso permite que os diversos testes possam correr em qualquer ordem, o que é muito conveniente.

```
private static boolean
        testConstructor(int expected, int initial) {
    SafeBankAccount ba = new SafeBankAccount( initial );
    return expected == ba.getBalance();
}
```

Exemplo ilustrativo 2 - testar métodos `void`

```
private static boolean
      testDeposit(int expected, int initial, int amount) {
        SafeBankAccount ba = new SafeBankAccount( initial );
        ba.deposit( amount );
        return expected == ba.getBalance();
}

private static boolean
   testWithdraw(int expected, int initial, int amount) {
        SafeBankAccount ba = new SafeBankAccount( initial );
        ba.withdraw(amount);
        return expected == ba.getBalance();
}
```

Exemplo ilustrativo 2 - testar métodos `void`

Até este ponto, o método contendo todos os testes poderá incluir os seguintes cenários de teste:

```
private static boolean allTests() {
    boolean result = true;
    result = result && testConstructor(5000, 5000);
    result = result && testDeposit(7300, 5000, 2300);
    result = result && testWithdraw(7000, 10000, 3000);
    ...
    return result;
}
```

O facto dos valores concretos de determinado cenário serem parametrizados, tem a vantagem de que podemos acrescentar mais cenários para as mesmas operações sem haver necessidade de criar mais um método de teste.

Por exemplo, podemos (e devemos) acrescentar pelo menos um cenário para o caso de tentativa de levantamento sem provimento (i.e., saldo suficiente):

```
result = result && testWithdraw(4800,  5000, 6000);
```

Exemplo 2 - testar métodos não **`void`**

E se o método a testar não é **`void`** e devolve um resultado concreto?

É o caso do método `computeInterest()`:

```
private static boolean testComputeInterest(
    int expected, int initialBalance) {
    SafeBankAccount ba = new SafeBankAccount( initialBalance );
    return expected == ba.computeInterest();
}
```

É prudente acrescentar pelo um cenário de teste para cada escalão de juros:

```
private static boolean allTests() {
    boolean result = true;
    ...
    result = result && testComputeInterest(   9,     900);
    result = result && testComputeInterest(  80,    4000);
    result = result && testComputeInterest(3000,  100000);
    ...
```

Exemplo 2

Testando o método `applyInterest()`:

```
private static boolean testApplyInterest(
        int expected, int initialBalance) {
    SafeBankAccount ba = new SafeBankAccount( initialBalance );
    ba.applyInterest();
    return expected == ba.getBalance();
}
```

Novamente, incluir pelo um cenário de teste para cada escalão de juros:

```
private static boolean allTests() {
    boolean result = true;
    ...
    result = result && testApplyInterest(   909,     900);
    result = result && testApplyInterest(  4080,    4000);
    result = result && testApplyInterest(103000,  100000);
```

Exemplo 2

Esta técnica é simples e permite acrescentar muitos cenários de teste

```
private static boolean allTests() {
    boolean result = true;
    result = result && testConstructor(5000,    5000);
    result = result && testDeposit(7300, 5000, 2300);
    result = result && testWithdraw(7000, 10000, 3000);
    result = result && testWithdraw(4800,  5000, 6000);
    result = result && testComputeInterest(   9,     900);
    result = result && testComputeInterest(  80,    4000);
    result = result && testComputeInterest(3000,  100000);
    result = result && testApplyInterest(   909,     900);
    result = result && testApplyInterest(  4080,    4000);
    result = result && testApplyInterest(103000,  100000);
    return result;
}
```

Porém, tem uma limitação…

Interpretar os resultados

O que fazer, quando pelo menos um dos testes falha?

Nesses casos, tudo o que o programador vê, é a mensagem na consola

```
At least one test failed :-(
```

```
...
private static void driveTests() {
    if(allTests())
        System.out.println("All tests passed :-)");
    else System.err.println("At least one test failed :-(");
}
```

Como saber que teste ou testes falharam?

Interpretar os resultados

Uma abordagem simples e fácil, é "neutralizar" a maioria dos testes, colocado-os dentro de comentários de linha

```
result = result && testConstructor(5000,    5000);
//result = result && testDeposit(7300, 5000, 2300);
//result = result && testWithdraw(7000, 10000, 3000);
//result = result && testWithdraw(4800,  5000, 6000);
//result = result && testComputeInterest(   9,     900);
//result = result && testComputeInterest(  80,    4000);
//result = result && testComputeInterest(3000,  100000);
//result = result && testApplyInterest(   909,     900);
//result = result && testApplyInterest(  4080,    4000);
//result = result && testApplyInterest(103000,  100000);
...
```

Depois, ir gradualmente "reactivando" os testes, executando o método `driveTests()` até surgir novamente a mensagem:

"`At least one test failed`"

A importância dos testes na Programação

Como já vimos, os valores a usar em cada cenário de teste não devem ser escolhidos ao acaso. Os testes devem incluir:

- Pelo menos um caso “normal”;
- Todos os “casos especiais” e “casos limite” que seja possível imaginar, tanto relativamente ao parâmetros recebidos pelos métodos, como relativamente aos seus dados de retorno.
- Será que estes métodos têm pré-condições? Como reage um método, se este recebe valores de entrada que violam as condições?

- Em processamentos que usam vectores (e.g., pesquisas, inserções ou remoções ordenadas), os erros tendem a ocorrer nos “casos fronteira”, i.e., situações que envolvem o primeiro ou último elemento do vector. Também em situações em que o vector está “vazio”, ou tem a capacidade esgotada.

A importância dos testes na Programação

Note-se que muitas vezes, o valor do objecto (e.g., um vector) não está directamente acessível, pois é um campo privado da classe. O teste teria de ser colocado na classe que inclui o campo de dados, o que pode não ser o mais conveniente.

Esse obstáculo pode ser contornado facilmente, fazendo o teste chamar operações públicas que levam à situação que se pretende testar, e.g.

- vector vazio ou todo cheio
- procurar valor que não existe
- inserção ordenada de valor inferior/superior a qualquer dos valores já existentes

Agora, o teste já pode ser colocado em qualquer classe

- Muitas vezes, pode existir uma classe só para testes
- Se os testes forem muitos, pode justificar-se haver uma classe de teste para cada classe testada.