Tutorial 5 – Parte B

Testes

André Souto

Unidade Curricular de
Laboratórios de Programação

2020/2021

Resumo

Este é o tutorial sobre testes para as aulas de LabP 2020/2021. O material aqui apresentado é uma adaptação (com cópia parcial) de um documento sobre o mesmo tema produzido pelo **Professor Doutor Vasco Thudichum Vasconcelos**, a quem são devidos os créditos de autoria e a quem se agradece a gentileza da cedência do material. Este documento apenas serve para suporte às aulas de LabP.

O que é e porquê fazer testes?

Ao longo do semestre já tem visto e usado testes através do JUnit. Certamente já se deve ter perguntado o que é um teste e deve ter alguma perceção sobre a importância de testar o código que produzimos.

Designa-se por **teste unitário** um pedaço de código que é concebido para verificar a conformidade de um método, classe ou outro artefacto de software que possa ser suscetível de falhas.

Concretizando esta ideia num exemplo, suponhamos que se pretende testar um método que calcula a soma de todos os números inteiros não negativos até um dado n. Um exemplo de teste é um método que seleciona um determinado número inteiro positivo (ou outro, como veremos mais à frente), calcula o resultado esperado de somar todos os números até ao inteiro escolhido usando um outro mecanismo qualquer de cálculo (por exemplo, o mental) e verifica se a aplicação do método que está a ser testado a este mesmo inteiro devolve o resultado esperado.

A questão "*porquê fazer testes*" tem para a maioria dos alunos uma resposta certamente empírica e evidente: "*para minimizar problemas (erros ou falhas) do desenvolvimento do código*". Esta é claramente uma resposta que motiva o uso de testes. *Mas porquê unitários?* Como o próprio nome indica deve testar-se "*um a um*" cada artefacto (método, classe, etc.) de modo a ser fácil o rastreio de qual é a parte do código que está a causar as falhas ou problemas.

De um modo geral há muitos artefactos de software que podem ser testados. Neste documento apenas focamos testes sobre métodos. De um modo generalizado, pode dizer-se que o espaço de entrada de um método contém não só os parâmetros que aparecem explícitos na definição do método, mas também valores de atributos *static*, para além de todo o ambiente de execução do programa. Neste documento, de todos estes componentes do espaço de entrada, vamos considerar unicamente os parâmetros de entrada dos métodos, tratando assim apenas métodos que nã acedem a atributos de classe, ou seja, aos atributos declarados como *static*.

Chama-se a atenção para a possibilidade de se poder testar outros artefactos que não só métodos, nomeadamente classes, objetos, etc…

Testes por partição do espaço de entrada

Para gerar testes usaremos *partições*. Dado um método, como obter as partições necessárias? As partições são identificadas com base no *modelo do espaço de entrada*. Esta é a fase mais criativa do processo de teste. Um modelo do espaço de entrada descreve o espaço de entrada do método a um certo nível de abstração que deve descrever a estrutura do espaço de entrada em termos de *características*. Cada característica dá origem a uma partição formada por um conjunto de blocos. Dos blocos retiramos depois valores para os testes.

O domínio de entrada é *particionado* em *regiões* que assumimos terem valores igualmente úteis do ponto de vista da atividade de teste. Selecionamos depois valores contidos nestas regiões.

Este processo de gerar testes traz algumas vantagens:

- Requer pouco treino;
- Quem testa, não precisa de entender a implementação do método;
- Os testes podem ser reutilizados em caso de alteração da implementação.

Para realizar testes considerando a partição do espaço de entrada devem-se seguir os seguintes cinco passos:

1. Identificar os parâmetros de entrada;

2. Modelar o domínio de entrada através da definição de uma ou mais *características* deste. Para além dos parâmetros, tomamos também em consideração o comportamento esperado para o método;

3. Definir regiões que particionam o espaço de entrada tendo por base as características;

4. Aplicar algum critério sobre as características do domínio de entrada, identificando deste modo um conjunto de regiões e valores;

5. Derivar os dados de *input* para os testes (isto é, os casos de teste).

No caso da linguagem Java, como *parâmetro de entrada*, iremos considerar apenas os parâmetros dos métodos que estão explícitos na sua assinatura. Por exemplo, para o método `encontra` definido abaixo, os parâmetros são `lista` e `elemento`.

```
/**
 * Um dado valor ocorre numa dada lista?
 * @param lista - uma lista de valores
 * @param elemento - o valor a procurar
 * @return true se elemento ocorre em lista e false caso contrario
 */
private static boolean encontra (List<String> lista, String elemento)
```

Cada *característica* deve representar uma faceta do domínio de entrada que interessa explorar: o sinal do número, a cor do semáforo, o número de elementos na lista, por exemplo. Eis algumas guias para a escolha das características:

- Valores comumente utilizados;
- Valores fronteira;
- Relações entre os vários parâmetros de entrada;
- Relação entre os parâmetros e a funcionalidade do método.

No caso do método `encontra`, procuramos características para a `lista` e para o `elemento`. A pré-condição diz-nos que o primeiro parâmetro é do tipo lista. Olhando apenas para a *interface* do método, podemos utilizar a característica mais comum para as listas:

- A lista está vazia.

Pensando agora em termos da *funcionalidade* do método, podemos considerar, por exemplo, duas características:

- O número de ocorrências de `elemento` na lista, e
- O `elemento` ocorre na última posição da lista (o tratamento deste caso revela-se frequentemente crítico).

Depois de identificadas as características, é necessário particioná-las em conjuntos de valores, denominados *regiões*. Temos então de identificar as regiões e os valores representativos de cada uma delas. Note que mais regiões implicam mais testes, requerendo mais recursos (tempo para codificar e, mais tarde, para executar os testes), mas podem revelar mais falhas no programa. Menos regiões resultam em menos testes, e portanto numa poupança de recursos, podendo no entanto reduzir a eficácia dos testes.

Como o próprio nome indica, as várias regiões devem constituir uma *partição do espaço de entrada*, isto é, as regiões não se devem **sobrepor** e todas juntas devem **cobrir** o espaço completo. A título de exemplo, considere a característica *"número n de ocorrências do elemento na lista"*. Note que o espaço de todas as

possíveis ocorrências de um elemento na lista é dado pela inequação $n \geq 0$. As regiões $n > 1$ e $n > 2$ sobrepõem-se e como tal não podem ser tomadas em conjunto. Por outro lado, as regiões $n = 0$ e $n > 1$ não cobrem o espaço total. Contudo se se juntar a região dada por $n = 1$ obtém-se então uma partição do espaço.

No caso do método `encontra`, e de acordo com as características enunciadas acima, podemos identificar as seguintes regiões:

Característica	**Regiões**
A lista está vazia	v_1 = true, v_2 = false
Número de ocorrência na lista	$n_1 = 0$, $n_2 = 1$, $n_3 > 1$
Elemento ocorre na última posição da lista	p_1 = true, p_2 = false

Note que foram escolhidas três regiões para a característica *"Número de ocorrências na lista"*, mas poderíamos ter escolhido um outro número (duas ou quatro, por exemplo). A escolha está relacionada com a discussão de um dos parágrafos anteriores.

O último passo é identificar os casos de teste. Impõe-se então a questão: *"Como considerar as várias partições em simultâneo?"*. Por outras palavras, *"Que combinações de regiões vamos usar para escolher valores?"*.

Há muitos critérios para a escolha de combinações. Na disciplina de Laboratórios de Programação vamos apenas usar um: o de *cobertura de todas as combinações*.

No caso do método `encontra` identificámos três características com regiões $[v_1, v_2]$, $[n_1, n_2, n_3]$ e $[p_1, p_2]$. Fundamentalmente, precisamos então de, no máximo, 12 = 2 x 3 x 2 testes para cobrir todas as combinações possíveis das várias características. São eles:

(v_1, n_1, p_1), (v_1, n_1, p_2), (v_1, n_2, p_1), (v_1, n_2, p_2), (v_1, n_3, p_1), (v_1, n_3, p_2),

(v_2, n_1, p_1), (v_2, n_1, p_2), (v_2, n_2, p_1), (v_2, n_2, p_2), (v_2, n_3, p_1) e (v_2, n_3, p_2).

Está fácil de ver que, de um modo geral, a utilização deste critério pode tornar-se impraticável quando tiverem sido identificadas mais do que 2 ou 3 partições.

Um ponto algo subtil sobre o método de partição do espaço de entrada é que algumas combinações de regiões são *inviáveis*, isto é, nunca podem acontecer. Por exemplo, no caso do método `encontra`, não podemos combinar a região v_1 com a região n_2, porque por um lado é necessário uma lista vazia e por outro que a mesma lista contivesse uma ocorrência do elemento. De forma semelhante, não se pode combinar a região n_1 com a região p_2, porque construir uma lista com zero ocorrências do elemento, onde este ocorra na última posição é impossível. Usando o critério de cobertura de todas as combinações podemos construir uma tabela com as regiões e os valores de teste.

Identificação	Vazia	Ocorrências	Último	Teste	Resultado
(v_1, n_1, p_1)	True	0	True	inviável	
(v_1, n_1, p_2)	True	0	False	([ ], "c")	False
(v_1, n_2, p_1)	True	1	True	inviável	
(v_1, n_2, p_2)	True	1	False	inviável	

(v_1, n_3, p_1)	True	>1	True	inviável	
(v_1, n_3, p_2)	True	>1	False	inviável	
(v_2, n_1, p_1)	False	0	True	inviável	
(v_2, n_1, p_2)	False	0	False	([“a”, “b”], “c”)	False
(v_2, n_2, p_1)	False	1	True	([“a”, “c”], “c”)	True
(v_2, n_2, p_2)	False	1	False	([“a”, “c”, “b”], “c”)	True
(v_2, n_3, p_1)	False	>1	True	([“c”, “a”, “b”, “c”], “c”)	True
(v_2, n_3, p_2)	False	>1	False	([“a”, “c”, “c”, “b”], “c”)	True

Vemos então que, das 12 possíveis combinações, 6 são inviáveis restando somente as outras seis que dão, cada uma, origem a um caso de teste. Na tabela anterior usa-se a notação entre parênteses retos para indicar, por ordem, os elementos que compõem a lista.

```
//(v1,n1,p2)
ArrayList<String> lista1 = new ArrayList<>(Arrays.asList());
encontra(lista1, "c");
//Resultado: False

//(v2,n1,p2)
ArrayList<String> lista2 = new ArrayList<>(Arrays.asList("a", "b"));
encontra(lista2, "c");
//Resultado: False

//(v2,n2,p1)
ArrayList<String> lista3 = new ArrayList<>(Arrays.asList("a", "c"));
encontra(lista3, "c");
//Resultado: True

//(v2,n2,p2)
ArrayList<String> lista4 = new ArrayList<>(Arrays.asList("a", "c", "b"));
encontra(lista4, "c");
//Resultado: True

//(v2,n3,p1)
ArrayList<String> lista5 = new ArrayList<>(Arrays.asList("c", "a", "b", "c"));
encontra(lista5, "c");
//Resultado: True

//(v2,n3,p2)
ArrayList<String> lista6 = new ArrayList<>(Arrays.asList("a", "c", "c", "b"));
encontra(lista6, "c");
//Resultado: True
```

Identificação de características e de regiões

Na secção anterior identificámos algumas guias gerais para a escolha de características. No que se segue apresentam-se alguns casos concretos que ajudam na escolha das características e na identificação das correspondentes regiões para funções Java.

- Para cada parâmetro numérico x:

    - Não havendo restrições ao valor do número, escolhemos a característica que identifica o sinal do número (negativo, zero, positivo). Extraímos três regiões: $x < 0$, $x = 0$ e $x > 0$.
    - Se a pré-condição do método ditar que o parâmetro é, por exemplo, não negativo, então podemos extrair três regiões: $x = 0$, $x = 1$ e $x > 1$.
    - Se o contrato falar de algum valor n em especial, então uma escolha para três regiões será: $x < n$, $x = n$ e $x > n$.
    - Se a pré-condição atribuir limites n e m para o parâmetro x, então escolhem-se as regiões: $x = n$, $n < x < m$ e $x = m$.
- Para cada lista escolhemos a característica “lista vazia”, da qual resultam duas regiões: lista vazia e lista não vazia.
- Para cada dicionário analisamos as suas várias entradas.
- Para cada tuplo analisamos as suas componentes.
- Para cada resultado possível do método procedemos como se de um parâmetro se tratasse.
- Tomamos em consideração o nosso conhecimento da funcionalidade do método, bem como outras condições constantes no contrato do método.

Exemplos

Nesta secção analisamos alguns exemplos comuns.

Parâmetro numérico único

```
/**
 * O simetrico de um dado numero
 * @param x - numero do qual se pretende o simetrico
 * @return o simetrico de x
 */
public static double simetrico(double x)
```

Dado que não existem restrições ao parâmetro x, pode-se considerar a característica mais geral de número: o seu sinal (negativo, zero ou positivo). As regiões a considerar para esta característica são então x < 0, x = 0 e x >0. Note novamente que estas regiões formam uma partição do espaço. As regiões consideradas dão então origem a 3 testes.

Identificação	Sinal	Teste	Resultado
s_1	negativo	x = -2 .7	2.7
s_2	zero	x = 0	0
s_3	positivo	x = 3.1	-3.1

```
//s1
double x1 = -2.7;
simetrico(x1);
//Resultado: 2.7
```

```
//s2
double x2 = 0;
simetrico(x2);
//Resultado: 0
```

```
//s3
double x3 = 3.1;
simetrico(x3);
//Resultado: -3.1
```

Parâmetro numérico com restrições

```
/**
 * O factorial de um numero
 * @param n - numero do qual se pretende o factorial
 * @return factorial de n, i.e., n.(n-1). … 1
 * @requires n nao negativo
 */
public static int factorial(int n)
```

No caso apresentado, o número n do qual se pode calcular o factorial tem a restrição de ter de ser não negativo, isto é, n >= 0. A restrição ao parâmetro n impossibilita o uso da mesma característica do exemplo anterior. No entanto, podemos considerar outra característica como, por exemplo, a “dimensão do número”, para a qual podem ser geradas inúmeras regiões distintas. Pode-se, por exemplo, considerar as regiões n = 0, n = 1 e n > 1. Note novamente que estas regiões formam uma partição do espaço. As regiões consideradas dão origem a 3 testes.

Identificação	dimensão	Teste	Resultado
d_1	= 0	n = 0	1
d_2	= 1	n = 1	1
d_3	> 1	n = 5	120

```
//d1
int n1 = 0;
factorial(n1);
//Resultado: 1
```

```
//d2
int n2 = 1;
factorial(n2);
//Resultado: 1
```

```
//d3
int n3 = 5;
factorial(n3);
//Resultado: 120
```

Escolhas de acordo com a funcionalidade

```java
/**
 * A representacao hexadecimal de um numero entre 0 e 15
 * @param n - numero do qual se pretende o valor hexadecimal
 * @return o valor hexadecimal de n
 * @requires  0 <= n <= 15
 */
public static char hexadecimal (int n)
```

Novamente, para este exemplo, podemos considerar a característica “dimensão do número”. Note que podemos escolher regiões baseadas nos casos fronteira isto é, n = 0, 0 < n < 15 e n = 15. Contudo, como existe uma transição substancialmente diferente entre 9 e 10 e que pode gerar potencialmente falhas, podemos enriquecer a partição da característica considerada e incluir a transição. Geramos então a partição n = 0, 0 < n < 9, n = 9, n = 10, 10 < n < 15 e n = 15, gerando 6 casos de teste.

Identificação	dimensão	Teste	Resultado
d_1	= 0	n = 0	‘0’
d_2	> 0 e < 9	n = 5	‘5’
d_3	= 9	n = 9	‘9’
d_4	= 10	n = 10	‘A’
d_5	>10 e < 15	n = 13	‘D’
d_6	= 15	n = 15	‘F’

```java
//d1
int n1 = 0;
hexadecimal(n1);
//Resultado: ‘0’

//d2
int n2 = 5;
hexadecimal(n2);
//Resultado: ‘5’

//d3
int n3 = 9;
hexadecimal(n3);
//Resultado: ‘9’
```

```java
//d4
int n4 = 10;
hexadecimal(n4);
//Resultado: ‘A’

//d5
int n5 = 13;
hexadecimal(n5);
//Resultado: ‘D’

//d6
int n6 = 15;
hexadecimal(n6);
//Resultado: ‘F’
```

Combinando características

```
/**
 * Um dado inteiro nao negativo e' par?
 * @param n - numero do qual se pretende determinar se e' par
 * @return true sse n e' par,
 * @requires  n >= 0
 */
public static boolean ehPar (int n)
```

O exemplo serve para ilustrar como combinar parâmetros com resultados. Note que, à semelhança de um exemplo anterior, podemos considerar a característica "dimensão do número" ou então a característica "n é zero". Contudo, outra característica importante resulta da análise do resultado esperado do método, isto é, do facto de devolver um booleano. Daqui resulta a característica: "o método devolve true". Ficamos então com as características:

- n é zero: z_1 = true, z_2 = false;
- O método devolve true: t_1 = true, t_2 = false;

Note que para este caso a combinação das regiões, z_1 e t_2 é inviável dado que sendo n = 0, o resultado do método deverá ser `true` e nunca poderá ser `false`. Combinando:

Identificação	Zero	Retorno	Teste	Resultado
z_1 , t_1	`true`	`true`	n = 0	true
z_1 , t_2	`true`	`false`	inviável	
z_2 , t_1	`false`	`true`	n = 20	true
z_2 , t_2	`false`	`false`	n = 33	false

```
//z1, t1
int n1 = 0;
ehPar(n1);
//Resultado: true

//z2, t1
int n2 = 20;
ehPar(n2);
//Resultado: true

//z2, t2
int n3 = 33;
ehPar(n3);
//Resultado: false
```

Combinando mais características

```
/**
 * Dadas duas listas ordenadas devolve uma nova lista tambem
 * ordenada com a multiplicidade de elementos repetidos
 * @param lista1 - uma lista ordenada
 * @param lista2 - outra lista ordenada
 * @return uma lista ordenada com multiplicidade dos elementos repetidos
 * @requires  lista1 e lista2 estejam ordenadas
 */
public static List<Integer> fusao (List<Integer> lista1, List<Integer> lista2)
```

À semelhança do primeiro exemplo, da assinatura pode-se escolher as características: “a primeira lista está vazia” e “a segunda lista está vazia”. Contudo, dado que a lista a retornar deve ser ordenada, podemos considerar ainda muitas outras características relativas à relação entre as duas listas. Por exemplo, podemos considerar as características: “número de elementos em comum nas listas” e “relação entre máximos e mínimos das duas listas”. Combinando estas características temos as seguintes regiões:

Característica	Regiões
A primeira lista está vazia	$l1_1$ = true, $l1_2$ = false
A segunda lista está vazia	$l2_1$ = true, $l2_2$ = false
número de elementos comuns	$c_1 = 0$, $c_2 = 1$, $c_3 > 1$
Relação máximo e mínimo	r_1 = max (l1) < min (l2), r_2 = min (l2) < max (l1) < max (l2), r_3 = max (l2) < min (l1), r_4 = Outra

Note que para a característica “Relação entre mínimo e máximo” há necessidade de criar uma região designada de Outra. Porquê? Como se disse no ínicio deste documento, as regiões devem compor uma partição do espaço associado à característica, isto é, as regiões não se devem sobrepor e todas juntas devem cobrir todo o espaço. Note que as três primeiras regiões não cobrem o espaço: por exemplo, fica de fora o caso em que min(l1) < max(l2) < max(l1), bem como o caso em que o mínimo e o máximo não estão definidos (por a lista estar vazia). Na região “Outra” são considerados esses casos.

Como vimos, pelo critério de cobertura que utilizamos -- cobertura de todas as combinações-- seria necessário considerar 2 x 2 x 3 x 4 = 48 casos de teste. Na análise das regiões, acontece que grande parte das combinações não são viáveis. Por exemplo, não conseguimos construir um teste em que a lista1 é vazia e tem um elemento em comum com a lista2.

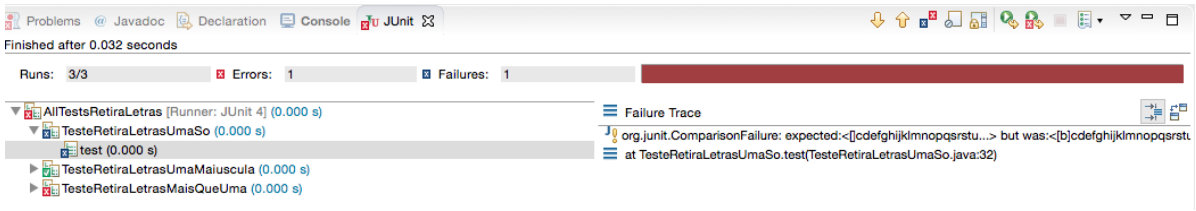
De modo a simplificar a construção da tabela que associa regiões a testes (e que identifica as combinações de regiões inviáveis), comecemos por pensar nas listas vazias. Note que quando ambas as listas são vazias a única região que é possível utilizar é a que diz que o número de elementos em comum é zero e que a relação min/max é “Outra”. Todas as outras combinações são inviáveis. E quando apenas a primeira lista é vazia? Neste caso só podemos contar com 0 elementos em comum e “Outra” para a relação min/max. O raciocínio é análogo para o caso em que a segunda lista é vazia. Analisamos depois cada um dos restantes 3 x 4 = 12 casos em que ambas as listas não são vazias. Eis uma possível tabela de testes.

Identificação	l1	l2	Com.	min/max	Teste	Resultado
($l1_1$, $l2_1$, c_1, r_4)	t	t	0	Outra	([ ], [ ])	[ ]
($l1_1$, $l2_2$, c_1, r_4)	t	f	0	Outra	([2,3,3], [ ])	[2,3,3]
($l1_2$, $l2_1$, c_1, r_4)	f	t	0	Outra	([ ], [7,24])	[7,24]
($l1_2$, $l2_2$, c_1, r_4)	f	f	0	$\max(l_1) < \min(l_2)$	([3,7], [9,17])	[3,7, 9,17]
($l1_2$, $l2_2$, c_1, r_1)	f	f	0	$\min(l_2) < \max(l_1) < \max(l_2)$	([2,4],[3,9])	[2,4,3,9]
($l1_2$, $l2_2$, c_1, r_2)	f	f	0	$\max(l_2) < \min(l_1)$	([2,37, 37],[1])	[1,2,37,37]
($l1_2$, $l2_2$, c_1, r_3)	f	f	0	Outra	([3,9], [2,4])	[2,3,4,9]
($l1_2$, $l2_2$, c_2, r_4)	f	f	1	$\max(l_1) < \min(l_2)$	inviável	
($l1_2$, $l2_2$, c_2, r_2)	f	f	1	$\min(l_2) < \max(l_1) < \max(l_2)$	([4],[2,4,8])	[2,4,4,8]
($l1_2$, $l2_2$, c_2, r_3)	f	f	1	$\max(l_2) < \min(l_1)$	inviável	
($l1_2$, $l2_2$, c_1, r_4)	f	f	1	Outra	([2,4,8],[4])	[2,4,4,8]
($l1_2$, $l2_2$, c_3, r_1)	f	f	>1	$\max(l_1) < \min(l_2)$	inviável	
($l1_2$, $l2_2$, c_3, r_2)	f	f	>1	$\min(l_2) < \max(l_1) < \max(l_2)$	([1,4],[1,4,8])	[1,1,4,4,8]
($l1_2$, $l2_2$, c_3, r_3)	f	f	>1	$\max(l_2) < \min(l_1)$	inviável	
($l1_2$, $l2_2$, c_3, r_4)	f	f	>1	Outra	([1,4,8],[1,4])	[1,1,4,4,8]

O Uso de JUnit

Ao longo do semestre teve oportunidade de usar testes unitários através do JUnit. No que diz respeito à cadeira de LabP, o intuito dos testes foi sempre o de serem usados e não programados. Contudo, é relevante reforçar alguns pontos dos quais foi tomando consciência ao longo do semestre.

Tal como explicado no guião inicial sobre o IDE Eclipse, existem 3 resultados possíveis para cada um dos testes realizados: “ok” que significa que correu sem problemas, isto é, o valor que é obtido por correr o método é igual ao valor que era esperado (assinalado com um visto a verde); “erro” (assinalado com um “x” vermelho) que significa que ocorreu um erro durante a tentativa de execução do teste (normalmente uma exceção não tratada pelo teste); ou “falha” (assinalada com um “x” azul) que significa que o teste correu até ao fim mas que o resultado esperado não é o que foi obtido ao correr o método. Recorde que do lado direito da consola aparece uma descrição sucinta do erro ou da falha que o teste detetou.



A estrutura de uma classe de teste

Consideremos o seguinte exemplo da classe Operador:

```
public class Operador {

    double valor;

    /**
     * construtor da classe
     */
    public Operador(double inicial){
        this.valor = inicial;
    }

    /**
     * A soma de um valor dado com o valor
     * do operador
     * @param a – o valor a somar
     */
    public double soma(double a){
        this.valor += a;
        return this.valor;
    }
}
```

Para testarmos o método `soma` vamos comparar o seu resultado com o resultado esperado. Como é uma soma de números do tipo `double`, a comparação não pode ser feita com `==`. Vamos usar um método da biblioteca do JUnit que permite fazer este tipo de comparações entre `doubles`.

Uma possível classe de teste para o método `soma` é a seguinte:

```
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class OperadorTest {

    private  Operador op;

    public static final double DELTA = 0.001;

    @Before
    public void setUp() {
        op = new Operador(2.0);
    }

    @After
    public void tearDown() {
    }

    @Test(timeout=1000)
    public void testSomaPositivo () {
        double a =  3.0;

        double expected = 5.0;
```

```
        double obtained = op.soma(a);

        assertEquals(expected , obtained, DELTA);
    }

    @Test(timeout=1000)
    public void testSomaZero () {
        double a =  0.0;

        double expected = 2.0;
        double obtained = op.soma(a);

        assertEquals(expected , obtained, DELTA);
    }

    @Test(timeout=1000)
    public void testSomaNegativo () {
        double a =  -3.0;

        double expected = -1.0;
        double obtained = op.soma(a);

        assertEquals(expected , obtained, DELTA);
    }
}
```

Note que é feito o *import* do pacote `Assert` (Afirmação) que permite especificar em linguagem própria se uma dada afirmação é verdadeira ou falsa ou se dois artefactos são iguais.

A classe usa um atributo de tipo `Operador`. Note que para cada teste devemos reiniciar o objeto `Operador op`. Isso é feito através do método `setUp` cuja anotação `@Before` indica exatamente isso. É corrido antes de qualquer um dos casos de teste, colocando o `Operador op` com o valor 2.0.[1]

O método `tearDown` (deixado propositadamente vazio) com a anotação `@After` serve para no final de cada teste, caso haja necessidade repor ou alterar o objeto criado para os testes.

Os casos de teste são anotados com `@Test`. A opção `(timeout = 1000)` permite estabelecer um tempo máximo que o método tem para finalizar. Caso esse tempo seja excedido, é declarado que o método falha.

Nos testes propriamente ditos usa-se uma nomenclatura do pacote `Assert`. Neste pacote existem vários métodos com utilidades diferenciadas. Ao longo do semestre demos especial ênfase ao `assertTrue` e `assertEquals`. Estes dois métodos permitem comparar se o valor obtido é igual ao esperado. Repare que o método `assertEquals` que aqui usamos tem um parâmetro (***DELTA***) que define o erro permitido na comparação.

Exemplo de teste JUnit baseado na partição do espaço de entrada

Consideramos um exemplo concreto de teste do método `encontra` introduzido na página 3. Disponibilizamos duas versões deste método: uma certa e uma com um *bug*. O objetivo é mostrar como a utilização da metodologia da partição do espaço de entrada conjuntamente com a bateria

[1] Existe um outro método `setUpClass` com a anotação `@BeforeClass` que é utilizado para escrever código necessário para estabelecer o contexto de teste. Ao contrário do `@Before`, este apenas é corrido uma vez para todos os testes.

de testes unitários implementados em JUnit permitem identificar individualmente erros de implementação. Considere a classe `RunEncontra` a seguir:

```
import java.util.ArrayList;

public class RunEncontra {

    /**
     * Um dado valor ocorre numa dada lista?
     * @param lista - uma lista de valores
     * @param elemento - o valor a procurar
     * @return true se elemento ocorre em lista e false caso contrario
    */
    public static boolean encontra (List<String> lista, String elemento){
        int i = 0;
        boolean encontrado = false;
        while (i < lista.size() && !encontrado){
               if (lista.get(i).equals(elemento)){
                      encontrado = true;
               }
               i++;
        }
        return encontrado;
    }

    /**
    * variante errada do metodo acima
    *  @param  lista  -  uma lista de valores
    *  @param  elemento  -  o valor a procurar
    *  @return  true de elemento ocorre na lista e false caso contrario
    */
    public static boolean buggedEncontra (List<String> lista, String elemento){
        int i = 0;
        while (i < lista.size() && !lista.get(i).equals(elemento)){
               i++;
        }
        return i <= lista.size();
    }

    public static void main (String[] args){
        System.out.println("Para testar as funções execute os tests junit");
    }
}
```

Consegue identificar o erro no método `buggedEncontra`? O problema é a condição no `return`, e o erro acontece quando a lista não contém o elemento. Na verdade, esta implementação do método é equivalente ao código *return true*.

Apanhamos o erro utilizando uma implementação JUnit dos testes baseados na partição do espaço de entrada listada na página 5:

```
import static org.junit.Assert.assertEquals;

import java.util.ArrayList;
import java.util.Arrays;

import org.junit.Test;

public class TesteEncontra {
```

```java
@Test
//(v1,n1,p2)
public void runEncontra1() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList());
    assertEquals(RunEncontra.encontra(lista, "c"), false);
}

@Test
//(v1,n1,p2)
public void runBuggedEncontra1() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList());
    assertEquals(RunEncontra.buggedEncontra(lista, "c"), false);
}

@Test
//(v2,n1,p2)
public void runEncontra2() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList("a", "b"));
    assertEquals(RunEncontra.encontra(lista, "c"), false);
}

@Test
//(v2,n1,p2)
public void runBuggedEncontra2() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList("a", "b"));
    assertEquals(RunEncontra.buggedEncontra(lista, "c"), false);
}

@Test
//(v2,n2,p1)
public void runEncontra3() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList("a", "c"));
    assertEquals(RunEncontra.encontra(lista, "c"), true);
}

@Test
//(v2,n2,p1)
public void runBuggedEncontra3() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList("a", "c"));
    assertEquals(RunEncontra.buggedEncontra(lista, "c"), true);
}

@Test
//(v2,n2,p2)
public void runEncontra4() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList("a", "c", "b"));
    assertEquals(RunEncontra.encontra(lista, "c"), true);
}

@Test
//(v2,n2,p2)
public void runbuggedEncontra4() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList("a", "c", "b"));
    assertEquals(RunEncontra.buggedEncontra(lista, "c"), true);
}

@Test
//(v2,n3,p1)
public void runEncontra5() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList("c", "a", "b", "c"));
    assertEquals(RunEncontra.encontra(lista, "c"), true);
}

@Test
//(v2,n3,p1)
public void runbuggedEncontra5() {
    ArrayList<String> lista = new ArrayList<>(Arrays.asList("c", "a", "b", "c"));
    assertEquals(RunEncontra.buggedEncontra(lista, "c"), true);
```

```
    }

    @Test
    //(v2,n3,p2)
    public void runEncontra6() {
        ArrayList<String> lista = new ArrayList<>(Arrays.asList("a", "c", "c", "b"));
        assertEquals(RunEncontra.encontra(lista, "c"), true); }
    @Test
    //(v2,n3,p2)
    public void runbuggedEncontra6() {
        ArrayList<String> lista = new ArrayList<>(Arrays.asList("a", "c", "c", "b"));
        assertEquals(RunEncontra.buggedEncontra(lista, "c"), true);
    }

}
```

Concluímos mostrando o *output* dos testes que evidencia o problema da implementação *errada* nos casos (v_1,n_1,p_2) e (v_2,n_1,p_2). Para solucionar o problema do método *buggedEncontra* basta que a sua última linha passe a ser `return i < lista.size();` tomando em consideração a importância da característica "último elemento da lista". Verifique, adaptando os testes, que o método funciona corretamente depois desta alteração.