DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Uma Introdução ao JUnit (3.8)

Nuno Mamede

João D. Pereira

Outubro de 2012

Este documento tem como objectivo introduzir a necessidade de realização de testes de software e a sua concretização utilizando a *framework* de testes JUnit.

Testes de Software é uma área de Engenharia de Software. A Engenharia de Software preocupa-se com o o processo de desenvolvimento de software. Um dos aspectos da Engenharia de Software é garantir a qualidade do software a desenvolver. Um programa desenvolvido tem qualidade se ele fizer aquilo que é suposto fazer e não tiver *bugs*. Actualmente, a abordagem mais utilizada para garantir a qualidade de um programa é através da realização de testes de software. Os teste de software podem ser manuais ou automáticos. Testes de software manuais exigem a intervenção humana. Os testes de software automáticos não exigem a intervenção humana e correspondem à execução de um programa (que corresponde ao teste de software a realizar) que vai verificar se outro programa tem o comportamento esperado ou não.

Os testes de software são constituídos por *casos de teste*. Cada caso de teste verifica se o software a testar tem um dado comportamento esperado ou não. Se o software não tiver o comportamento esperado, então encontrou-se um bug[1]. Os casos de teste podem ser agrupados em *test suites* (baterias de teste). Por exemplo, pode-se constituir uma bateria de teste que agrupa todos os casos de teste que dizem respeito a uma dada classe ou a um dado método.

No caso de software desenvolvido com linguagens de programação com objectos, um caso de teste tem aquilo a que se designa como o *objecto alvo*. O objecto alvo de um caso de teste é o objecto sobre o qual se quer verificar que tem um dado comportamento esperado. Um caso de teste deve então verificar se a invocação de um dado método do objecto alvo numa dada condição tem o comportamento/efeito esperado. A condição do caso de teste é constituída pelo estado do objecto alvo antes da invocação do método a testar, pelos valores dos argumentos do método a testar e pelo estado do sistema. Para verificar que um método teve o comportamento/efeito esperado é necessário comparar o resultado obtido com o resultado esperado. O resultado a comparar pode incluir o valor de retorno do método a testar (caso exista), o estado do objecto alvo após a invocação do método e ainda o estado do sistema após a invocação do método a testar.

O **Junit** é um *framework* de testes escrito por Erich Gamma e Kent Beck que facilita a implementação de unidades de teste em *Java*. O **JUnit** é *open source* e oferece um conjunto de classes permitindo a fácil integração e execução regular

[1] Ou o caso de teste está mal concretizado.

de testes durante o processo de desenvolvimento, de forma a ter uma medida mais concreta do seu progresso.

Em `http://www.junit.org/`, poderá fazer o *download* da versão mais recente de **Junit**. Após guardar o ficheiro para o seu disco, descomprima-o numa pasta à sua escolha e está concluída a instalação. Eventualmente, poderá ser conveniente incluir o ficheiro **junit.jar** na variável de ambiente `classpath`.

Utilizando a framework de testes **JUnit**, cada caso de teste deve ser concretizado como um método de uma classe de teste. As classes de teste têm que herdar da classe `junit.framework.TestCase` definida na framework **JUnit**. Esta classe fornece um conjunto de métodos e mecanismos que facilitam a realização de testes de software automáticos.

Na versão 3 do **JUnit**, o nome de todos os métodos de uma classe de teste que correspondam a casos de teste devem começar por `test`. A restante parte do nome deve indicar qual é o objectivo do teste concretizado pelo método em causa. A framework **JUnit** executa de forma automática todos os métodos de teste definidos numa classe de teste. Não está definido qual é a ordem pela qual os métodos de teste são executados. Além disso, o facto de um método de teste encontrar ou não um bug, não deve ter influência na execução dos testes seguintes. Devido a estas duas razões, cada método de teste deve ser completamente independente dos restantes.

A parte final de um método de teste tem como objectivo verificar que o resultado obtido após a execução do método a testar é igual ao resultado esperado. Quando o valor obtido é diferente do esperado, isso quer dizer que o teste potencialmente encontrou um bug no código que está a ser testado. Quando isto acontece, a execução do método de teste que encontrou o bug é interrompida (não é executado o código do método de teste que esteja a seguir à comparação que falhou), o método de teste é marcado como um teste que falhou e a framework **JUnit** executa o método de teste seguinte.

Por forma a comparar o resultado obtido após a execução do comportamento a verificar com o resultado esperado, a classe `TestCase` fornece um conjunto de métodos que facilitam a comparação dos dois tipos de resultados. Tipicamente estes métodos começam com `assert` e permitem especificar uma mensagem de erro, o valor esperado e o o valor obtido:

- **assertTrue([String message], boolean condition)** Verifica se o argumento boleano (que deve representar o valor obtido) tem o valor `true`. O primeiro

argumento pode ou não ser indicado e representa a mensagem de erro a ser apresentada ao utilizador caso a verificação falhe.

- **assertFalse([String message], boolean condition)** Semelhante ao caso anterior mas em que se espera que o valor obtido seja igual a `false`.
- **assertsEquals([String message], esperado, obtido)** Verifica se os dois parâmetros `esperado` e `obtido` são iguais. Caso o tipo dos dois parâmetros seja `Object`, então a verificação da igualdade é realizada através da invocação do método equals. Nota, no caso em que os dois parâmetros são vectores, a comparação realizada é se os dois parâmetros referenciam o mesmo vector, não se verifica se o conteúdo dos dois vectores referenciados pelos dois parâmetros são iguais. É de notar que os valores que queremos comparar com este método devem estar na seguinte sequência: Primeiro o valor que estamos à espera de encontrar e em seguida o valor obtido após a invocação do método a testar. Assim, caso exista uma falha no teste, a mensagem de erro indicará corretamente esta mesma informação.
- **assertNull([String message], Object object)** Verifica se o argumento `object` tem o valor `null`. A verificação falha caso o argumento seja diferente de `null`.
- **assertNotNull([String message], Object object)** Realiza a verificação oposta do caso anterior. A verificação falha case o argumento `object` seja igual a `null`.
- **fail([String message])** Termina a execução do método de teste, indicando que o teste falhou. Normalmente, este método é utilizado para verificar que uma determinada zona do código não deve ser atingida (é útil quando se quer testar métodos que podem lançar excepções como veremos na seção 6). Também pode ser utilizado para ter um método de teste a falhar enquanto não está concretizado.

Cada comportamento distinto de um dado método deve ser exercitado num caso de teste distinto. Poder-se-ia exercitar vários comportamentos distintos de um dado método no mesmo caso de teste mas isso teria a desvantagem de que caso a verificação de um dado comportamento falhasse, a parte restante do teste já não seria executada e os restantes comportamentos ainda não verificados não seriam exercitados enquanto o bug não fosse corrigido.

Cabe ao programador dos testes elaborar testes que abranjam o maior número de situações distintas para garantir uma melhor fiabilidade do seu código. O que pode parecer trabalho inútil para programas tão simples, pode levar à deteção de

erros que, de outro modo, quando programas mais complexos integrassem estas classes, levariam muito mais tempo a depurar.

Para ilustrar a concretização de casos de testes utilizando o **JUnit** vai ser utilizado o seguinte exemplo. Imagine que pretende representar uma classe que defina o funcionamento das portas. Uma porta pode ser aberta ou fechada. Ações redundantes (abrir uma porta já aberta ou fechar uma porta já fechada) não alterarão o estado da porta. Escreva, no ficheiro `Porta.java` (assumindo que está na pasta `/javacode/porta/`), a seguinte classe:

```
public class Porta {
    private boolean _fechada = true;

    public Porta() {
        _fechada = true;
    }

    public Porta(boolean fechada) {
        _fechada = fechada;
    }

    public boolean estaAberta() {
        return !_fechada;
    }

    public void abrir() {
        if (_fechada)
            _fechada = false;
    }

    public void fechar() {
        if (!_fechada)
            _fechada = true;
    }
}
```

Agora convém testar se a classe `Porta` que escrevemos está a funcionar corretamente. Para criar uma unidade de testes através do `JUnit` basta criar uma subclasse de `TestCase`. Vamos chamar-lhe `TestePorta`. Esta classe vai testar se o comportamento de uma porta é o pretendido. Escreva, no ficheiro `TestePorta.java` (assumindo que está na pasta `/javacode/porta/`), o seguinte código:

```
import junit.framework.TestCase;

public class TestePorta extends TestCase {

    private Porta _porta;
```

```
    public static void main(String[] args) {
        junit.textui.TestRunner.run(TestePorta.class);
    }

    public void setUp() {
        _porta = new Porta();
    }

    public void testConstrutorSemArgumentos() {
        assertFalse(_porta.estaAberta());
    }

    public void testAbrirPortaFechada() {
        _porta.abrir();
        assertTrue("A porta devia estar aberta", _porta.estaAberta());
        assertEquals("A porta devia estar aberta", true, _porta.estaAberta());
    }

    public void testAbrirPortaAberta() {
        Porta portaAberta = new Porta(false);

        portaAberta.abrir();

        assertEquals("A porta devia estar aberta", true, portaAberta.estaAberta());
    }

    public void testFecharPortaFechada() {
        _porta.fechar();

        assertEquals("A porta devia estar fechada", false, _porta.estaAberta());
    }

    public void testFecharPortaAberta() {
        Porta portaAberta = new Porta(false);

        _porta.fechar();

        assertFalse("A porta devia estar fechada", _porta.estaAberta());
    }
}
```

Note-se que, como fazemos referência à classe `TestCase`, é necessário importar esta classe definida no *package* `junit.framework`. Foi definido o método `main` de modo a que para executar os testes baste apenas executar a classe `TestePorta`.

A nossa classe de testes tem cinco testes (os métodos cujos nomes começam por `test`) que verificam o comportamento dos diferentes métodos da classe `Porta`. Por exemplo, para testar o método `fechar` vamos considerar as duas situações distintas em que este método pode ser invocado e verificar se o método tem o comportamento esperado em cada uma delas:

- fechar uma porta fechada não deve alterar o estado da porta. Isto é verificado no método `testFecharPortaFechada`.

- fechar uma porta aberta deve alterar o estado da porta, a qual deve passar a estar fechada. Isto é verificado no método `testFecharPortaAberta`.

No método de teste `testAbrirPortaFechada` estão a ser invocados dois métodos `assert` apenas para exemplificar que estes métodos são equivalentes. Bastava invocar apenas um deles, tal como é feito nos restantes métodos desta classe de teste. O método `assertTrue` é construído à custa do método `assertEquals`.

Por vezes existe algum código que se encontra repetido nos métodos de teste e que está relacionado com a construção do objecto alvo e em colocá-lo num determinado estado relevante para a situação a testar. Por forma a evitar esta repetição de código, a framework **JUnit** permite a refactorização deste código no método `setUp` que pode ser definido na classe de teste. Este método já está definido na classe `TestCase`,mas por omissão não faz nada. A framewotk **JUnit** garante que este método é sempre invocado antes de cada método de teste. Na classe `TestePorta` este método cria uma instância da classe `Porta` que representa uma porta aberta e coloca uma referência para esta instância no atributo `_porta`.

Por vezes, a parte final dos testes também pode aparecer com alguma repetição de código. Tipicamente isto acontece quando é necessário desfazer o efeito da invocação do método a testar no estado do sistema por forma a que a ordem de execução dos casos de testes não tenha impacto no sucesso/insucesso dos casos de teste. Os casos de teste devem ser independentes uns dos outros por forma a que o facto de um caso de teste falhar isso não provoque que os casos de teste seguintes passem a falhar também. Este código repetido pode ser colocado no método `tearDown` da classe de teste. Este método também se encontra definido na classe `TestCase` e caso não seja substituído não faz nada. A framework **JUnit** invoca este método após a execução de cada método de teste. Resumindo, para cada método de teste de uma classe de teste, a framework **JUnit** invoca o método `setUp`, o método de teste e o método `tearDown` por esta ordem.

Se estiver na pasta `/javacode/porta/` pode compilar o código da seguinte forma:

```
javac -classpath /junit3.8/junit.jar:. *.java</PRE>
```

Notas:

- Substitua `/junit3.8/` pelo caminho onde instalou o **Junit**;

- Se o **Junit** já estiver presente na variável de ambiente *CLASSPATH* não necessita de utilizar o parâmetro `-classpath` da **JVM** (Java Virtual Machine).

A execução deste comando com sucesso criará dois novos ficheiros, `Porta.class` e `TestPorta.class`, no mesmo diretório onde o comando foi executado. Para testar o correto comportamento da classe `Porta` é necessário executar os casos de teste concretizados na classe `TestePorta`. A execução dos casos de teste é realizado através da classe `junit.textui.TestRunner`, dando-lhe como parâmetro a classe de teste `TestePorta`. Isto pode ser feito da seguinte forma:

```
java -classpath /javacode/porta:/junit3.8/junit.jar junit.textui.TestRunner TestePorta
```

ou executando o método `main` da classe `TestePorta`, o qual foi programado para invocar o método `run` da classe `junit.textui.TestRunner`, passando-lhe como argumento um objeto que representa a classe de teste a correr, `TestaPorta` no nosso exemplo:

```
java -classpath /javacode/porta:/junit3.8/junit.jar TestePorta
```

Se tudo tiver sido bem feito o resultado deverá ser:

```
...

Time: 0
OK (2 tests)
```

Note que nem todos os métodos da classe `Porta` estão a ser testados. Falta testar o construtor da classe que aceita um argumento do tipo boleano. Como exercício de aplicação acrescente dois métodos de teste à classe `TestePorta` que testam este construtor, compile da classe de teste de novo e corra os testes, tal como foi feito anteriormente.

Os métodos a testar podem também lançar excepções. A verificação do comportamento esperado de um método que pode lançar excepções é realizada de forma ligeiramente diferente da de um método que não lança excepções. Além de verificar se o resultado obtido é igual ao esperado, o caso de teste tem também que

considerar se o método lançou ou não uma dada excepção e se isso corresponde a um erro ou não. Por exemplo, se o comportamento a testar deve implicar que o método deve lançar uma excepção e esta não ocorre então encontrou-se um *bug*.

Para ilustrar a realização de casos de teste considerando métodos que lançam excepções vamos considerar o seguinte exemplo. A classe `PortaComTranca` representa uma porta com uma tranca. Esta classe estende a classe `Porta`, acrescentando dois métodos novos: `trancar` e `destrancar`. Só é possível trancar uma porta que esteja fechada e destrancada. Só é possível destrancar uma porta que esteja fechada e trancada. Em ambos os casos, caso não seja possível realizar a operação pretendida então é lançada uma excepção, `NaoPossoTrancarPortaException` no primeiro caso e `NaoPossoDestrancarPortaException` no segundo.

```
public class PortaComTranca extends Porta {
    private boolean _trancada = true;

    public PortaComTranca(boolean trancada) {
         _trancada = trancada;
    }

    public void abrir() {
        if (!_trancada)
            super.abrir();
    }

    public boolean estaTrancada() {
        return _trancada;
    }

    public void destrancar() throws NaoPossoDestrancarPortaException{
if (_trancada)
  _trancada = false;
          else
            throw new NaoPossoDestrancarPortaException();
    }

    public void trancar() throws NaoPossoTrancarPortaException {
        if (!estaAberta() && !_trancada)
            _trancada = true;
        else
          throw new NaoPossoTrancarPortaException();
    }
}
```

As classes de excepção utilizadas neste exemplo são as seguintes:

```
public class NaoPossoDestrancarPortaException extends Exception {
    public NaoPossoDestrancarPortaException() {
    }
}
```

e

```
public class NaoPossoTrancarPortaException extends Exception {
    public NaoPossoTrancarPortaException() {
    }
}
```

A verificação do comportamento esperado no que diz respeito ao lançamento ou não de uma dada excepção é realizada através do método `fail` definido na classe `TestCase`, e portanto disponível em qualquer classe de teste, Existem duas versões deste método, uma sem argumentos e outra que recebe um argumento que representa a mensagem de erro a apresentar ao utilizador. Este método serve para indicar que a execução de um dado caso de teste falhou (equivale à instrução `assertTrue(false)`). Este método pode ser útil quando queremos indicar que se chegarmos a uma determinada linha no nosso código de testes é porque algo falhou.

Para que possamos testar se um dado método lança ou não uma exceção numa determinada situação basta que essa situação seja simulada e que se tente apanhar a eventual exceção através de um bloco de `try-catch`. Se a excepção não devia ocorrer basta que se coloque um `fail` no bloco de código associado ao `catch`, onde a exceção será tratada. Caso contrário, se a excepção deve ocorrer, basta que se coloque um `fail` na linha imediatamente a seguir à instrução onde se invoca o método a testar (e que não deveria lançar a excepção na situação exercitada neste caso de teste) sobre o objecto alvo, ainda dentro do bloco de código associado ao `try`.

A seguinte classe permite testar se os novos métodos `trancar` e `destrancar` definidos na classe `PortaComTranca` têm o comportamento esperado ou não.

```
import junit.framework.TestCase;

public class TestePortaComTranca extends TestCase {

  public static void main(String[] args) {
       junit.textui.TestRunner.run(TestePortaComTranca.class);
  }

  public void testTrancarPortaFechadaDestrancada() {
      PortaComTranca porta = new PortaComTranca(false);

      try {
          porta.trancar();
      } catch (NaoPossoTrancarPortaException exec) {
           fail("Não foi possível trancar uma porta fechada e destrancada");
      }

      assertFalse(porta.estaAberta());
      assertTrue(porta.estaTrancada());
  }

  public void testTrancarPortaFechadaTrancada() {
```

```
    PortaComTranca porta = new PortaComTranca(true);

    try {
        porta.trancar();
        fail("Não deve ser possível trancar uma porta trancada");
    } catch (NaoPossoTrancarPortaException exce) {
    }

    assertFalse(porta.estaAberta());
    assertTrue(porta.estaTrancada());
}

public void testTrancarPortaAberta() {
    PortaComTranca porta = new PortaComTranca(false);
    porta.abrir();

    try {
        porta.trancar();
        fail("Não deve ser possível trancar uma porta aberta");
    } catch (NaoPossoTrancarPortaException exce) {
    }

    assertTrue(porta.estaAberta());
    assertFalse(porta.estaTrancada());
}

public void testDestrancarPortaFechadaDestrancada() {
    PortaComTranca porta = new PortaComTranca(false);

    try {
        porta.destrancar();
        fail("Não deve ser possível destrancar uma porta fechada e destrancada");
    } catch (NaoPossoDestrancarPortaException exce) {
    }

    assertFalse(porta.estaAberta());
    assertFalse(porta.estaTrancada());
}

public void testDestrancarPortaFechadaTrancada() {
    PortaComTranca porta = new PortaComTranca(true);

    try {
        porta.destrancar();
    } catch (NaoPossoDestrancarPortaException exce) {
        fail("Não foi possível destrancar uma porta fechada e trancada");
    }

    assertFalse(porta.estaAberta());
    assertFalse(porta.estaTrancada());
}

public void testDestrancarPortaAberta() {
    PortaComTranca porta = new PortaComTranca(false);
    porta.abrir();

    try {
        porta.destrancar();
        fail("Não deve ser possível destrancar uma porta aberta");
    } catch (NaoPossoDestrancarPortaException exce) {
    }
```

```
        assertTrue(porta.estaAberta());
        assertFalse(porta.estaTrancada());
    }
}
```

Como se pode ver pela análise dos métodos de teste, o método `fail` é invocado dentro do bloco `try` nas situações em que a excepção não deve ser lançada e é invocado dentro do bloco `catch` nos casos em que não deve ocorrer nenhuma excepção.

O **JUnit** vem também acompanhado de interfaces gráficas (`junit.awtui.TestRunner` e `junit.swingui.TestRunner`) para apresentar os resultados dos testes corridos. Para usar uma destas interfaces gráficas em vez da interface textual basta correr os testes utilizando uma destas classes em vez da classe `junit.textui.TestRunner` utilizada anteriormente para demostrar a execução dos testes **JUnit**. Para mais informação sobre a utilização, destas interfaces gráficas consulte o ficheiro `doc\cookbook\cookbook.htm` acessível a a partir da pasta de instalação.

Uma nota final: a maioria dos IDE `Java` permite configurar pastas de bibliotecas `Java` para cada aplicação e utilizá-las de forma transparente no processo de criação da mesma, evitando a configuração da variável `CLASSPATH`. É algo que merece ser investigado pois pode facilitar o seu processo de trabalho. No entanto, deverá conhecer os procedimentos acima para os casos em que apenas tiver uma linha de comando num terminal não gráfico à sua disposição.

O arquivo de instalação contêm um conjunto de documentos, tutoriais e exemplos valiosos para a compreensão do `framework`. Inclui também a descrição completa da API. Após correr o sua primeira unidade de testes, investigue como ligar hierarquicamente várias unidades de testes e como ativar um particular teste de um conjunto de unidades.