Laboratórios de Informática 2016–2017

DETI - Universidade de Aveiro

Créditos Estes guiões constituem um trabalho continuado de refinamento que contou com a colaboração de João Paulo Barraca, Diogo Gomes, André Zúquete, João Manuel Rodrigues, António Adrego da Rocha, Tomás Oliveira e Silva, Sílvia Rodrigues e Óscar Pereira.

TEMA 14

Testes e Depuração

Objetivos:

- Test Driven Development
- Testes Unitários
- Testes Funcionais
- Depuração

14.1 Introdução

A validação do software é vital para que as equipas consigam realizar entregas dos componentes que desenvolvem, com alguma garantia do funcionamento do código em causa. Ao realizar testes de forma continuada e logo desde o início do desenvolvimento, é possível construir aplicações mais robustas e previsíveis. A existência de testes sistemáticos permite aferir o progresso e detetar regressões no desenvolvimento de uma aplicação. Quando se detetam problemas, é necessário encontrar a razão da anomalia, o que normalmente se faz através de depuração interativa e revisão do código desenvolvido. Este guião irá abordar ambos os temas, na perspectiva de pequenas equipas, ou programadores isolados, que pretendem desenvolver aplicações funcionais.

14.2 Desenvolvimento guiado por testes

A metodologia Test Driven Development (TDD) tem ganho adeptos nos últimos anos. Adota uma abordagem alternativa perante o desenvolvimento de aplicações, dando um ênfase especial à definição e execução de testes ao software. Considera-se que não é possível determinar qual o estado de uma aplicação se ela não for testada em todos os seus componentes. Sem testes sistemáticos é possível que alguns problemas passem despercebidos e venham a provocar problemas no futuro.

Segundo a TDD, o desenvolvimento de uma nova funcionalidade deve começar pela definição de testes à funcionalidade desejada, ainda antes de existir qualquer código. Claro que inicialmente todos os testes irão falhar, pois as funcionalidades não estão implementadas. A este estado dá-se o nome de *RED* (vermelho). À medida que se implementa cada funcionalidade e o teste respetivo é satisfeito, diz-se que estado da funcionalidade passa para *GREEN* (verde). De forma a tornar a aplicação consistente e evitar que se torne uma colagem de funcionalidades, depois de cada teste deverá ser realizada uma análise do que foi produzido e harmonização da solução, a fase *REFACTORING*. Este processo está representado na Figura 14.1. Quando todos os testes forem bem sucedidos, considera-se que a aplicação possui a funcionalidade desejada, para todos os casos de utilização previstos.

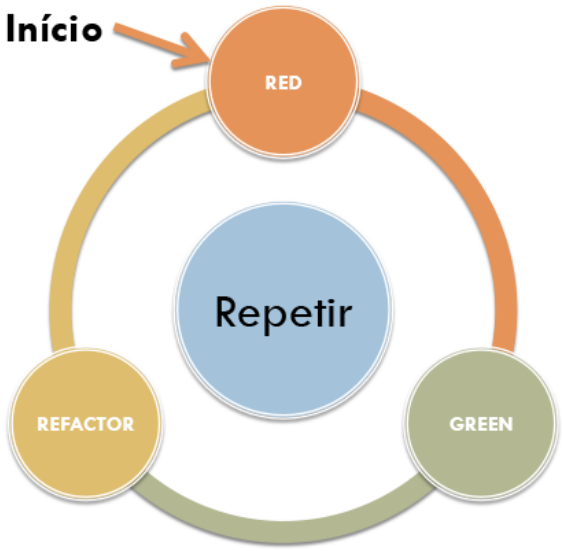

Figura 14.1: Fluxo de processos na metodologia TDD.

Esta metodologia é completamente independente da linguagem de programação, e pode implementar-se sem qualquer ferramenta específica. No entanto, na prática é comum

utilizar ferramentas como *Jenkins*,[1] que de forma pontual executa testes a aplicações, permitindo acompanhar de forma detalhada qual o estado de desenvolvimento da aplicação. Permite igualmente que os programadores evitem um erro muito comum: começar a implementar código da primeira solução, ignorando todos os outros casos que o algoritmo também tem de considerar. Permite igualmente que se pense no problema de forma objetiva e em como será implementado, de uma forma mais distante e considerando o que seria a arquitetura ótima da solução. Começar imediatamente a programar implica que a solução final irá apenas realizar parte do que é pedido, os módulos irão comportar-se da maneira que dá mais jeito ao programador enquanto desenvolve, e nunca se terá uma caraterização completa da solução implementada.

É importante realçar que existem vários tipos de testes aos quais se pode sujeitar uma solução. Este guia irá focar-se no teste individual dos seus componentes (unidades) e no teste das funcionalidades necessárias da aplicação. Aplicações mais complexas irão necessitar de muitos outros testes.

14.3 Testes Unitários

Os testes unitários são testes aplicados pelos programadores às unidades que compõem os programas que desenvolvem. Uma unidade é um pequeno trecho de código que pode ser testado de forma independente, tal como: parte de uma função, uma função ou uma pequena classe. Podem e devem ser aplicados diversos testes a uma unidade, de forma a cobrir todos os casos de interesse, normais e excecionais. Se um dado caso não for coberto por um dos testes, esse caso é considerado indeterminado, não se podendo presumir que a solução cobre a situação de forma correta.

Foram criadas diversas ferramentas, adequadas a cada linguagem, que permitem criar testes e automatizar a sua validação. Em *Python* pode-se encontrar o módulo `unittest` ou a ferramenta `py.test`, que iremos usar de seguida. Em *Java* a classe `JUnit` apresenta funcionalidades semelhantes. Muitas outras plataformas permitem realizar o mesmo. Encontra uma lista no endereço `http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`.

O programa `py.test` permite a execução e validação de testes em programas Python, de

[1] `http://jenkins-ci.org/`

forma simplificada.

Exercício 14.1

Verifique se o comando **py.test** está disponível no seu computador, executando:

```
$ py.test
```

Se aparecer uma mensagem parecida com esta:

```
============================= test session starts ==============================
platform linux2 -- Python 2.7.6 -- pytest-2.5.1
collected 0 items

==============================  in 1.82 seconds ===============================
```

então já está instalado.

Se aparecer uma mensagem de erro, terá de instalar o pacote **py.test**. Para isso é necessário realizar **uma** das seguintes operações.

Se tiver permissões para a instalação de pacotes, num sistema *Ubuntu* ou *Debian*, pode executar:

```
$ sudo apt-get install python-pytest
```

Se não tiver permissões, pode instalar a aplicação na área pessoal usando um instalador de pacotes python:

```
$ pip install --user pytest        # instala o pacote
$ export PATH=~/.local/bin:$PATH   # indica à shell onde encontrar o comando
```

Isto irá instalar a aplicação em **~/.local/bin/py.test**.

Vejamos um exemplo de utilização da ferramenta **py.test** para o desenvolvimento de uma função segundo a metodologia TDD.

Considere que pretendemos criar uma função chamada **fibonacci**, que determine os **n** primeiros valores da sequência de Fibonacci. Antes de iniciar a implementação devem especificar-se os testes, e estes devem cobrir tanto os casos considerados normais, como os

casos especiais. Decidimos que a função irá aceitar um argumento e irá devolver depois uma lista com a sequência.

Começamos por enuniciar os seguintes testes:

- Para valores de **n** inferiores a 1 a função deverá devolver uma lista vazia.
- Para **n** igual a 1 a função deverá devolver **[1]**.
- Para **n** igual a 2 a função deverá devolver **[1, 1]**.
- Para **n** igual a 5 a função deverá devolver **[1, 1, 2, 3, 5]**.
- Para qualquer **n** a função deverá devolver uma lista com **n** elementos.

Após a definição destes testes podemos iniciar o desenvolvimento, tratando cada um dos testes de forma isolada. Isto envolve implementar o código, validar o teste e re-arranjar o código de forma a continuar consistente. Neste caso, o primeiro teste requer que a função tenha a seguinte estrutura:

```
def fibonacci(n):
    res = []
    if n < 1:
        return res
```

Como é óbvio, esta implementação não pode ser considerada correta pois caso **n** seja superior ou igual a 1 a função não irá devolver nada. No entanto, do ponto de vista do primeiro teste ele será concretizado com sucesso. A implementação pode então continuar, procurando satisfazer os testes seguintes, um de cada vez, até que todos sejam bem sucedidos.

A execução dos testes pode ser feita de forma bastante simples, recorrendo a sequências de instruções que validam o valor devolvido pelas funções. O primeiro teste poderia ser implementado fazendo:

```
def teste1():
    if fibonacci(0) == [] and fibonacci(-1) == []:
        print "Teste OK"
    else:
        print "Teste Falhou"
```

Esta função irá imprimir **Teste OK** ou **Teste Falhou** dependendo do valor devolvido pela função. No entanto, isto não é suficiente para, de uma forma sistemática, executar

testes e avaliar a situação do desenvolvimento da aplicação. Por isso recomenda-se a utilização da ferramenta **pytest**.

Para usar esta ferramenta, os testes têm de ser codificados em funções com nomes começados por **test_**. Normalmente essas funções são definidas em ficheiros auxiliares com nomes começados por **test_** e com extensão **.py**. Por exemplo, o ficheiro **test_fib.py** poderia começar por definir uma função para o primeiro teste que enunciámos:

```
# test_fib.py
import pytest
from fib import fibonacci

def test_inferior_1():
    print "Testa comportamento com n < 1"
    assert fibonacci(0) == []
    assert fibonacci(-1) == []
```

De notar que é necessário importar a função **fibonacci**, que deverá estar num ficheiro chamado **fib.py**. A palavra reservada **assert** tem o significado de *afirmar* e usa-se para verificar *asserções* ou seja, condições que se presume serem sempre verdadeiras. Num programa normal, se a condição avaliada num **assert** for falsa, é gerada uma exceção que interrompe o programa com uma mensagem de erro informativa. Pelo contrário, quando executada pelo **py.test**, uma asserção falsa é reportada como uma falha do teste, mas a verificação avança para os testes seguintes.

Exercício 14.2

Implemente os testes que definimos acima no ficheiro **test_fib.py** e verifique que executam através da ferramenta **py.test**. **Depois de todos os testes estarem implementados**, implemente progressivamente o código num ficheiro **fib.py** de forma a cumprir cada um dos testes.

Exercício 14.3

Defina testes unitários para seis funções que realizem as operações aritméticas de soma, subtração, multiplicação, divisão, resto da divisão inteira e raíz quadrada sobre valores reais.

Implemente as funções de forma a cumprirem os testes desenvolvidos. Tenha em consideração os valores aceitáveis (o domínio) para cada uma das operações.

14.4 Testes Funcionais

Os testes unitários dedicam-se à verificação de componentes do software tão pequenos quanto possível. Mas uma aplicação é formada por muitos componentes que interagem de forma complexa e a correção dos componentes isolados não basta para garantir a correção da aplicação. Para verificar a funcionalidade de uma aplicação é necessário considerar aspetos de execução e de interface com o utilizador, tomando a aplicação como um todo. Esse é objetivo dos testes funcionais.

Os testes funcionais podem ser encarados da mesma forma que os unitários, mas focam-se sobre aspetos mais amplos da aplicação. Eles devem ser desenvolvidos de forma única para cada aplicação, visto que cada aplicação possui funcionalidades distintas. No caso dos exemplos tratados anteriormente, faz sentido avaliar os resultados debitados pelo programa (o seu *output*), quando se lhe fornece certos dados (o seu *input*).

A ferramenta **`py.test`** também pode ser utilizada para testes funcionais, pois é possível executar uma aplicação, capturar o que escreve para o ecrã e comparar o resultado obtido com o esperado. A chave para isto é o módulo **`subprocess`** e a classe **`Popen`**. Usando estas classes é possível executar uma aplicação da seguinte forma:

```
from subprocess import Popen
from subprocess import PIPE

proc = Popen("comando a executar", stdout=PIPE, shell=True)

return_code = proc.wait()
output = proc.stdout.read()
```

A linha **`proc.stdout.read()`** obtém tudo o que a aplicação escreve para o ecrã. Para se iterar sobre cada linha seria possível fazer:

```
...
for line in iter(proc.stdout.readline,''):
    sys.stdout.write(line)
```

Exercício 14.4

Implemente um programa que execute o comando **`"ls -la "+sys.argv[1]`** e imprima o seu resultado, mas descartando qualquer linha que contenha um termo fornecido no segundo argumento ao programa. Pode verificar se uma linha contém um termo usando **`if termo in line:`**.

O exemplo anterior pode ser combinado de forma a ser parte integrante de um teste. Pode-se comparar a impressão da execução de um comando e o seu código de retorno, bastando para isso que o código pertença a uma função com nome iniciado por `test_`.

Exercício 14.5

Defina um conjunto de testes funcionais para as aplicações consideradas anteriormente (Fibonacci e Calculadora). Considere a introdução de argumentos inválidos (*Strings*), a total ausência de argumentos, ou a introdução de valores inválidos para operações específicas. Em cada caso a aplicação deverá devolver mensagens de erro específicas.

Implemente os testes usando o formato necessário pela ferramenta `py.test`.

Não implemente qualquer funcionalidade na aplicação que vá de acordo aos testes.

Exercício 14.6

Ordene os testes e, um de cada vez, implemente as funcionalidades necessárias para a aplicação o passar com sucesso.

14.5 Depuração

Um programa pode apresentar uma miríade de erros que impedem o seu funcionamento correto. No contexto da linguagem *Python* podemos encontrar:

Erros de sintaxe: Encontrados pelo interpretador de *Python* quando converte o código fonte para instruções. Estes erros são detetados imediatamente após a tentativa de execução de uma aplicação, resultando numa mensagem de `SyntaxError: invalid syntax`. Parêntesis ou outro caráter em falta, indentação incorreta, ou erros na escrita das palavras reservadas levam a que seja produzido este erro.

Erros de execução: Encontrados pelo interpretador quando uma situação excepcional é encontrada durante a execução. São situações como uma divisão por zero, ou uma operação entre tipos incompatíveis, que ocorrem devido ao fluxo de execução particular, não sendo possível prever esta situação quando o ficheiro é carregado.

Erros semânticos: O programa executa sem detetar qualquer erro, mas o resultado não é o esperado. Os testes funcionais e unitários podem detetar estes erros.

O primeiro tipo de erros é detetado facilmente pelo interpretador, que os localiza e identifica. Os erros de execução também são detetados pelo interpretador, mas não é detetada a sua causa. O terceiro tipo de erros apenas pode ser detetado com testes, mas mais uma vez, apenas se saberá que o erro existe dentro de uma unidade ou de uma funcionalidade, não se sabendo em concreto qual o problema.

Para estes casos existem mecanismos nas linguagens e ferramentas que permitem executar de forma interativa uma aplicação, sendo possível inspeccionar cada ponto da execução. Estas ferramentas, denominadas por depuradores (*debuggers*), executam os programas fornecendo ao programador muito mais controlo sobre a sua execução. São exemplos o módulo **pdb**, o **pydbgr** ou o **ipdb**, todos eles disponíveis para instalação com **pip** ou **easy_install**. Alguns Integrated Development Environment (IDE) já possuem esta funcionalidade incluída, como é o caso do Eclipse[2], do Microsoft Visual Studio[3] ou do PyCharm[4], entre outros.

Considere o seguinte programa que, de uma forma simples (e incorreta) tenta verificar se um dado número é primo:

```
import sys

def main(argv):
    n = int(argv[1])
    for x in xrange(n/2):
        if n % x != 0:
            print "False"
    print "True"

main(sys.argv)
```

Este programa não apresenta nenhum erro sintático. No entanto apresenta um erro de execução e um semântico. Neste caso é simples de detetar por revisão do código, mas poderia não ser, servindo mesmo assim como exemplo para utilização de um depurador.

Pode-se iniciar o depurador para um dado programa executando:

```
$ python -m pdb prime.py 10
```

De notar que os argumentos, neste caso 10, são passados para o programa tal como se não estivesse a ser depurado. A partir deste momento o programa é carregado e está pronto a ser depurado.

[2] https://eclipse.org/
[3] http://www.visualstudio.com/
[4] https://www.jetbrains.com/pycharm/

```
> prime.py(1)<module>()
-> import sys
(Pdb)
```

O depurador indica a instrução que está pronta para ser executada, apresenta um prompt e aguarda por comandos.

Alguns comandos bastante úteis que se podem utilizar são os seguintes:

`continue`: Continua a execução;

`run`: Volta a executar o programa;

`break <nome-da-funcao>`: Define que a execução deve ser parada na função indicada;

`break <numero-da-linha>`: Define que a execução deve ser parada na linha de código indicada;

`print <nome-da-variavel>`: Mostra o valor de uma variável;

`list`: Lista o código fonte;

`next`: Executa a próxima instrução;

`step`: Executa a próxima instrução. Se a instrução chamar uma função, a execução interativa entra dentro da função, de forma a que a próxima instrução a executar interactivamente seja a primeira instrução da função.

Neste caso interessa definir um *breakpoint* na linha 4 e depois executar cada instrução passo a passo até encontrar um erro. O resultado seria:

```
$ python -m pdb prime.py  10
> prime.py(1)<module>()
-> import sys
(Pdb) break 4
Breakpoint 1 at prime.py:4
(Pdb) continue
> prime.py(4)main()
-> n = int(argv[1])
(Pdb) n
> prime.py(5)main()
-> for x in xrange(n/2):
(Pdb) n
> prime.py(6)main()
-> if n % x == 0:
(Pdb) n
```

```
ZeroDivisionError: 'integer division or modulo by zero'
> prime.py(6)main()
-> if n % x == 0:
(Pdb) print n
10
(Pdb) print x
0
```

Após a ocorrência do erro, pode-se verificar em que linha aconteceu e inspeccionar o valor das variáveis atuais. Neste caso, verifica-se que a variável **`x`** é igual a 0, o que gera uma divisão por 0. Sem o depurador seria possível obter informação do erro, mas não poderíamos inspecionar a memória no momento do erro. Um aspeto interessante do depurador é que ele permite inspecionar a memória de uma aplicação quando são encontrados problemas, mesmo que não tenhamos definido um *breakpoint* previamente.

Exercício 14.7

Volte a executar o programa no depurador e não defina nenhum *breakpoint* nem execute o programa de forma interativa. Simplesmente forneça a instrução de **`continue`**. Repare que o programa é interrompido na ocorrência de um erro.

Verifique que pode inspeccionar o estado de todas as variáveis e pode mesmo alterar o seu conteúdo através de sintaxe *Python* (p. ex. **`x = 1`**).

Exercício 14.8

Corrija o erro em causa e voltando a utilizar o depurador, verifique a correta execução do programa. Caso encontre um problema, corrija-o e volte a iniciar uma sessão de depuração.

Exercício 14.9

Crie e implemente testes funcionais de forma a validar o programa acima referido. Ele deverá aceitar um argumento inteiro superior a 0 na linha de comandos e determinar se é primo ou não. O resultado é apresentado através da impressão das palavras **`False`** ou **`True`** e definição do código de execução: 1 para primo, 0 para não primo.

14.6 Para Aprofundar

Exercício 14.10

Construa testes unitários para os programas do segundo guião de Programação 2. Utilize para isso a linguagem *Java*. Em *Java* existe a classe *JUnit* que pode facilitar a sua especificação e que possui funcionalidades semelhantes à ferramenta `py.test`. Pode encontrar exemplos de como utilizar a class *Junit* no endereço: `https://github.com/junit-team/junit/wiki/Getting-started`

Exercício 14.11

Construa testes funcionais para os programas do segundo guião de Programação 2 e valide a sua execução. Utilize para isso a linguagem *Java* ou *Python*.

Glossário

IDE Integrated Development Environment

TDD Test Driven Development