Laboratórios de Informática 2014/2015

DETI - Universidade de Aveiro

Docentes
João Paulo Barraca <jpbarraca@ua.pt>
Diogo Gomes <dgomes@ua.pt>
João Manuel Rodrigues <jmr@ua.pt>
Mário Antunes <mario.antunes@ua.pt>

TEMA 14

Testes e Depuração

Objetivos:

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

14.1 Introdução

A validação de uma dada aplicação 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 no início do desenvolvimento, é possível construir aplicações que robustas e previsíveis. A existência de testes permite igualmente verificar que existem regressões no desenvolvimento, ou qual o estado de desenvolvimento de uma dada aplicação. Depois de identificada a existência de problemas, torna-se necessário encontrar a razão da anomalia, o que normalmente se faz através de depuração interactiva 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 Testes

A metodologia Test Driven Development (TDD) é muito utilizada ao longo dos últimos anos. Assume uma aproximação alternativa perante o desenvolvimento de aplicações, com foco na definição, execução e correção de testes. Considera-se que não é possível determinar qual o estado de uma aplicação se ela não for testada em todos os seus componentes. Sem estes testes é possível a existência de problemas não descobertos, ou casos excepcionais que irão provocar problemas no futuro.

Segundo TDD, o início de qualquer desenvolvimento é iniciado através da definição de testes, mesmo quando ainda não existe qualquer código presente. Todos os testes irão falhar, pois as funcionalidades não estão implementadas. A este estado dá-se o nome de *RED* (vermelho), cada teste deverá ser satisfeito de forma independente, evoluindo o estado da funcionalidade para o estado de *VERDE* (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 *REFACTOR*. Este processo está descrito na Figura 14.1. Quando todos os testes forem bem sucedidos, considera-se que a aplicação possui toda a programação prevista, para todos os casos de utilização previstos.

A metodologia é completamente agnóstica da linguagem de programação, e não necessita de qualquer ferramenta para a sua implementação. 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 arquitectura 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 soluções que desenvolvem. Neste contexto, uma solução é partida em várias unidades. Cada unidade deve compreen-

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

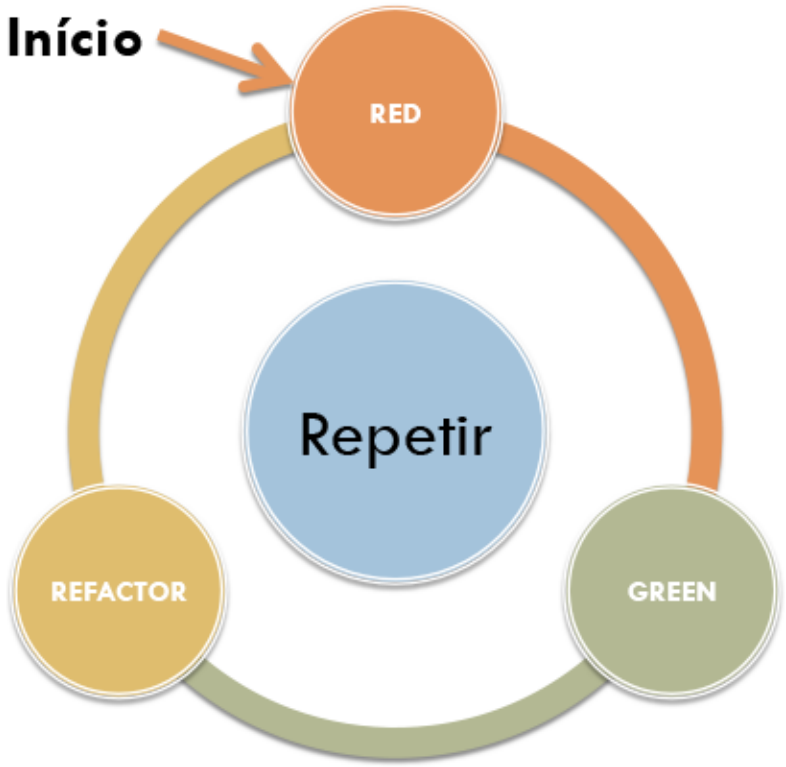

Figura 14.1: Fluxo de processos na metodologia TDD.

der um conjunto de código pequeno, tal como parte de uma função, uma pequena função, ou uma pequena classe com um método. Podem e devem ser aplicados diversos testes a uma unidade, de forma a cobrir todos os casos. Se um dado caso não for coberto por um dos testes, a execução desse caso é considerado indeterminado, não se podendo assumir 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, sendo que uma lista é mantida no endereço `http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`.

De forma a instalar o pacote **`py.test`** é necessário realizar **UMA** das seguintes operações.

Se se pretender instalar a aplicação na área pessoal, sem que sejam necessárias

permissões (ex, nos computadores das salas):

```
$ easy_install --user pytest
$ export PATH=~/.local/bin:$PATH
```

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

Caso se possuam permissões para a instalação de pacotes, num sistema *Ubuntu* ou *Debian*, pode ser executado:

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

ou:

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

Em ambos os casos o comando **`py.test`** deverá estar disponível para execução e validação de testes, o que pode ser verificado, executando:

```
$ py.test
```

Considerando uma solução que apresente os **`n`** primeiros valores da sequência de Fibonacci contida numa função chamada **`fibonacci`**, antes de iniciar o desenvolvimento devem-se determinar os testes, cobrindo estes tanto os valores normalmente considerados normais, como os casos especiais. Assume-se que a função irá aceitar um argumento e irá devolver depois uma lista com a sequência em que o tamanho da lista deverá ser igual ao valor especificado no argumento.

No caso da sequência de Fibonacci, consideram-se 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 o tamanho indicado em **`n`**.

Após a definição destes testes é possível inicial 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 se pode notar, 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. Os testes seguintes seriam depois implementados, um de cada vez, de forma a que todos validassem corretamente.

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

```
def teste1():
    if fibonacci(0) == []:
        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 e por isso se recomenda a utilização da ferramenta **pytest**.

Usando esta ferramenta, os testes são criado através de funções em que o nome inicia por **test_**, normalmente localizadas em ficheiros auxiliares, com nome iniciado em **test_** e com extensão **.py**. O conteúdo do ficheiro com o código (**test_fib.py** que valida o primeiro teste desenhado anteriormente seria:

```
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 possui a capacidade de interromper gerar uma situação excepcional caso a afirmação seja falta.

Exercício 14.1

Implemente os testes anteriores no ficheiro `test_fib.py` e verifique que executam através da ferramente `py.test`. **Depois de todos os testes estarem implementados**, implemente o código que cumpra cada um dos teste.

Exercício 14.2

Defina os testes unitários para seis funções realizando 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 para cada uma das operações.

14.4 Testes Funcionais

Os testes unitários realizados anteriormente garantem que o funcionamento de cada unidade é correto, sendo que por unidade entendeu-se considerar uma função. Nada garante que a operação de uma aplicação usando as funções em causa seja a correta. Consideram-se neste caso aspetos de execução e de interface com o utilizador, sendo que a operação da aplicação requer que diversos processos sejam executados de forma correta. Ou seja, embora o funcionamento interno seja garantido, não existem garantias que uma aplicação desenvolvida seja considerada funcional. Neste sentido é necessária a realização de 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 a entrada e saída do programa, através de análise do que escreve para o ecrã (`stdout`).

A ferramenta `py.test` pode ser utilizada da mesma forma, pois é possível executar aplicações, capturar o que escrevem para o ecrã e comparar este texto com outro de referência. 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()
```

De notar que 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 14.3

Implemente um programa que execute o comando **`"ls -la "+sys.argv[1]`** e imprima o seu resultado mas ignore (não imprima) qualquer linha que contenha um termo fornecido no segundo argumento ao programa. Pode verificar se uma linha contem 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 14.4

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 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 14.5

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 de *Python* podemos encontrar:

- Erros de Sintaxe: Encontrados pelo interpretador de *Python* a quando da conversão do 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. Por exemplo, pode-se considerar uma função que deveria devolver um inteiro, mas por um erro na implementação, afinal devolve `None` e este valor é depois somado a um inteiro. O erro acontece devido ao fluxo de execução particular, não sendo possível prever esta situação a quando da interpretação.

· Erros Semânticos: O programa executa sem que seja apresentado 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, indicando ele onde existe e qual é o problema. O tipo seguinte (Erros de Execução) é detetado mas não é detetada a razão da sua existência. O terceiro tipo de erros apenas pode ser detetada 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 interactiva 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], Microsoft Visual Studio[3], ou o 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:

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

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

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

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

Alguns comandos bastante úteis que se podem utilizar no depurador de *Python*:

- **continue** - continua a execução;
- **run** - volta a executar o programa;
- **break <nome-da-funcao>** ou **break <numero-da-linha>** - define que a execução deve ser parada na função ou linha de código em causa;
- **print <nome-da-variavel>** - permite inspeccionar o valor de qualquer variável;
- **list** - lista o código fonte;
- **next** - executa a próxima instrução;
- **step** - executa a próxima instrução. Se a instrução chamar uma função a execução interactiva 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 seria impossível inspeccionar a memória no momento do erro. Um aspeto interessante do depurador é que ele permite inspeccionar a memória de uma aplicação quando são encontrados problemas, mesmo que não tenhamos definido um *breakpoint*.

Exercício 14.6

Volte a executar o programa no depurador e não defina nenhum *breakpoint* nem execute o programa de forma interactiva. 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* (ex, `x = 1`).

Exercício 14.7

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 14.8

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 14.9

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 14.10

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