



**ALL CODE IS GUILTY  
UNTIL PROVEN INNOCENT**

<http://www.eternalis-software.com/images/tdd.jpg>

# TESTES E DEPURAÇÃO

# Testes: Para quê?

## O código funciona?

- “Funciona porque sim!”
- “Não funciona.”
- “Funciona às vezes.”
- “Deve funcionar quase tudo.”
- Estas respostas são inaceitáveis!

# Testes

- Não se pode gerir o que não se conhece.
- Responder à questão implica conhecer o comportamento do programa.
  - ▣ O que funciona?
  - ▣ Como funciona?
  - ▣ Que problemas apresenta?
  - ▣ ...

# Testes: Como?

- ❑ Comparar resultados obtidos com resultados esperados.
- ❑ Testar correção: casos dentro do domínio, tanto normais, como extremos!
- ❑ Testar robustez: casos fora do domínio.

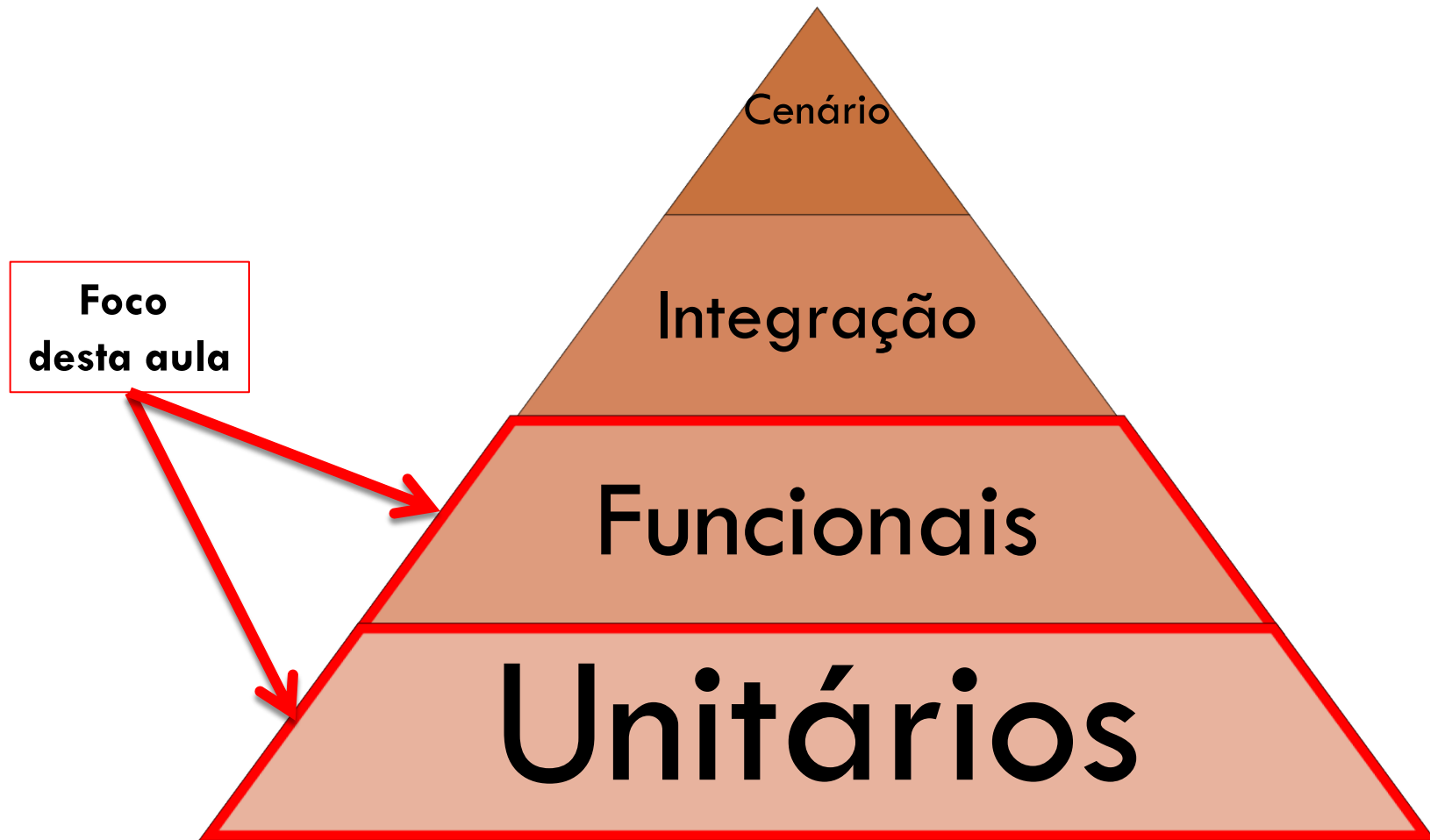
# Testes: Interpretação

- Programa correto  $\Rightarrow$  todos os testes satisfeitos!
- Isso significa:
  - ▣ Um teste falha  $\Rightarrow$  programa incorreto!
  - ▣ Todos os testes passam **não implica** um programa correto! (Só se fossem exaustivos.)

# Hierarquia de Testes



# Hierarquia de Testes



# Testes Unitários

- Focam-se no teste de uma unidade
- Uma unidade realiza 1 função
  - ▣ Peçaço de código
  - ▣ Função ou método
  - ▣ Classe pequena
- Testes feitos pelo próprio programador



# Testes Funcionais

- Focam-se no teste de uma funcionalidade
- Uma funcionalidade envolve vários algoritmos
  - ▣ Pode envolver várias classes ou funções
  - ▣ Resultado externo de um módulo ou programa
- Testes feitos pelo programador ou equipa

# Testes

## □ Integração

- ▣ Funcionamento coerente entre módulos diferentes
- ▣ Funcionalidades compostas
- ▣ Ex: Testar integração mapa em página

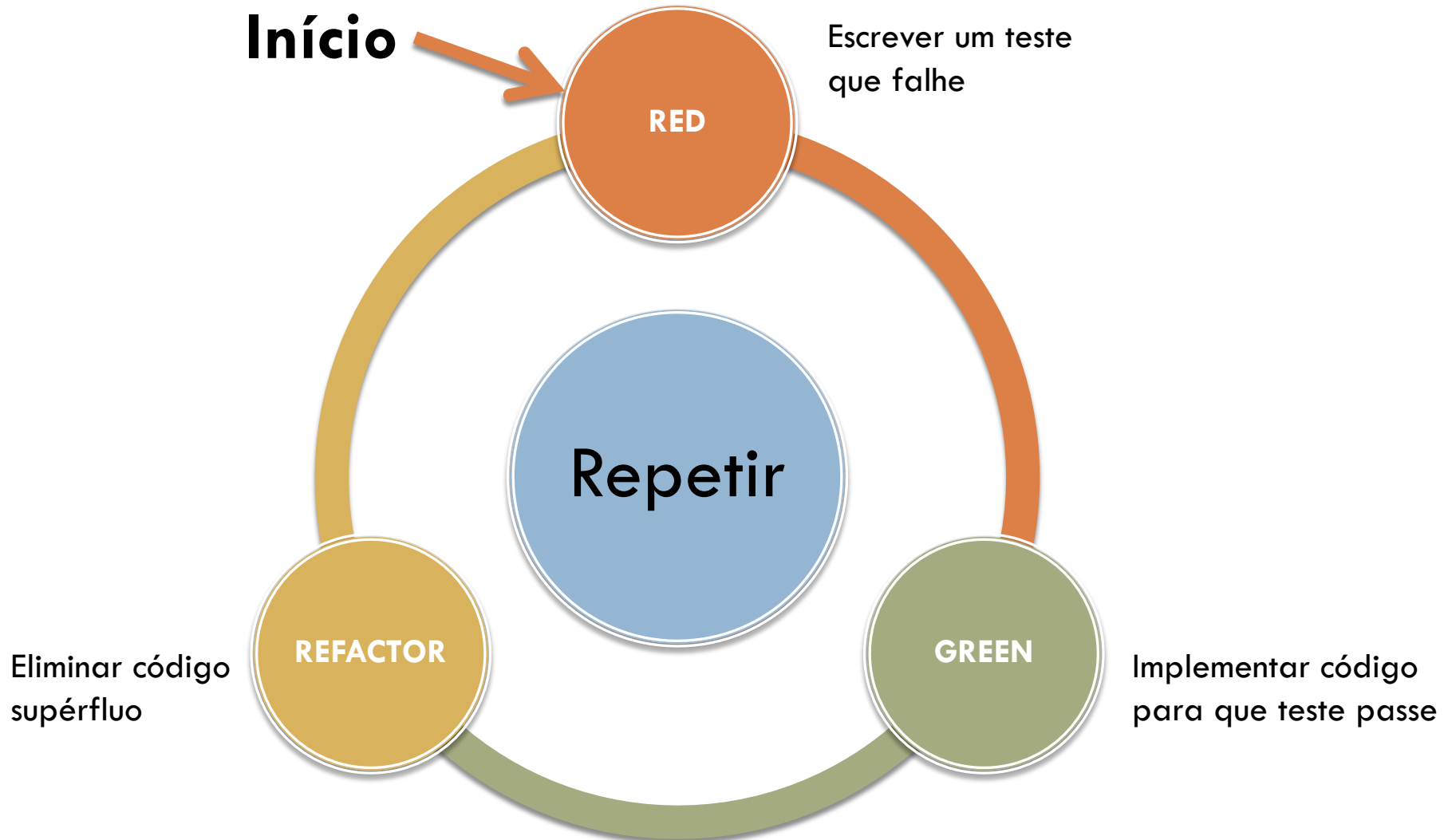
## □ Cenário

- ▣ Concretizar corretamente um cenário inteiro
- ▣ P. ex. Página permite localizar rede de lojas

# Test Driven Development

- Metodologia muito comum focada em testes
- Desenvolvimento **inicia-se com especificação dos testes**
  - ▣ Não com desenvolvimento do código!
- Testes podem representar 60-80% do custo de desenvolvimento!
- Sabe-se sempre o que funciona, ou não.

# Ciclo TDD



# Exemplo: Factorial

- ❑ Cliente: Quer calcular factorial na consola
- ❑ Objetivo: Programa que dado um número, calcule o fatorial e o imprima para o ecrã seguido de uma nova linha.

```
$ python3 factorial.py 10  
3628800  
$
```

# Metodologia TDD

## □ Definir testes unitários que falhem

- ▣ `factorial(-1) == "undefined"`
- ▣ `factorial(0) == 1`
- ▣ `factorial(5) == 120`
- ▣ `factorial(10) == 3628800`

## □ Definir testes funcionais

- ▣ `python3 factorial.py` deve mostrar ajuda
- ▣ `python3 factorial.py foobar` deve mostrar ajuda
- ▣ `python3 factorial.py 5` deve imprimir 120

# RED: factorial(-1) == undefined

- py.test auxilia realização de testes (Python)
  - ▣ Poderiam ser feitos testes com funções próprias

**factorial.py**

```
def factorial(x):  
    return x
```

**test\_factorial.py**

```
import pytest  
from factorial import factorial #importar unidade (função)  
  
def test_negativos():  
    assert factorial(-1) == "undefined"
```

# RED: factorial(-1) == undefined

```
$ py.test
===== test session starts =====
platform darwin -- Python 3.6.4, pytest-3.4.1, py-1.5.2, pluggy-0.6.0
collected 1 items

test_factorial.py F

===== FAILURES =====
_____ test_negativos _____

    def test_negativos():
>         assert factorial(-1) == "undefined"
E         assert -1 == 'undefined'
E         + where -1 = factorial(-1)

test_factorial.py:5: AssertionError
===== 1 failed in 0.03 seconds =====
```



# Todos os testes Unitários

```
import pytest
from factorial import factorial

def test_negativos():
    assert factorial(-1) == "undefined"

def test_zero():
    assert factorial(0) == 1

def test_valor_pequeno():
    assert factorial(5) == 120

def test_valor_grande():
    assert factorial(10) == 3628800
```

```
===== test session starts =====
platform darwin -- Python 3.6.4, pytest-3.4.1, py-1.5.2, pluggy-0.6.0
collected 4 items

test_factorial.py ....

===== 4 passed in 0.02 seconds =====
```

# Implementação

```
def factorial(x):
```

```
    if x < 0:
        return "undefined"
```

**Test\_negativo**

```
    if x == 0:
        return 1
```

**Test\_zero**

```
    res = 1
```

```
    while x > 0:
        res = res * x
        x = x - 1
```

**Test\_valor\_pequeno**  
**Test\_valor\_grande**

```
    return res
```

# Testes funcionais

- Neste caso: verifica-se a execução
  - ▣ comparação da saída (stdout)
  - ▣ comparação do código de execução

## **test\_factorial\_funcional.py**

```
import pytest
from subprocess import Popen
from subprocess import PIPE

def test_no_args():
    proc = Popen("python factorial.py", stdout=PIPE, shell=True)
    assert proc.wait() == 1 #Check Return Code
    assert proc.stdout.read().decode('utf-8') == "Usage: python factorial.py number\n"
```

# Implementação

```
def factorial(x):  
    ...  
  
def usage(progname):  
    print("Usage: python %s number" % (progname))  
  
def main(argv):  
    if(len(argv) != 2):  
        usage(argv[0])  
        sys.exit(1);  
  
    if not argv[1].isdigit():  
        usage(argv[0])  
        sys.exit(2)  
  
    print(factorial(int(argv[1])))  
    sys.exit(0)  
  
if __name__ == "__main__":  
    main(sys.argv)
```

# Depuração

- Foi detetado um comportamento incorreto
- Como se deteta o erro específico?
  - ▣ Revisão do código
  - ▣ Execução interativa com depuração
- Depurar: Remover impurezas, sujidade ou imperfeições (em software: bugs)

# Depuração

- Integrado num IDE: Eclipse, PyCharm
- Na execução: gdb, pdb (python)
- Funcionalidades usuais
  - ▣ Interromper programa em qualquer ponto
  - ▣ Inspeccionar memória (variáveis)
  - ▣ Interceptar propagação de erros
  - ▣ Executar passo a passo

# Depuração: Conceitos

- Breakpoint: Pontos de pausa do código
  - ▣ Possível verificar variáveis/memória naquele momento
- Step: Executa uma linha
  - ▣ Step-Over: não entra nas funções dessa linha
  - ▣ Step-Into: entra nas funções dessa linha

# Python: pdb

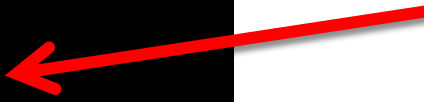
## Iniciar Depuração

```
$ python3 -m pdb factorial.py 10
> /tmp/fact/factorial.py(2)<module>()
-> import sys
>
```

## Criar Breakpoint na função factorial(x) e resumir a execução

```
...
(Pdb) break factorial
Breakpoint 1 at factorial.py:5
(Pdb) continue
> factorial.py(6)factorial()
-> if x < 0:
```

**Execução suspensa  
no início da função**





# Python: pdb

## Ver código atual

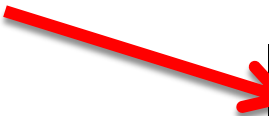
Breakpoint

Instrução atual

```
$ (Pdb) list
1      # encoding=utf-8
2      import sys
3
4      # Implementação do programa
5  B def factorial(x):
6      ->      if x < 0:
7              return "undefined"
8
9          if x == 0:
10             return 1
11
```

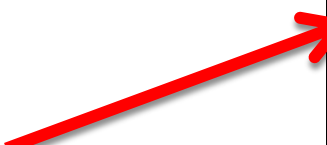
# Python: pdb

**Inspeccionar  
Valor de x**




```
(Pdb) p x
10
```

**Executar uma linha**



```
(Pdb) next
> factorial.py(9)factorial()
-> if x == 0:
(Pdb) next
> factorial.py(12)factorial()
-> res = 1
```

**Modificar o valor de x**



```
(Pdb) x = 3
(Pdb) p x
3
```

# Python: PyCharm

