Testes na Prática

... e na teoria

Osvaldo Santana Neto (https://osantana.me)

Teste automatizado

... é o uso de software para controlar a execução dos testes.

Test-Driven Development

... é um processo de desenho e desenvolvimento de software fundamentado no uso de testes automatizados

Testes automatizados

- Unitários
- Integração
- Funcionais
- Aceitação (BDD)
- Outros: performance, estático, segurança, ...

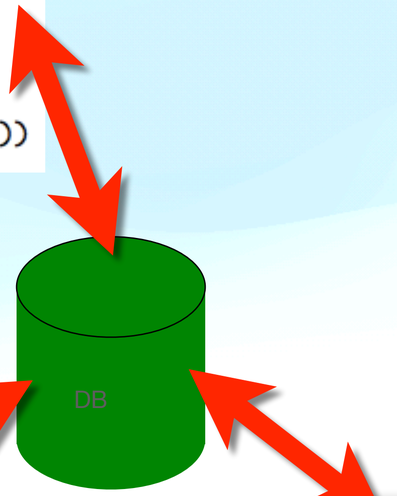
Inventory

```python
class Inventory(object):
    def __init__(self):
        self.inventory = {}

    def add(self, name, qty):
        self.inventory[name] = self.inventory.get(name, 0) + qty

def test_add_item_into_inventory():
    inventory = Inventory()
    inventory.add("item #1", 1)
    assert inventory.get("item #1") == 1
```

Unitário

```python
class TestOrder(unittest.TestCase):
    def setUp(self):
        self.inventory = Inventory()
        self.inventory.add("item #1", 50)

    def test_order_filled(self):
        order = Order("item #1", 50)
        order.fill(self.inventory)
        self.assertTrue(order.filled)
        self.assertEquals(0, self.inventory.get("item #1"))
```

Integração



Order

```python
class Order(object):
    def __init__(self, name, qty):
        self.item = name, qty
        self.filled = False

    def fill(self, inventory):
        if inventory.has_inventory(*self.item):
            inventory.remove(*self.item)
            self.filled = True

def test_fill_order():
    order = Order("item #1", 50)

    mocker = Mocker()
    inventory = mocker.mock()
    inventory.remove("item #1", 50)

    mocker.replay()
    order.fill(inventory)

    mocker.verify()
    assert order.filled
```

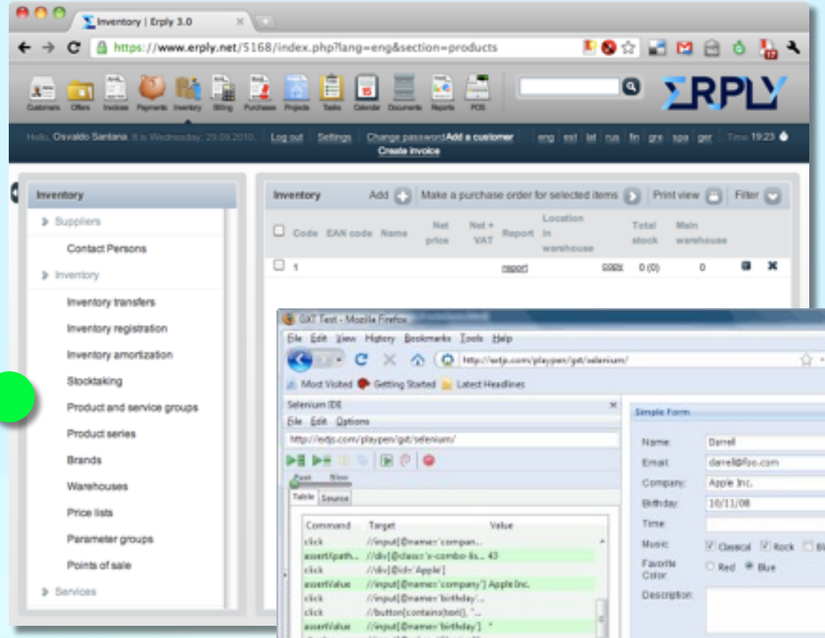
Unitário

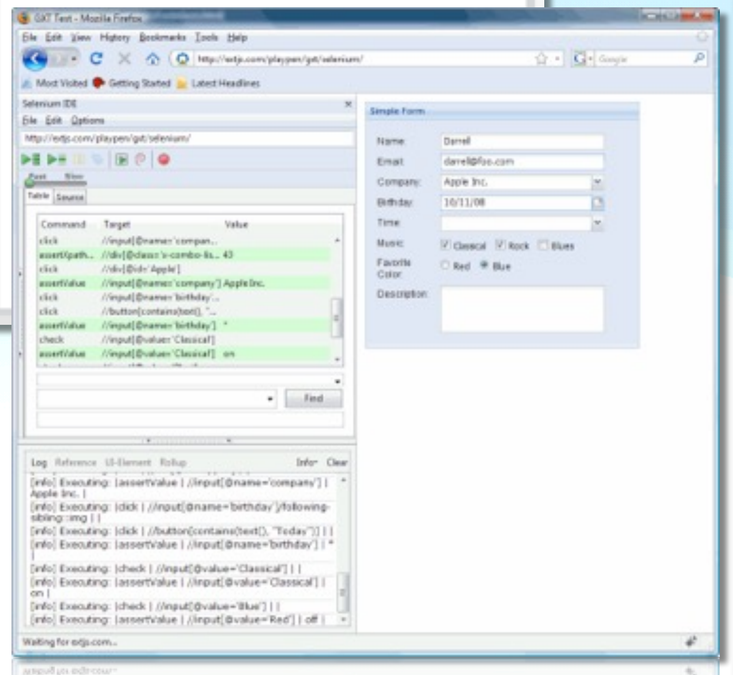
Inventory

```python
class Inventory(object):
    def __init__(self):
        self.inventory = {}

    def add(self, name, qty):
        self.inventory[name] = self.inventory.get(name, 0) + qty

def test_add_item_into_inventory():
    inventory = Inventory()
    inventory.add("item #1", 1)
    assert inventory.get("item #1") == 1
```

Unitário

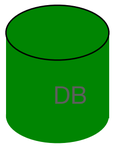
Funcionais






```
class TestOrder(unittest.TestCase):
    def setUp(self):
        self.inventory = Inventory()
        self.inventory.add("item #1", 50)

    def test_order_filled(self):
        order = Order("item #1", 50)
        order.fill(self.inventory)
        self.assertTrue(order.filled)
        self.assertEquals(0, self.inventory.get("item #1"))
```

Integração



Order

```
class Order(object):
    def __init__(self, name, qty):
        self.item = name, qty
        self.filled = False

    def fill(self, inventory):
        if inventory.has_inventory(*self.item):
            inventory.remove(*self.item)
            self.filled = True

def test_fill_order():
    order = Order("item #1", 50)

    mocker = Mocker()
    inventory = mocker.mock()
    inventory.remove("item #1", 50)

    mocker.replay()
    order.fill(inventory)

    mocker.verify()
    assert order.filled
```

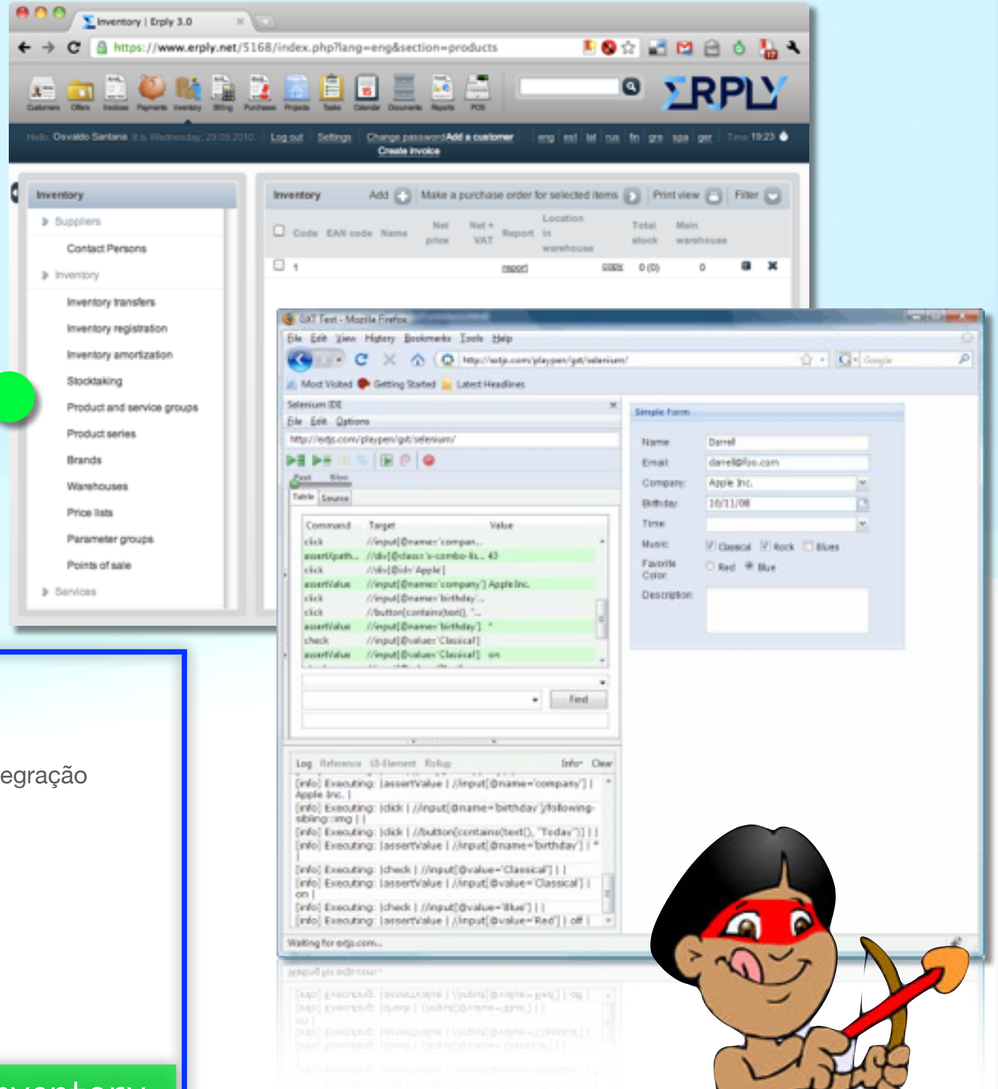
Unitário

Inventory

```
class Inventory(object):
    def __init__(self):
        self.inventory = {}

    def add(self, name, qty):
        self.inventory[name] = self.inventory.get(name, 0) + qty

def test_add_item_into_inventory():
    inventory = Inventory()
    inventory.add("item #1", 1)
    assert inventory.get("item #1") == 1
```

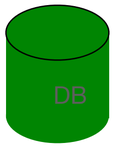
Unitário

Aceitação




```
class TestOrder(unittest.TestCase):
    def setUp(self):
        self.inventory = Inventory()
        self.inventory.add("item #1", 50)

    def test_order_filled(self):
        order = Order("item #1", 50)
        order.fill(self.inventory)
        self.assertTrue(order.filled)
        self.assertEquals(0, self.inventory.get("item #1"))
```

Integração



Order

```
class Order(object):
    def __init__(self, name, qty):
        self.item = name, qty
        self.filled = False

    def fill(self, inventory):
        if inventory.has_inventory(*self.item):
            inventory.remove(*self.item)
            self.filled = True

def test_fill_order():
    order = Order("item #1", 50)

    mocker = Mocker()
    inventory = mocker.mock()
    inventory.remove("item #1", 50)

    mocker.replay()
    order.fill(inventory)

    mocker.verify()
    assert order.filled
```

Unitário

Inventory

```
class Inventory(object):
    def __init__(self):
        self.inventory = {}

    def add(self, name, qty):
        self.inventory[name] = self.inventory.get(name, 0) + qty

def test_add_item_into_inventory():
    inventory = Inventory()
    inventory.add("item #1", 1)
    assert inventory.get("item #1") == 1
```

Unitário

Aceita

```
→ cucumber
Feature: Viewer signs up for the newsletter
  In order to recieve the newsetter
  As a user of the website
  I want to be able to sign up for the newsletter

  Scenario: View form page                                    # features/form_page.feature:
    Given I am on "/form"                                     # features/step_definitions/s
    Then I should see "Fill out this form to receive our newsletter." # features/step_definitions/s

  Scenario: Fill out form
    Given I am on "/form"
    When I fill in "name" with "John Doe"
    And I fill in "email" with "john@doe.com"
    And I click "Sign Up!"
    Then I should see "Hi there, John Doe. You'll new receive our email newsletter at john@doe.com"

Feature: Viewer visits the home page
  In order to read the page
  As a viewer
  I want to see the home page of my app

  Scenario: View home page                    # features/home_page.feature:6
    Given I am on the home page               # features/step_definitions/sinatra.rb:2
    Then I should see "This is the home page." # features/step_definitions/sinatra.rb:23

  Scenario: Find heading on home page                 # features/home_page.feature:10
    Given I am on the home page                       # features/step_definitions/sinatra.rb:2
    Then I should see "MY APP" in the selector "h1" # features/step_definitions/sinatra.rb:28

  Scenario: Find the link to the form                       # features/home_page.feature:14
    Given I am on the home page                             # features/step_definitions/sinatra.r
    Then I should see "Sign up for our newsletter." in a link # features/step_definitions/sinatra.r

5 scenarios (5 passed)
13 steps (13 passed)
0m22.125s
```

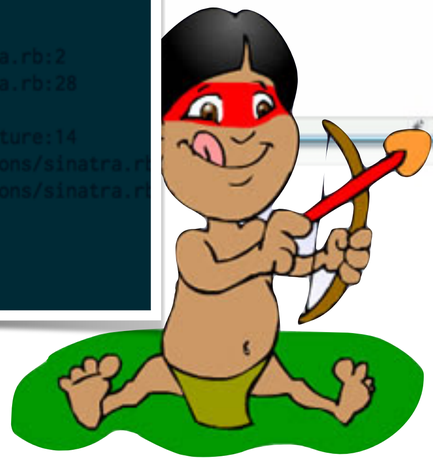
```
class TestOrder(unittest.TestCase
    def setUp(self):
        self.inventory = Inventor
        self.inventory.add("item

    def test_order_filled(self):
        order = Order("item #1",
        order.fill(self.inventory
        self.assertTrue(order.fil
        self.assertEquals(0, self
```

```
class Order(object):
    def __init__(self, name, qty):
        self.item = name, qty
        self.filled = False

    def fill(self, inventory):
        if inventory.has_inventory(*self.item):
            inventory.remove(*self.item)
            self.filled = True

def test_fill_order():
    order = Order("item #1", 50)

    mocker = Mocker()
    inventory = mocker.mock()
    inventory.remove("item #1", 50)

    mocker.replay()
    order.fill(inventory)

    mocker.verify()
    assert order.filled
```

Unitário

```
    self.inventory = {}

    def add(self, name, qty):
        self.inventory[name] = self.inventory.get(name, 0) + qty

def test_add_item_into_inventory():
    inventory = Inventory()
    inventory.add("item #1", 1)
    assert inventory.get("item #1") == 1
```

Unitário



Teste Automatizado

- É uma boa ferramenta de desenvolvimento como qualquer outra ferramenta
- Teste automatizado não entrega valor **direto** para o cliente
  - "*I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence*" - Kent Beck
- Mito: código sem teste é código quebrado.

Testabilidade

- Fácil testar: código bem desenhado, código criado com TDD, funções determinísticas, etc
- Difícil testar: GUI, código assíncrono, esquemas em banco de dados, componentes de aplicações distribuídas, funções não-determinísticas, etc

Cobertura de código

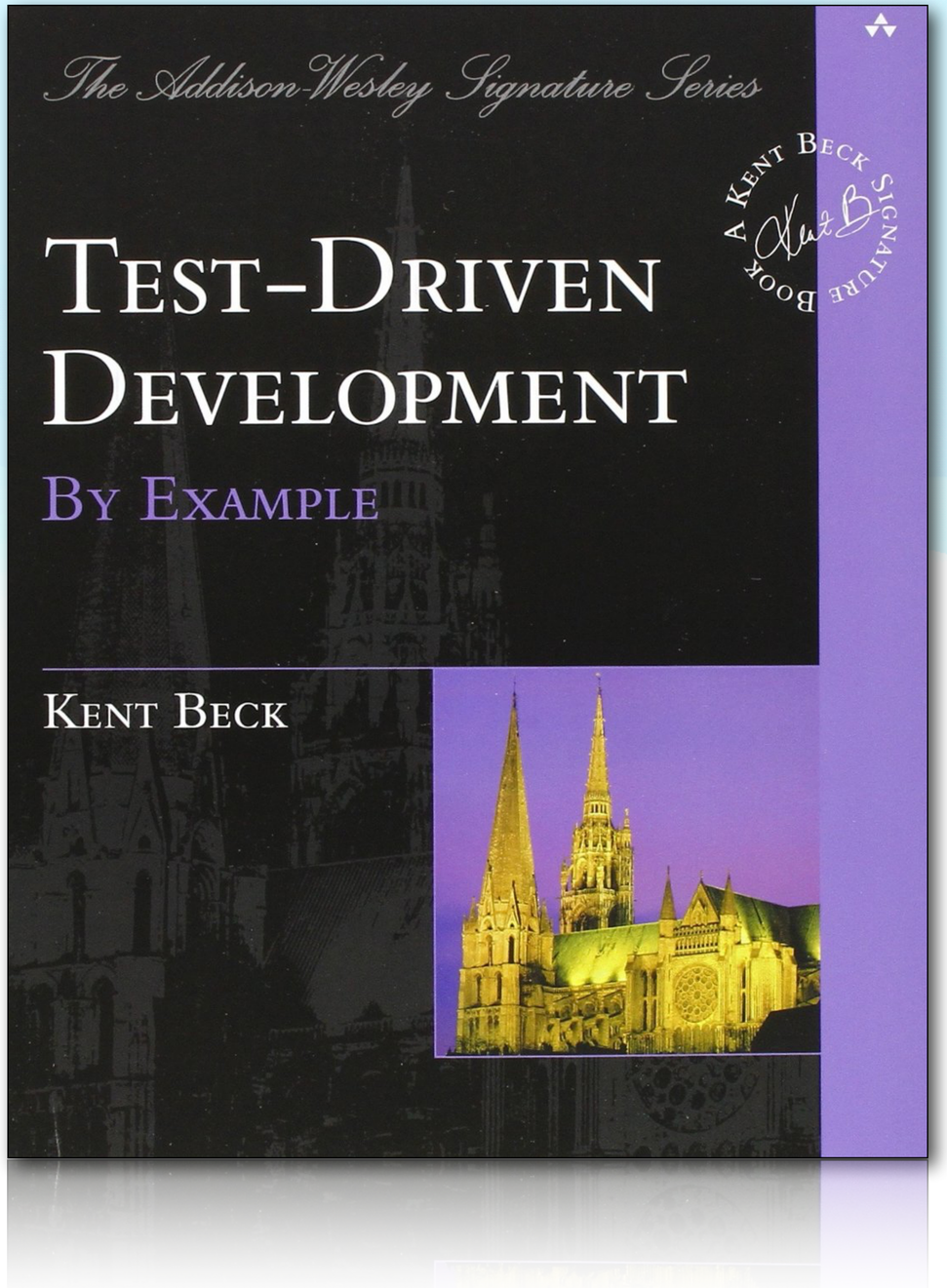
- É **conseqüência** e **não objetivo**!
- Cobertura maior é um **indício** de que os testes estão tratando de todos os cenários
- Não existe ferramenta capaz de medir com 100% de certeza a cobertura
- Código 100% coberto != código sem bugs
- TDD leva a altas taxas de cobertura

Test-Driven Development

Desenvolvimento guiado por testes

Test-Driven Development

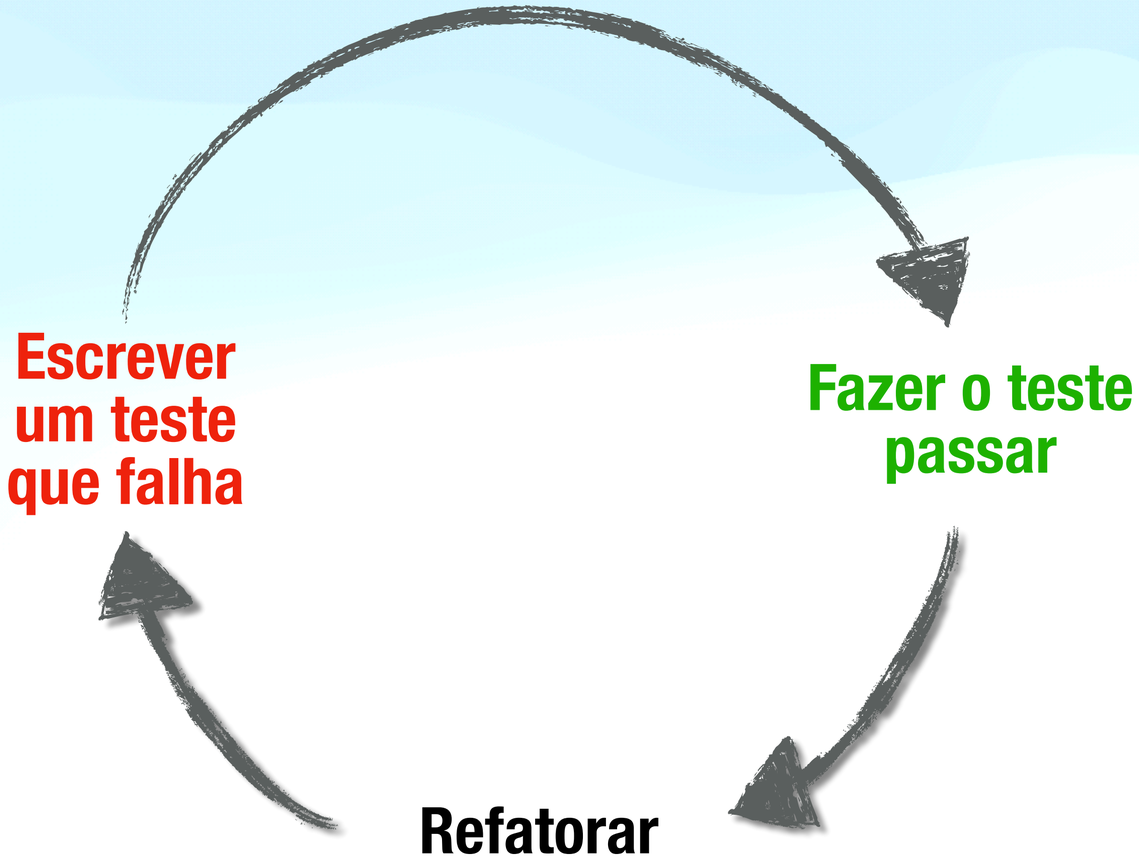
- Abreviação: TDD
- Kent Beck: prática de XP e depois em seu livro Test-Driven Development by Examples
- Utilização de testes automatizados na condução do desenvolvimento



Test-Driven Development

- TDD não é "ensinado".
- TDD é "praticado"
- Na fase de treinamento é importante seguir as regras. Depois podemos quebrá-las.
  - Baby Steps

Red. Green. Refactor.



Red

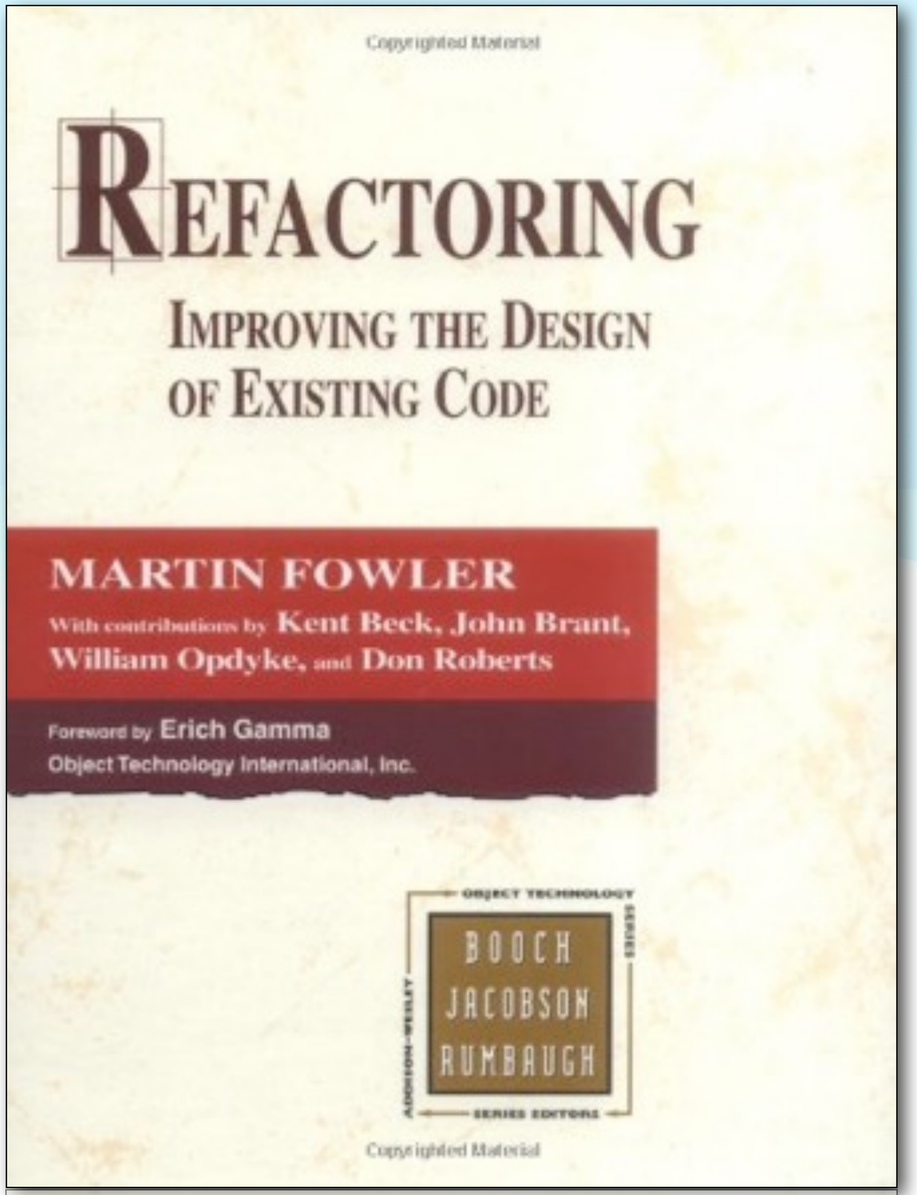
- Escrever um teste que inevitavelmente falhe
- Se o teste não falhar?
  - A nova funcionalidade já existe e, consequentemente, já deve ter sido testada
    - Mantê-lo é opcional mas se não tivermos segurança para removê-lo é melhor mantê-lo
  - Teste com problema

Green

- Escrever o mínimo de código que faça o teste passar… ou não…
  - *Fake It ('Til you make it)* — valores 'hard coded' ou objetos "fakes" no lugar de dependências ainda não implementadas
  - *Triangulate* — implementação real quando você tem dois ou mais testes / cenários
  - *Obvious Implementation ('Til you get red bars)* — implementações óbvias podem ser feitas

Refactor

- Refatoração:
  - Aperfeiçoar o código sem alterar o seu comportamento
- Remover duplicação de código entre o código implementado e o teste



Frameworks & Ferramentas

- Unidades de código: funções, métodos, classes, ...
  - *System Under Test* (SUT) — código que está sendo testado
- Framework xUnit: unittest, pytest
  - Criado por Kent Beck para Smalltalk e posteriormente para Java (jUnit)
- Doctests — documentação “executável" (não use!)

Testes unitários

Teste de uma unidade de código

Passos de um Teste

- Setup (`setUp`) — preapara o ambiente onde o teste será executado (*fixtures*)
- Exercise (`test*`) — executa o procedimento a ser testado
- Assert (`assert*`/`verify`) — verifica os resultados
- Teardown (`tearDown`) — limpa o ambiente

Características

- Características
  - Isolamento — testes são independentes
  - Legibilidade — devem privilegiar legibilidade
  - Velocidade — devem executar rapidamente
  - Manutenabilidade — manutenção deve ser fácil
  - Não intrusivos — código de teste deve ficar somente no teste e não no SUT

Isolamento

- Um teste devem funcionar de forma independente de outros testes e assumir um ambiente "limpo" para execução
- Pode ser necessário substituir as dependências do código testado por "*doubles*" (*fakes*, *stubs* ou *mocks*)
  - Mocks aren't Stubs - Martin Fowler
    http://bit.ly/mockrnstubs
  - Usar Dependency Injection
  - Não faça mocks de objetos não injetados. Ex. mock.patch(“ requests.get”)

Legibilidade

- Legibilidade
  - O código do teste não precisa ser elegante, precisa ser legível. Testes são para "consumo" humano

Pior	Melhor

`banco = Banco()`
`banco.indice("USD", "BRL", TAXA_PADRAO)`
`banco.comissao(COMISSAO_PADRAO)`
 	
`res = banco.converte(`
`  Nota(100, "USD"), "BRL")`
 	
`assert res == Nota(49.25, "BRL")`
	`banco = Banco()`
	`banco.indice("USD", "BRL", TAXA_PADRAO)`
	`banco.comissao(COMISSAO_PADRAO)`
	
	`res = banco.converte(`
	`  Nota(100, "USD"), "BRL")`
	
	`assert Nota(49.25, "BRL") == res`

Legibilidade

- Nome de teste:

```
def test_[fail_][verb]_[subject/objects/details/...](self): ...
    |    |      |      |
    |    |      |      `- basic_calendar, invalid_calendar, ...
    |    |      `-------- login, create, logout, remove, ...
    |    `---------------- fail, restart, start, end, ...
    `-------------------- required*
```

- Apenas um ciclo setup/exercise/verify/teardown por teste

Velocidade

- Testes devem executar instantaneamente
- Se o SUT precisar interagir com serviços lentos:
  - Otimize o sistema em questão para melhorar performance (ex. banco de dados em RAM)
  - Em último caso, use um objeto fake.

Manutenabilidade

- Uma refatoração que muda a **implementação** do SUT mas preserva seu comportamento e API **não** deve quebrar nenhum teste.
  - mock.patch() frequentemente viola esse princípio. Injete o mock no SUT
- Programe as fixtures manualmente e cuide para que não seja necessário várias delas
  - Evite geradores e carregadores de fixtures

Dicas

- O melhor "primeiro teste" é o teste que verifica o caso de uso mais comum. O teste mais básico.
- Comece a escrever pelas "*assertions*"
- O ciclo completo de red/green/refactor deve ser curto para privilegiar o ritmo.
- Se o teste está ficando grande? quebre-o em testes menores

Dicas

- Programando sozinho? Deixe o último teste "quebrado" no fim de uma sessão de programação para saber de onde retomar o desenvolvimento
- Programando em equipe? Faça 'commit' somente se todos os testes estiverem passando
- Usa um sistema de controle de versão distribuído? Deixe 'quebrado' localmente

Testes problemáticos

Test Smells

Testes problemáticos

- Tipos de problemas:
  - *Code Smells* — problemas relacionados com o código dos testes
  - *Behaviour Smells* — problemas relacionados ao comportamento dos testes
- Técnicas e padrões podem ser usados para resolver esse tipo de problema

Problemas com código dos testes

Code Smells

Teste obscuro

Dificuldade em entender o código do teste

Causas	Possíveis Soluções
Teste verifica muitas informações ao mesmo tempo	Reduzir número de verificações
O número de objetos construídos no setup é maior que o necessário	Construir somente as fixtures necessárias para aquele teste
Interação com o SUT não se dá de forma direta e sim através de um intermediário	Remover a indireção e testar o SUT diretamente
Não é possível identificar o que está sendo testado	Simplificar o processo de setup
Excesso de informações irrelevantes no teste	Partir da verificação e remover todos os objetos e informações desnecessárias

Lógica condicional

Código que pode ou não ser executado no teste

Causas	Possíveis Soluções
Teste verifica coisas diferentes dependendo de como executado	Desacoplar o SUT de suas dependências e/ou dividir o teste
Modificar o valor esperado numa verificação dependendo de um caso especial	Criar testes individuais dedicados apenas para os casos especiais e excluí-los do teste genérico
Restauração do ambiente é muito complexa e cheia de verificações	Fazer a restauração do ambiente no método tearDown no lugar de deixá-lo dentro do teste
Múltiplos testes condicionais no mesmo teste percorrento uma collection (input, output)	Separando os testes para privilegiar a localização de um problema eventual

Código difícil de testar

Código é muito difícil de testar

Causas	Possíveis Soluções
Código extremamente acoplado	Desacoplar o código e parametrizar as dependências para substituí-las por objetos fake
Código assíncrono	Separar partes síncronas do código assíncrono e testar somente esse código

Duplicação de código

O mesmo código de teste repetido muitas vezes

Causas	Possíveis Soluções
Reproveitamento de código no estilo Copy & Paste	Aplicar padrões de refatoração ao código do teste (ex. Extract Method)
"Reinvenção da Roda" - Escrita de trechos de testes já escrito por outra pessoa	Aplicar padrões de refatoração ao código do teste (ex. Extract Method)

Lógica de teste no código

Código de teste no código sendo testado

Causas	Possíveis Soluções
"Ganchos" para teste: `if testing: ... else: ...`	Substituir o teste lógico por uma dependência que pode ser substituída por um objeto Fake.
Variações: dependências específicas para teste, reimplementações de métodos específicos para testes, etc.	Refatorar o código para eliminar esse tipo de lógica por uma dependência que pode ser substituída por um objeto Fake.

Problemas com comportamento dos testes

Behaviour Smells

Roleta de verificações

Difícil saber qual verificação falhou

Causas	Possíveis Soluções
Teste "fominha": um único teste verifica muitas funcionalidades	Dividir o teste em em vários
Não é possível identificar o problema com a mensagem da verificação quando ela falha.	Acrescentar mensagens nos casos onde a verificação não usa valores constantes/referência como parâmetro. (ex. `assert p1.x == p2.x`, "coordenada x deveria ser igual")

Testes erráticos (I)

Testes se comportam erraticamente

Causas	Possíveis Soluções
Teste depende de outro e falha quando a ordem de execução muda ou o teste quando é executado sozinho	Remover a dependência copiando-a para o teste em questão ou fazendo os dois compartilharem as mesmas fixtures
Mais de um teste roda simultaneamente compartilhando o mesmo ambiente	Cada teste deve rodar em seu próprio ambiente.
Testes vão ficando mais lentos ou ocupando mais recursos da máquina	Se o problema estiver no SUT o certo é corrigir o bug. Se estiver no teste o bug está no processo de tearDown
Testes dependem de recursos externos que podem estar indisponíveis	Criar uma cópia deste recurso localmente ou substituí-lo por um stub.

Testes erráticos (II)

Testes se comportam erraticamente

Causas	Possíveis Soluções
Teste passa na primeira execução e depois falha sucessivamente	O teste não está restaurando o ambiente inicial corretamente.
Teste falha aleatoriamente quando várias pessoas executam testes simultaneamente.	Criar ambientes de teste individuais
Teste falha aleatoriamente por estar testando código não-determinístico	Tentar eliminar, dentro do possível, os elementos não-determinísticos do teste.

Depuração manual

Depuração manual para localizar problemas

Causas	Possíveis Soluções
Existência de código sem cobertura de teste	Providenciar a cobertura do código em questão

Intervenção Manual

Intervenção manual na execução dos testes

Causas	Possíveis Soluções
O teste foi construído sem ter em mente que "teste automatizado" implica que não deve existir "intervenção manual"	Automatizar todo o processo

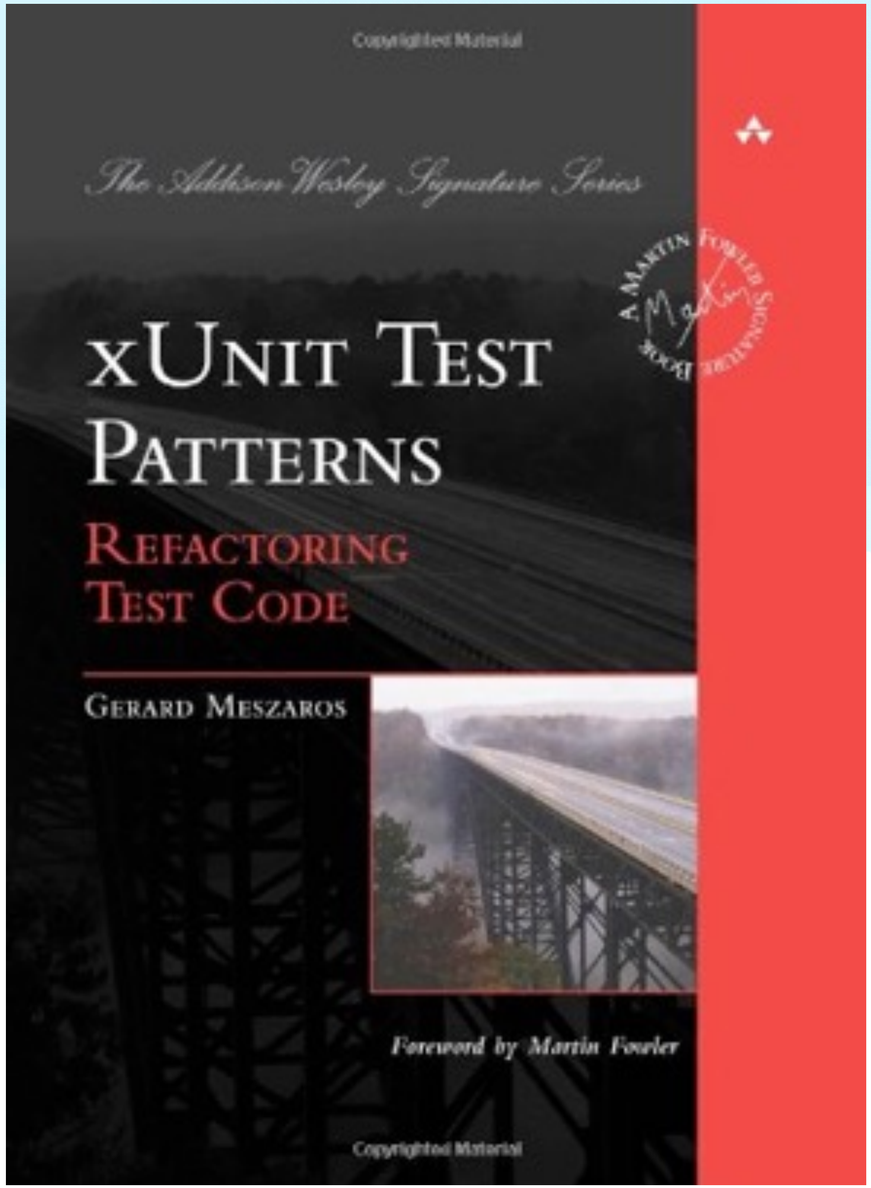
Testes lentos

Testes demoram para executar

Causas	Possíveis Soluções
O teste depende de recursos externos que têm uma latência muito alta	Tentar substituir esse recurso por um objeto fake
Testes executam fixtures muito extensas para cada um dos cenários	A solução ideal é simplificar a construção dessas fixtures. Não sendo possível fazer isso permita que os testes compartilhem as fixtures
Teste acrescenta explicitamente um intervalo para lidar com código assíncrono	Extrair a parte síncrona do código e testar somente ela
Muitos testes	Não é necessário executar todos os testes o tempo todo

Padrões de teste

Práticas e padrões para uso em testes



Padrões de estratégia

Test Strategy Patterns

Estratégia de Automação

- *Recorded Test* — usam a estratégia grava & reproduz. ex. Selenium IDE, Sikuli, VCR, etc
- *Data-Driven Test* — úteis para testar parsers, conversores de formatos, etc. ex. teste de um crawler
- *Scripted Test* — cria-se um programa especificamente para testar outro
- *Test Automation Framework* — é um tipo de programa de testes mas usa um framework para isso. ex. xUnit

Estratégia para fixtures

- *Minimal Fixture* — todo teste precisa de fixtures, com essa estratégia cria-se o mínimo necessário para executar apenas o teste em questão
- *Standard Fixture* — testes compartilham um método que cria as fixtures usadas por eles
- *Fresh Fixture* — teste constrói as suas próprias fixtures
- *Shared Fixture* — testes compartilham as mesmas fixtures

Estratégia de Interação

- *Back Door Manipulation* — nos casos onde não é possível avaliar o funcionamento do SUT diretamente as verificações são feitas com os dados das fixtures. Ex. verificar se o SUT manipulou os dados do banco de dados corretamente. **Evite!**
- *Layer Test* — escrever testes para cada uma das camadas de uma aplicação com várias camadas. Ex. testar o 'driver' do DB, o ORM, os objetos model, etc

Padrões básicos xUnit

xUnit Basics Patterns

Definição dos testes

- *Test Method* — um cenário de teste por método
  - *Four-Phase Test* — setup, exercise, assert, teardown
- *Assertion Method* — métodos de verificação (.assert*())
  - *Assertion Message* — exibida quando a verificação falha
- *Testcase Class* — agrupamento de testes relacionados

Execução dos testes

- *Test Runner* — aplicação que executa os testes e mostra os resultados
- *Testcase object* — instância contendo um conjunto de testes relacionados
- *Test Suite object* — objeto com a mesma interface de `TestCase` que agrupa um ou mais objetos `TestCase`
- *Test Discovery* — mecanismo pelo qual o *Test Runner* encontra os testes

Padrões para setup de fixtures

Fixture Setup Patterns

Setup de Fresh Fixtures

- *In-Line Setup* — cada teste constrói suas fixtures por conta própria
- *Delegated Setup* — testes constroem suas fixtures chamando um método auxiliar
- *Implicit Setup* — a construção das fixtures é implícita e executada dentro do método `.setUp()`

Criação compartilhada

- *Prebuilt Fixture* — as fixtures são compartilhadas pelos testes e são construídas por outro componente. Ex. `./manage.py loaddata data.json` do Django. **Não use isso!!!**
- *Suite Fixture Setup* — fixtures construídas no `.setUp()` da suíte e não no `TestCase`
- *Fixtures Factories* — fixtures construídas por bibliotecas de automação ex. Model Mommy, etc. **Evite!**
  - Precisar de muitas fixtures é um indício de problemas (*smell*). Se não tiver alternativa prefira hierarquia de TestCase + override de `.setUp()`.

Criação compartilhada

- *Fixtures Factories* — fixtures construídas por bibliotecas de automação ex. Model Mommy, etc. **Evite!**
  - Precisar de muitas fixtures é um indício de problemas (*smell*). Cenários de teste precisam ser simples e ter dependências injetadas.
  - Se não tiver alternativa prefira hierarquia de `TestCase` + *override* de `.setUp()`.

Padrões de verificação de resultados

Result Verification Patterns

Estratégia de verificação

- *State Verification* — verificamos o estado do SUT após o exercício. Ex.

```
sut.set(1)
assert sut.get() == 1
```

- *Behaviour Verification* — verificamos os resultados indiretos após o exercício do SUT. Ex.

```
web.open("http://j.mp", mock)
mocker.verify()
```

Estilos de verificação

- *Custom Assertion* — criamos uma verificação personalizada. Ex. `chk(d, r) { assert r == roman(d) }`
- *Delta Assertion* — verificamos a diferença do objeto antes do exercício e depois de exercitá-lo
- *Guard Assertion* — verifica o resultado com if. No caso de erro executa uma falha explicitamente.
- *Unfinished Test Assertion* — força falha pra indicar que o teste não está pronto

Padrões para teardown

Fixture Teardown Patterns

Estratégia para teardown

- *Garbage-Collected Teardown* — deixar o garbage collector da linguagem fazer a limpeza do ambiente
- *Automated Teardown* — registramos a criação de todos os objetos no setup para removê-los na fase de teardown

Organização do código

- *In-line Teardown* — a limpeza do ambiente é feita no próprio teste
- *Implicit Teardown* — a limpeza do ambiente fica por conta do método `.tearDown()`

Padrões com objetos falsos

Test Double Patterns

Exemplo de código

```
from time import localtime as ltime
from time import strftime as ftime

class TimeProvider(object):
    hour = property(lambda self: ltime().tm_hour)
    minute = property(lambda self: ltime().tm_min)
    period = property(lambda self: ftime("%p", ltime()))

class TimeDisplay(object):
    def __init__(self, provider=None):
        if not provider:
            provider = TimeProvider()
        self._provider = provider

    def get_time_as_html(self):
        p = self._provider
        ret = "<span>"
        if p.hour == 0 and p.minute <= 1:
            ret += "Midnight"
        elif p.hour == 12 and p.minute <= 1:
            ret += "Noon"
        else:
            ret += "%02d:%02d %s" % (p.hour, p.minute, p.period)
        return ret + "</span>"
```

Exemplo de código

```
from time import localtime as ltime
from time import strftime as ftime

class TimeProvider(object):
    hour = property(lambda self: ltime().tm_hour)
    minute = property(lambda self: ltime().tm_min)
    period = property(lambda self: ftime("%p", ltime()))

class TimeDisplay(object):
    def __init__(self, provider=None):
        if not provider:
            provider = TimeProvider()
        self._provider = provider

    def get_time_as_html(self):
        p = self._provider
        ret = "<span>"
        if p.hour == 0 and p.minute <= 1:
            ret += "Midnight"
        elif p.hour == 12 and p.minute <= 1:
            ret += "Noon"
        else:
            ret += "%02d:%02d %s" % (p.hour, p.minute, p.period)
        return ret + "</span>"
```

Dependência

Testes falham

```
import unittest

class DisplayTest(unittest.TestCase):
    def test_display_current_time_at_midnight(self):
        sut = TimeDisplay() # setup
        result = sut.get_time_as_html() # exercise SUT
        self.assertEquals('<span>Midnight</span>', result) # verify

    def test_display_current_time_at_noon(self):
        sut = TimeDisplay() # setup
        result = sut.get_time_as_html() # exercise SUT
        self.assertEquals('<span>Noon</span>', result) # verify

    def test_display_current_time_whenever(self):
        sut = TimeDisplay() # setup
        result = sut.get_time_as_html() # exercise SUT
        self.assertEquals('<span>02:30 AM</span>', result) # verify
```

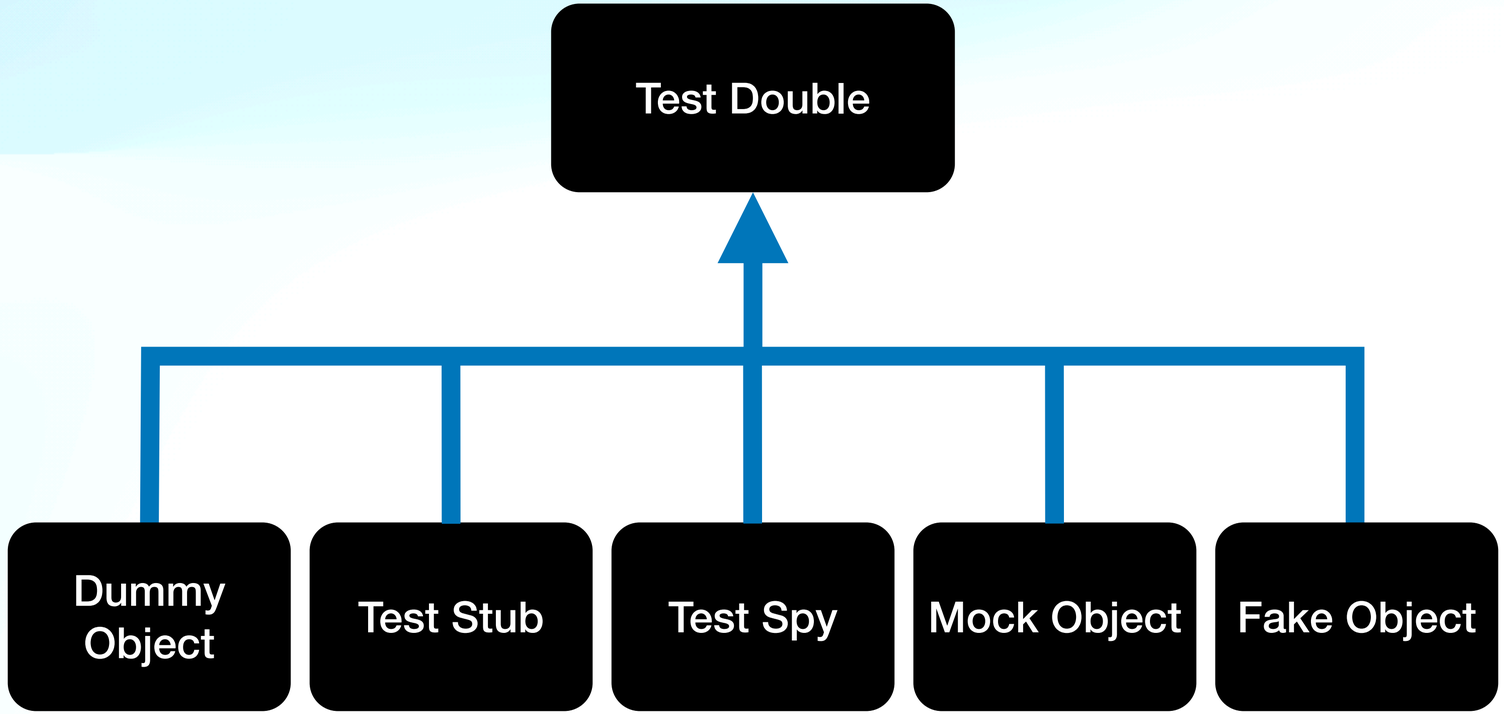
Testes falham

```
import unittest

class DisplayTest(unittest.TestCase):
    def test_display_current_time_at_midnight(self):
        sut = TimeDisplay() # setup
        result = sut.get_time_as_ht                    T
        self.assertEquals('                            sult) # verify

    def test_d
        sut =
        result                          ) # exercise SUT
        self.ass                  >Noon</span>', result) # verify

    def test_display_current_time_whenever(self):
        sut = TimeDisplay() # setup
        result = sut.get_time_as_html() # exercise SUT
        self.assertEquals('<span>02:30 AM</span>', result) # verify
```

FALHAM!

Test Double

- Substituir uma ou mais dependências do SUT por um equivalente específico para o teste:



Dummy

- Geralmente são valores que não são usados no teste.
- Algumas vezes são apenas passados como parâmetros para atender ao número de parâmetros de um método.

Fake

- Uma implementação funcional do objeto original

```
import unittest

class ProviderStub(object):
    def __init__(self, hour, minute, period="AM"):
        self.hour = hour
        self.minute = minute
        self.period = period

class DisplayTest(unittest.TestCase):
    def test_display_current_time_at_midnight(self):
        sut = TimeDisplay(ProviderStub(0,0)) # setup
        self.assertEquals('<span>Midnight</span>', sut.get_time_as_html())

    def test_display_current_time_at_noon(self):
        sut = TimeDisplay(ProviderStub(12,0)) # setup
        self.assertEquals('<span>Noon</span>', sut.get_time_as_html())

    def test_display_current_time_whenever(self):
        sut = TimeDisplay(ProviderStub(2,2)) # setup
        result = sut.get_time_as_html() # exercise SUT
        self.assertEquals('<span>02:02 AM</span>', sut.get_time_as_html())
```

Stubs

- Similares aos objetos Fake
- Não tem uma implementação funcional, apenas retorna valores pré-estabelecidos
- Podem gerar uma exceção para testar comportamento do SUT nessa situação
- Ex: Simular um erro de timeout na conexão com um banco de dados

Spy

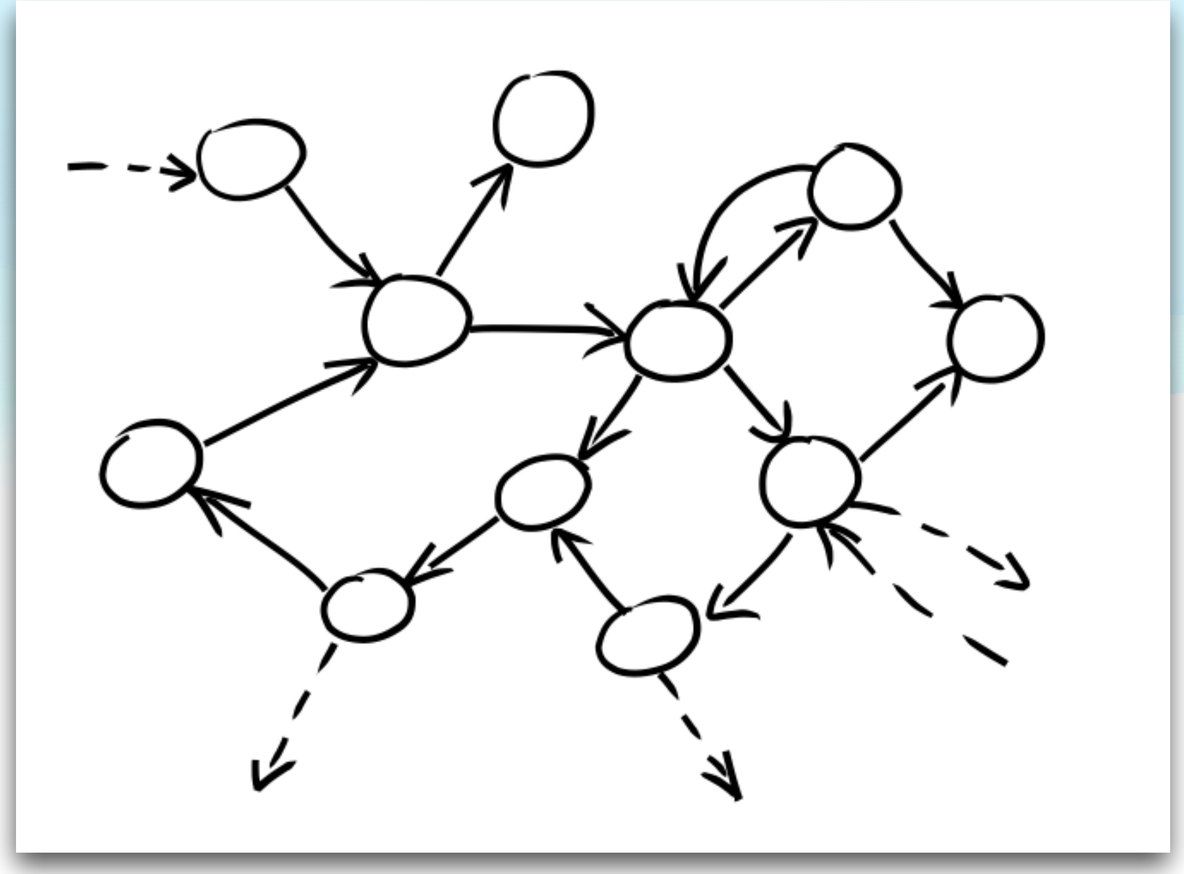
- Similares aos objetos Stub
- Registram chamadas para seus métodos para que seja possível fazer uma verificação indireta posteriormente
- Ex. Servidor de e-mail que registra e-mails "enviados"

Mock

- Tipo especial de objeto que pode ser programado com as expectativas de chamadas e retornos

Mock

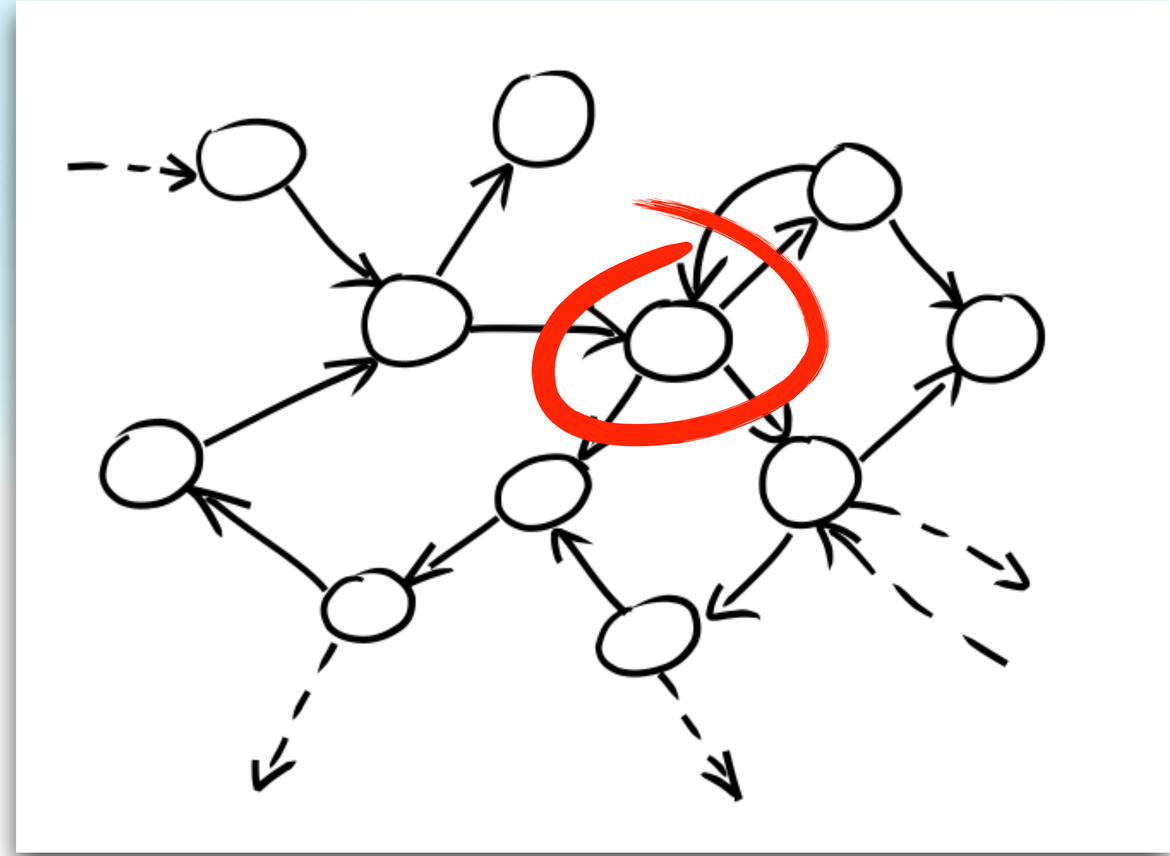
Rede de objetos



Fonte: Growing Object-Oriented Software, Guided by Tests

Mock

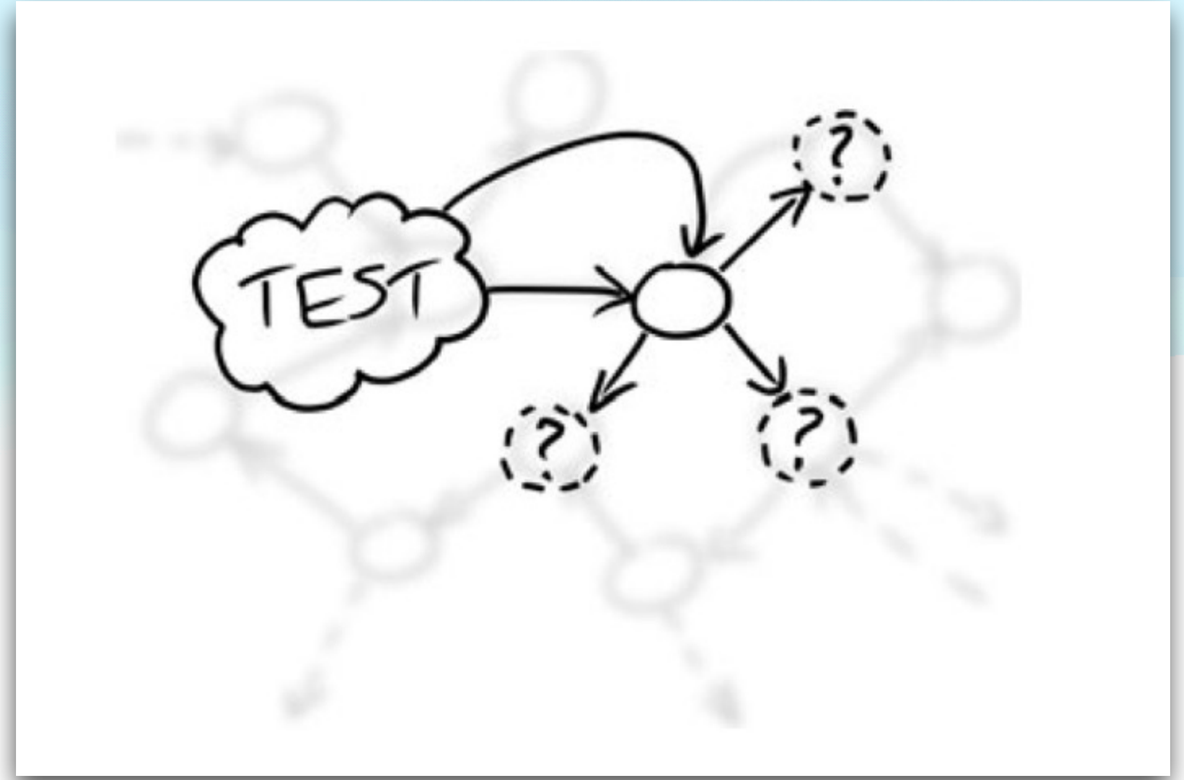
Testando objeto isoladamente



Fonte: Growing Object-Oriented Software, Guided by Tests

Mock

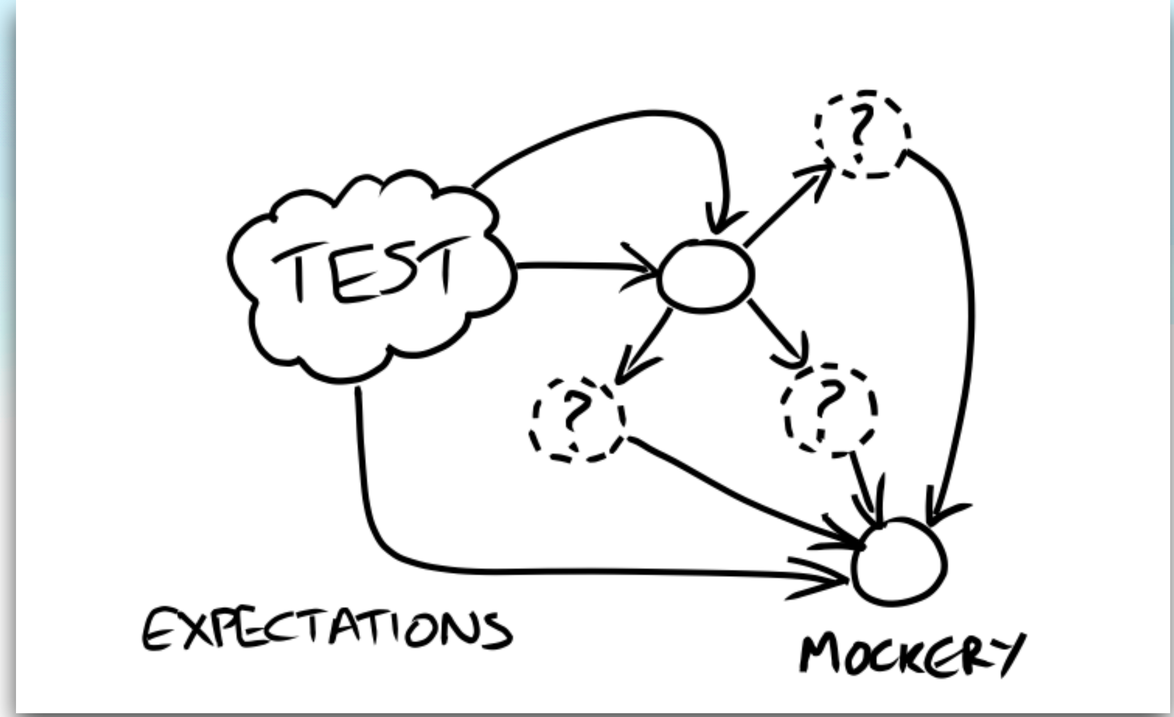
Testando com um objeto mock



Fonte: Growing Object-Oriented Software, Guided by Tests

Mock

Testando com um objeto mock



Fonte: Growing Object-Oriented Software, Guided by Tests

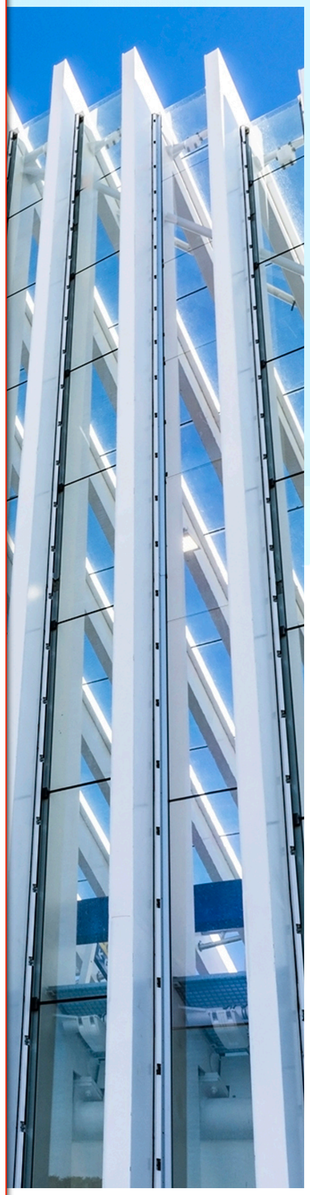
Mocker

- http://labix.org/mocker
- Desenvolvida pelo brasileiro Gustavo Niemeyer
- Usa a estratégia Record (para especificar as expectativas e retornos) & Play (para verificar os resultados)

Mock

Mocker

```
import mocker
class DisplayTest(mocker.MockerTestCase):
    def _get_mock(self, h, m, p="AM"):
        mock = self.mocker.mock()

        mock.hour   # access hour
        self.mocker.count(0, 3); self.mocker.result(h)
        mock.minute # access minute
        self.mocker.count(0, 3); self.mocker.result(m)
        mock.period # access period
        self.mocker.count(0, 3); self.mocker.result(p)

        self.mocker.replay()
        return mock

    def tearDown(self):
        self.mocker.verify()

    def test_display_current_time_at_midnight(self):
        sut = TimeDisplay(self._get_mock(0,0))
        self.assertEquals('<span>Midnight</span>',
                          sut.get_time_as_html())

    def test_display_current_time_at_noon(self):
        sut = TimeDisplay(self._get_mock(12,0))
        self.assertEquals('<span>Noon</span>',
                          sut.get_time_as_html())

    def test_display_current_time_whenever(self):
        sut = TimeDisplay(self._get_mock(2,2))
        self.assertEquals('<span>02:02 AM</span>',
                          sut.get_time_as_html())
```



Padrões para banco de dados

Database Patterns

Padrões de banco de dados

- *Database Sandbox* — cada desenvolvedor tem um banco de dados à sua disposição (ex. Django SQLite in memory)
- *Table Truncate Teardown* — truncar as tabelas na fase do teardown
- *Transaction Rollback Teardown* — iniciar uma transação na fase de setup e efetuar um rollback na fase de teardown. Deve-se cuidar para que não tenha nenhum commit no SUT

Padrões de desenho para testabilidade

Design-for-Testability Patterns

Padrões para testabilidade

- *Dependency Injection* — permite substituir dependências do SUT por *Test Doubles*.
  - A dependência pode ser passada na construção ou como parâmetro do método.
  - Ex. O objeto `TimeDisplay` depende de `TimeProvider` que, nos testes, é substituído por stub/mock
- *Dependency Lookup* — o objeto busca suas dependências em um local específico. Ex. registro, dict

Padrões para testabilidade

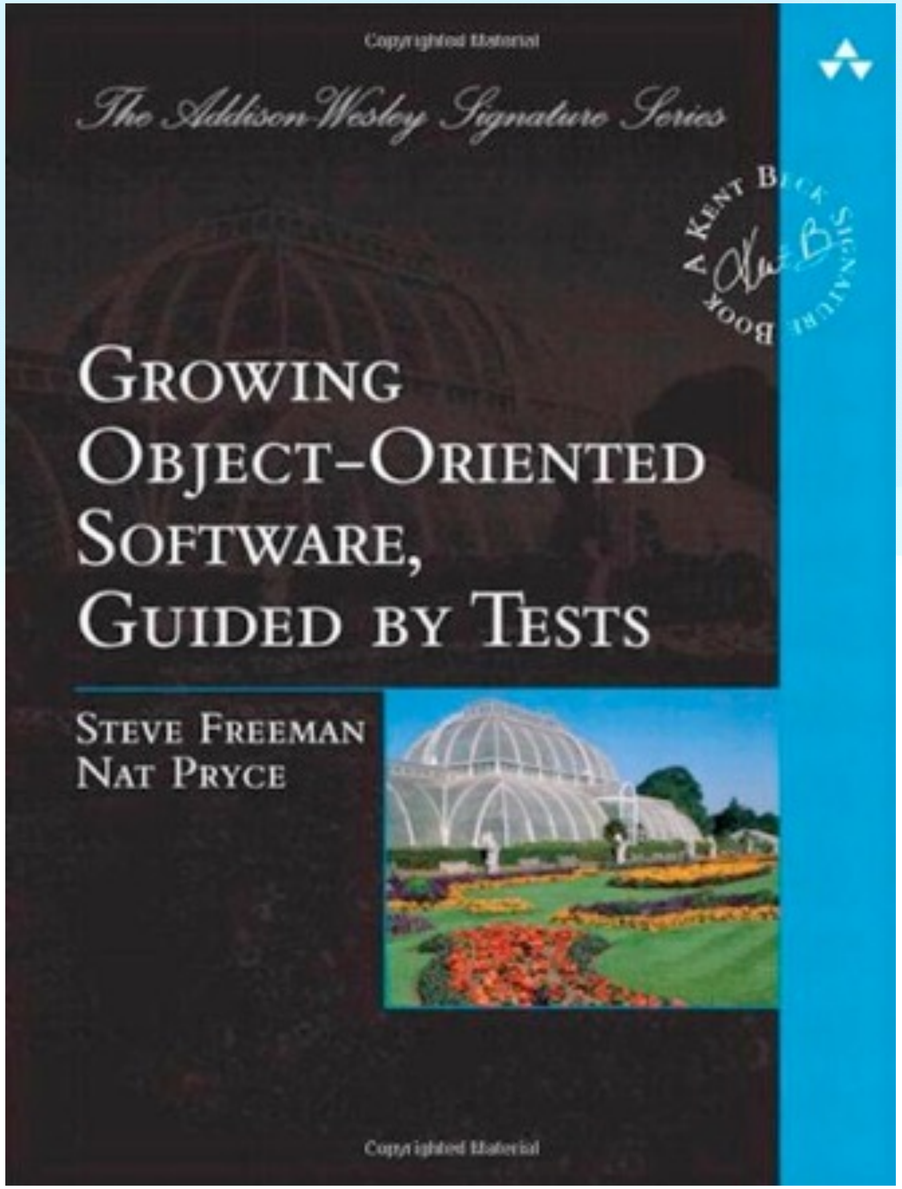
- *Humble object* — extrair a lógica num componente separado e mais fácil de testar.
  - Ex. extrair a parte síncrona de um objeto com operações assíncronas
  - Pode-se usar um método no lugar de um objeto
- *Test Hook* — não use: adicionar lógica condicional no SUT para se comportar de modo específico com os testes

Desenvolvendo aplicações

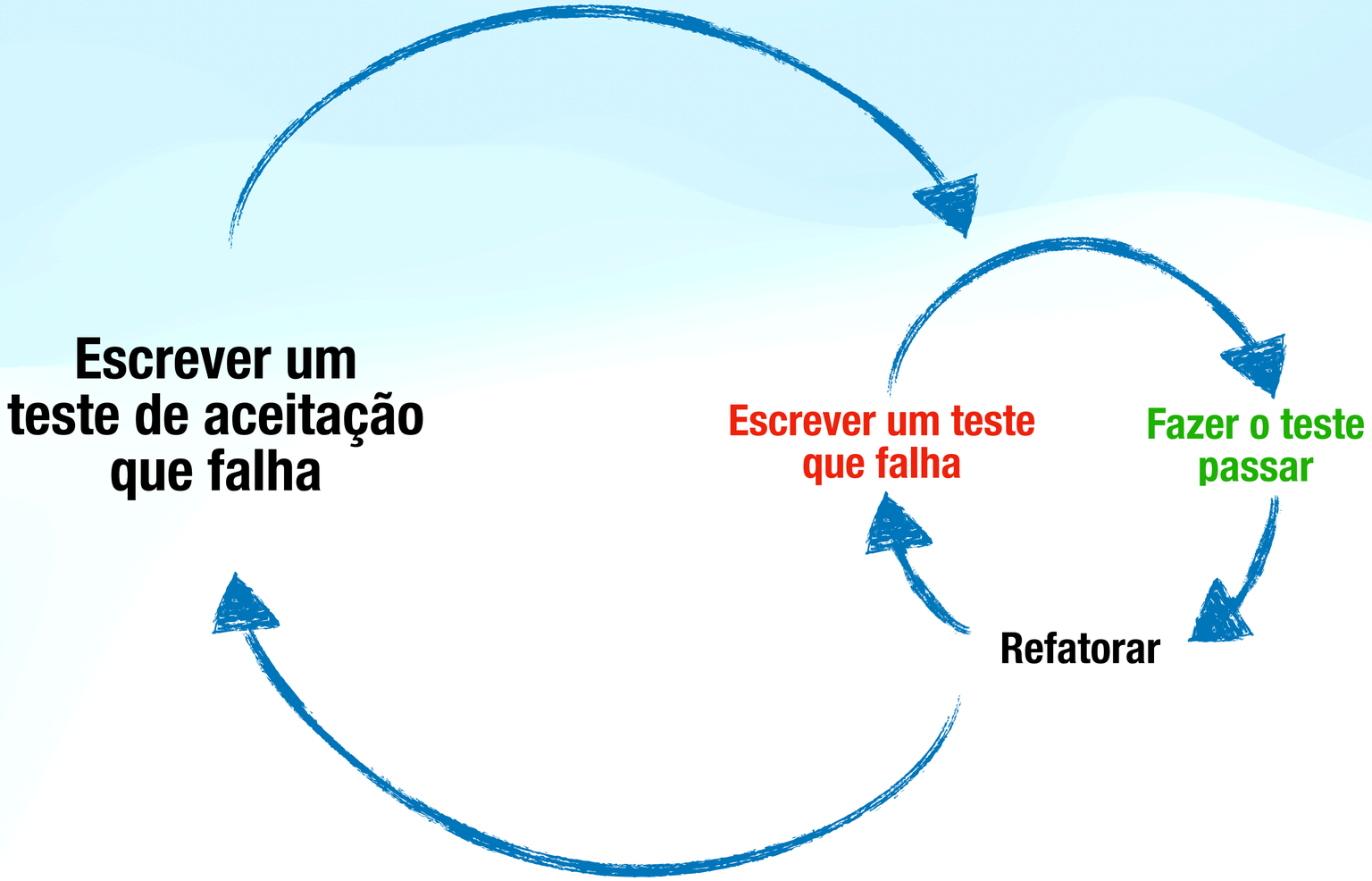
Desenvolvendo aplicações completas usando Test-Driven Development

Desenvolvendo aplicações

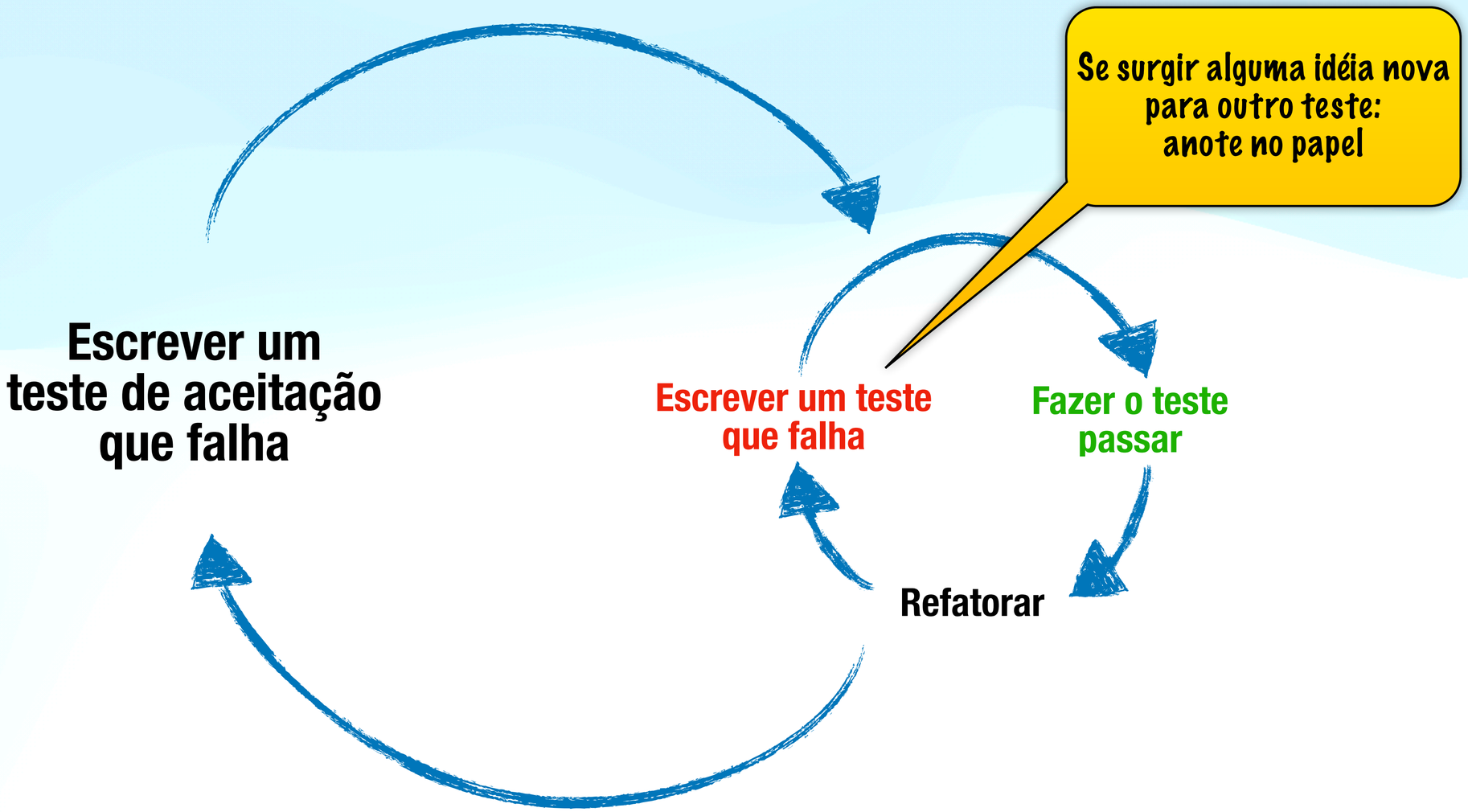
- Testes de aceitação
  - Validar requisitos dos clientes
  - Selenium2, Pyccuracy, Windmill, Django Client, etc.
- Inicia ciclo red/green/refactor



Desenvolvendo aplicações



Desenvolvendo aplicações



Atividade

Pastebin-like

Codb.in

- Usuário submete o código e a linguagem
- Uma URL curta é gerada
- O usuário é encaminhado para o Twitter:
  - http://twitter.com/home?status=???
- Usaremos a biblioteca Pygments
- Google App Engine

Iniciando a atividade

- Ativar o virtualenv:
  - source bin/activate
- Instalar os pacotes NoseGAE, BeautifulSoup e WebTest

Resumo

Sumário dos tópicos abordados

Resumo

- Testes são isolados
- A ordem dos testes não é garantida
- Não se deve adicionar lógica de teste no código de produção
- Testes devem assumir um ambiente limpo quando começam e limpar o ambiente quando terminam

Resumo

- red / green / refactor
- setup, exercise, verify, teardown

F.A.Q.

- Quando sei que os testes são suficientes?
  - Quando você tiver certeza que seu software está completo e sem bugs: nunca serão.
- Quando eu preciso fazer testes:
  - Resposta curta: sempre. Resposta longa: quando você não tiver segurança total daquilo que precisa ser feito

F.A.Q.

- Qual o tamanho ideal de um baby-step?
  - Resposta curta: do tamanho que te dê segurança. Resposta longa: TDD é uma prática e como tal requer treino. O ideal é que no início se use passos pequenos e posteriormente aumentá-los.
- Existe uma relação direta entre a cobertura de testes e a quantidade de bugs num código?
  - Existe essa relação mas ela não é linear.

Mensagens

- Teste é algo desejável num software. Melhor se forem automatizados e ótimos se o código foi feito depois do teste
- Falácia: "Código não testado é código bugado"
  - Não existe bala de prata, logo, teste automatizado não é uma delas
  - Atenção para os "radicais do teste". Radicalismo nunca é bom para um programador

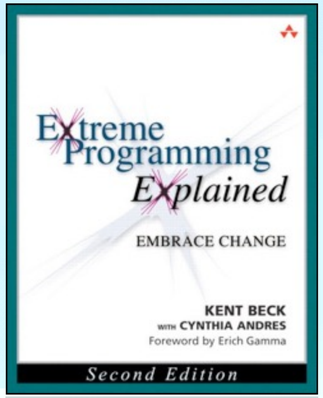
Leitura complementar

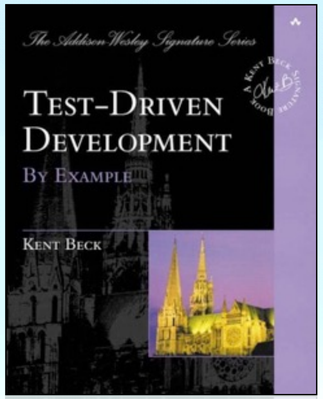
Não terminamos por aqui...

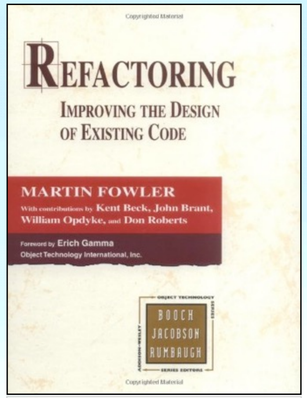
Leitura Complementar

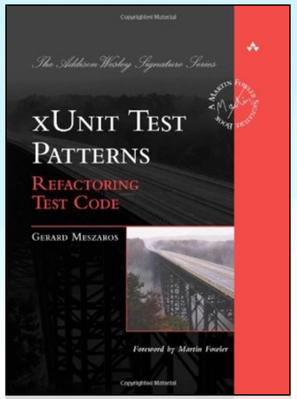
- Internet
  - TDD @ Wikipedia — http://j.mp/zBGgt
  - Mocks aren't Stubs — http://j.mp/7MdzF
  - Inversion of Control and Dependency Injection — http://j.mp/I0YAA

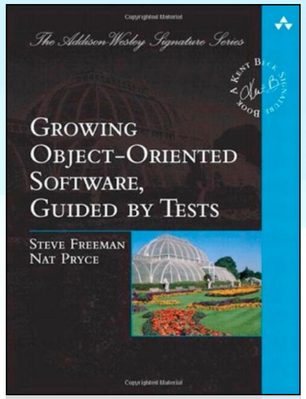
Leitura Complementar

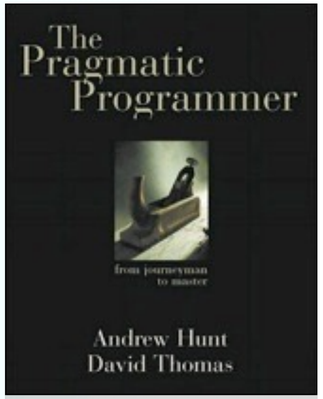


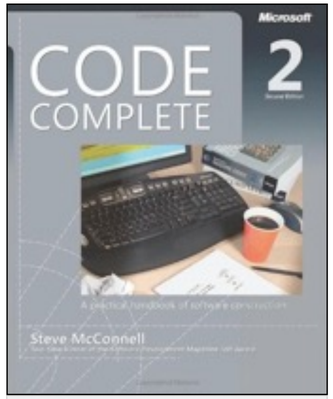












Atividade opcional

Coding-Dojo

Coding-Dojo

O melhor modo de aprender um jogo é jogando

Coding-Dojo

- Escolha do desafio
- Pair programming em uma máquina
- Piloto codifica a solução usando TDD
- Co-piloto troca com piloto em intervalos de 5 minutos
- Todos participam
- Solução construidas na hora (não vale usar bibliotecas dedicadas)
- Design reviews em intervalos
- Piloto deve descrever o que está fazendo

Coding-Dojo

- O piloto pode pedir ajuda para o co-piloto ou para a platéia
- A experiência é mais importante que a solução do problema
- Sessões com tempo fixo
- Análise de pontos positivos e negativos após a sessão
- No Brasil criou-se o #horaextra: uma "happy hour" após o Dojo
- Mais informações:
  http://codingdojo.org/

Sugestões de problemas

- Mão de poker
- Mostrador LCD
- Valor por extenso
- Caixa empilhadas

Mãos de Poker