

# TESTES DE SOFTWARE & TDD

ACH2006 – ENGENHARIA DE SISTEMAS DE INFORMAÇÃO

SIN5005 – TÓPICOS EM ENGENHARIA DE SOFTWARE

---

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

```
require 'spec_helper'

describe MoviesController do
  describe 'searching TMDb' do
    it 'should call the model method that performs TMDb search' do
      expect(Movie).to receive(:find_in_tmdb).with('Inception')
      post :search_tmdb, {:search_terms => 'Inception'}
    end
    it 'should select the Search Results template for rendering' do
      # por hora, o teste só inclui 'It renders Search Results view'
      post :search_tmdb, {:search_terms => 'Inception'}
      response.should render_template('search_tmdb')
    end
  end
end
```

# MOCKS & DUBLÊS

---

# IT 'SHOULD MAKE THE TMDB SEARCH RESULTS AVAILABLE TO THAT TEMPLATE'

- Outra ferramenta do rspec-rails: `assigns()`
  - passa o símbolo que dá nome a uma variável de instância do controlador
  - devolve o valor que o controlador atribuiu àquela variável
- Mas... nosso código atual não define nenhuma variável de instância:

```
def search_tmdb
  @movie = Movie.find_in_tmdb(params[:search_terms])
end
```

- OCQGDT: lista de resultados em `@movie`

```
it 'should select the Search Results template for rendering' do
  fake_results = [mock('Movie'), mock('Movie')]
  Movie.stub(:find_in_tmdb).and_return(fake_results)
  post :search_tmdb, {:search_terms => 'hardware'}
  expect assigns(:movies).to eq(fake_results)
end
```

- **mock** objeto “dublê de ação”
  - verifica se um valor foi propagado corretamente
  - fornece um lugar para armazenar um valor para que o código funcione, mesmo que o teste não dependa daquele valor
  - pode até prover métodos (*stub method*) em um dublê:  
`m=mock('movie1', :title=>'Rambo')`

### Objetivo:

Tal como as emendas, fornecer o *mínimo de funcionalidades necessárias* para testar algum comportamento *específico*.

- Cada spec deve testar *apenas um comportamento*
- Use quantas emendas forem necessárias para isolar o comportamento
- Determine qual tipo de expectativa verificará o comportamento
- Escreva o teste e garanta que ele falhe *pela razão certa*
- Adicione código até que o teste fique verde
- Fique atento para oportunidades de refatorar/embelezar o código

- **Testes de unidade:** comportamento dentro de um método/classe
  - classes colaboradoras são descritas com mocks
  - métodos colaboradores são descritos como *stubs* (nesta classe ou na classe colaboradora)
  - ambos são chamados genericamente de *dublês*
- **Teste funcional:** comportamento entre métodos/classes (múltiplas classes são exercitadas)
  - ex: fluxo do controlador desde o GET/POST até a renderização do *template* (que, na verdade, é um *stub* criado pelo rspec-rails)
  - (logo, não é um teste *full-stack* de verdade)

## FIXTURES & FÁBRICAS

---

- Objetivo: testar a busca de filmes por diretor ou por prêmio recebido

```
expect(m.award.type).to eq 'Oscar'  
expect(m.director.name).to eq 'Abrams'
```

- Configuração do mock:

```
a = mock('Award', :type => 'Oscar')  
d = mock('Director', :name => 'JJ Abrams')  
m = mock('Movie',  
  :award => a,  
  :director => d)
```

## QUANDO VOCÊ PRECISA DE UM OBJETO REAL...

```
fake_movie = mock('Movie')  
fake_movie.stub(:title).and_return('Casablanca')  
fake_movie.stub(:rating).and_return('PG')  
fake_movie.name_with_rating.should == 'Casablanca (PG)'
```

Criar esse mock dá o mesmo trabalho de se criar um objeto real!

## QUANDO VOCÊ PRECISA DE UM OBJETO REAL...

```
fake_movie = mock('Movie')  
fake_movie.stub(:title).and_return('Casablanca')  
fake_movie.stub(:rating).and_return('PG')  
fake_movie.name_with_rating.should == 'Casablanca (PG)'
```

Criar esse mock dá o mesmo trabalho de se criar um objeto real!

... de onde você o pega?

**Fixture** pré-carrega estaticamente alguns dados em tabelas do banco de dados

**Fábrica** cria apenas o que você precisa em cada teste

- (termo que vem da manufatura, em inglês significa “instalação de ensaio”)
- o banco de dados é zerado e recarregado com as *fixtures* antes de *cada spec*
- Prós / usos:
  - dados que nunca mudam, ex: informação de configuração que nunca muda
  - fácil de ver todos os dados do teste em um único lugar
- Contras / razões para não usar:
  - pode criar dependência do código de testes aos dados da fixture

## FIXTURE — EXEMPLO

```
# spec/fixtures/movies.yml
```

```
milk_movie:
```

```
  id: 1
```

```
  title: Milk
```

```
  rating: R
```

```
  release_date: 2008-11-26
```

```
documentary_movie:
```

```
  id: 2
```

```
  title: Food, Inc.
```

```
  release_date: 2008-09-07
```

---

```
# spec/models/movie_spec.rb:
```

```
require 'rails_helper'
```

```
describe Movie do
```

```
  fixtures :movies
```

```
  it 'includes rating and year in full name' do
```

```
    movie = movies(:milk_movie)
```

```
    expect(movie.name_with_rating).to eq('Milk (R)')
```

```
  end
```

```
end
```

- Define métodos auxiliares para criação de objetos com atributos padrão
- Prós / usos:
  - mantém os testes Independentes: não são afetados pela presença de objetos que não importam ao teste
- Contras / razões para não usar:
  - Relações complexas podem ser difíceis de expressar (mas também podem indicar grande acoplamento no código)
- Gema: **FactoryBot**

# EXEMPLO: GEMA FACTORYBOT (NO LIVRO CHAMADA DE FACTORYGIRL)

```
# spec/factories/movie.rb
```

```
FactoryBot.define do
  factory :movie do
    title 'A Fake Title' # default values
    rating 'PG'
    release_date { 10.years.ago }
  end
end
```

---

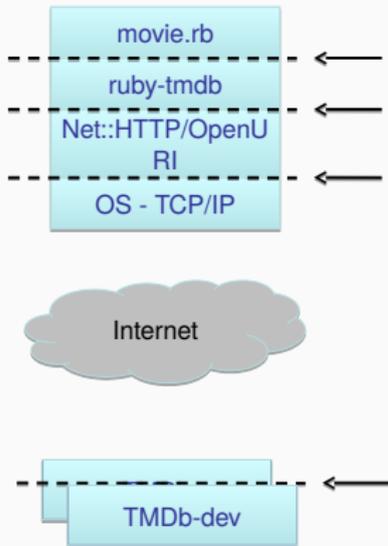
```
# in spec/models/movie_spec.rb
describe Movie do
  it 'should include rating and year in full name' do
    # 'build' creates but doesn't save object; 'create' also saves it
    movie = FactoryBot.build(:movie, :title => 'Milk', :rating => 'R')
    expect(movie.name_with_rating).to eq 'Milk (R)'
  end
end
```

# TDD PARA O MODELO & STUBS PARA A INTERNET

---

- `find_in_tmdb` deveria chamar a gema `TmdbRuby` com as palavras-chave do título
  - se não tivéssemos uma gem, deveria submeter uma requisição a uma API RESTful para o site remoto TMDb
- e se a gema indicar algum erro? Ex: chave de API inválida?

# ONDE DEVEMOS COLOCAR STUBS EM UMA ARQUITETURA ORIENTADA A SERVIÇOS?



Regra geral:

- para testes de unidade, crie o *stub* o mais próximo possível do código sendo testado, para o máximo de isolamento da classe sendo testada (Rápido, Independente)
- para testes de integração, crie o *stub* o mais longe possível para testar o maior número de interfaces possível

# COBERTURA, TESTES DE UNIDADE VS. INTEGRAÇÃO

---

## COMO SABER SE TEMOS TESTES O SUFICIENTE?

- Péssimo: “até que seja hora de entregar o código”
- Um pouco melhor:  $\frac{\text{Linhas de teste}}{\text{Linhas de código}}$ 
  - 1,2 – 1,5 parece razoável
  - em geral, a razão é *muito maior* em sistemas em produção
- Pergunta melhor: “quão abrangente é o meu teste?”
  - métodos formais
  - medidas de cobertura
  - vamos nos focar no segundo, mas o primeiro está ganhando popularidade <sup>1</sup>

---

<sup>1</sup>Veja o artigo How Amazon Web Services Uses Formal Methods em <http://dx.doi.org/10.1145/2699417>

```
class MyClass
  def foo(x,y,z)
    if x
      if (y && z) then bar(0) end
    else
      bar(1)
    end
  end
  def bar(x) ; @w = x ; end
end
```

- S0: todo método é chamado
- S1: todo método é chamado em todos os lugares possíveis?
- C0: todas as expressões foram executadas?
  - gema Ruby SimpleCov
- C1: todos os ramos foram testados em todas as direções?
- C1 + cobertura de decisão: todas as subexpressões nos condicionais
- C2: todos os caminhos (difícil e controverso sobre o valor de conseguir isso)

# QUE TIPOS DE TESTES?

## Unitário (um método/classe)

- roda rápido
- boa cobertura
- granularidade fina
- requer muitos mocks e não testa interfaces
- ex: **specs do modelo**

## Funcional ou módulo (alguns métodos/classes)

- ex: **specs de controladores**

## Integração /sistema

- poucos mocks
- testa a interface
- roda devagar
- menor cobertura
- granularidade grossa
- ex: **cenários do Cucumber**

- X Chutei o pau da barraca, já “tá” funcionando!

- X Chutei o pau da barraca, já “tá” funcionando!
- X Não entregue o app até que esteja 100% coberto & verde

- ✗ Chutei o pau da barraca, já “tá” funcionando!
- ✗ Não entregue o app até que esteja 100% coberto & verde
- Use cobertura para identificar código que ainda não foi testado ou que foi mal testado

- **X** Chutei o pau da barraca, já “tá” funcionando!
- **X** Não entregue o app até que esteja 100% coberto & verde
- Use cobertura para identificar código que ainda não foi testado ou que foi mal testado
- **X** Concentre-se nos testes de unidade, eles são mais abrangentes

- ✗ Chutei o pau da barraca, já “tá” funcionando!
- ✗ Não entregue o app até que esteja 100% coberto & verde
- Use cobertura para identificar código que ainda não foi testado ou que foi mal testado
- ✗ Concentre-se nos testes de unidade, eles são mais abrangentes
- ✗ Concentre-se nos testes de integração, eles são mais realísticos

- ✗ Chutei o pau da barraca, já “tá” funcionando!
- ✗ Não entregue o app até que esteja 100% coberto & verde
- Use cobertura para identificar código que ainda não foi testado ou que foi mal testado
- ✗ Concentre-se nos testes de unidade, eles são mais abrangentes
- ✗ Concentre-se nos testes de integração, eles são mais realísticos
- Cada teste encontra bugs que o outro não encontra

## OUTROS CONCEITOS SOBRE TESTES; TESTES VS. DEPURAÇÃO

---

- Cada exemplo do RSpec é executado em uma transação do banco de dados; as modificações são *rolled back* depois do teste
  - dados criados em `before(:each)` também são *rolled back*
  - porém, `before(:all)` é executado fora da transação
- Sintoma: “testes flutuantes” (passam, depois falham)
  - ou os specs passam quando executados sozinhos, mas falham quando executados em conjunto com outros specs (ou seja, não são Independentes)
  - use `rake db:test:prepare` para limpar o BD, e então reexecute os testes
- Evite usar `before(:all)` se possível

## OUTROS TIPOS DE TESTES QUE VOCÊ PODE TER OUVIDO FALAR

- Teste de Mutação: se introduzirmos um erro deliberadamente no código, algum teste quebra?
- Teste de *fuzz*: 10.000 macacos jogam alguma entrada aleatória no seu programa:
  - consegue encontrar 20% dos bugs da Microsoft, quebra 25% dos utilitários Unix
  - pega um app e testa *aquilo que ele não foi feito pra fazer*
- Cobertura DU: cada par <define x/ usa x> é executado no código?
- Funcional (caixa-preta) vs. Estrutural (caixa-branca/caixa de vidro)

# TDD VS. DEPURAÇÃO TRADICIONAL

Tradicional	TDD
Escreva dezenas de linhas, rode, encontre um bug, dispare o depurador	Escreva algumas linhas, com teste primeiro; descubra imediatamente se está quebrado
Insira <code>printfs</code> para imprimir variáveis a cada execução do programa	Teste pequenos pedaços de código usando expectativas
Pare o depurador e mexa nas variáveis para controlar o caminho do código	Use mocks e stubs para controlar o caminho do código
Droga, achei que já tivesse corrigido isso! Toca corrigir de novo	Execute os testes novamente, automaticamente

- Lição 1: TDD usa as mesmas habilidades e técnicas que depuração tradicional, mas é mais produtivo (FIRST)
- Lição 2: escrever testes *antes* do código demora mais tempo inicialmente, mas *menos tempo* no total



- TDD é difícil no começo, mas fica mais fácil depois
- Foi ótimo para notar erros [regressão] rapidamente e facilitou consertá-los
- Me ajudou a organizar os meus pensamentos e o meu código [o código que você gostaria de ter]



- Gostaríamos de ter nos comprometido com ele mais cedo & mais agressivamente
- Nem sempre testávamos antes de fazer *push*, e isso nos causou muito sofrimento



- Uma boa cobertura nos dava confiança de que não estávamos quebrando nada a cada nova implantação
- Nos sentimos ótimos ao ver uma nota boa no CodeClimate
- O modelo de *pull-request* para revisões constantes do código fez nosso código ter qualidade melhor



- Gostaríamos de ter nos comprometido com TDD & cobertura mais cedo

- Vermelho–Verde–Refatore
- Teste *um* comportamento por vez, usando emendas
- Descreva os testes de que precisará usando `it`
- Leia & entenda relatórios de cobertura
- “Defesa em profundidade”: não dependa muito de um só tipo de teste