

Testes de software & TDD

SIN5005 — Tópicos em Engenharia de Software

Daniel Cordeiro

2 de outubro de 2019

Escola de Artes, Ciências e Humanidades | EACH | USP

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan, *The Elements of Programming Style*

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger W. Dijkstra, *Notes On Structured Programming*

Antes

- desenvolvedores terminavam de escrever o código e faziam alguns testes *ad hoc*
- código era “jogado” pro pessoal de Quality Assurance (QA)
- pessoal de QA mexia manualmente no programa

Antes

- desenvolvedores terminavam de escrever o código e faziam alguns testes *ad hoc*
- código era “jogado” pro pessoal de Quality Assurance (QA)
- pessoal de QA mexia manualmente no programa

Hoje/Ágil

- teste é parte de **todas** as iterações Ágil
- desenvolvedores testam **o seu próprio** código
- ferramentas & processos de testes totalmente **automatizados**
- equipe de testes/QA encarregada de melhorar as ferramentas e testabilidade do código

A qualidade do software é resultado de um bom processo e não a responsabilidade de um grupo específico.

Projeto guiado por comportamento (BDD)

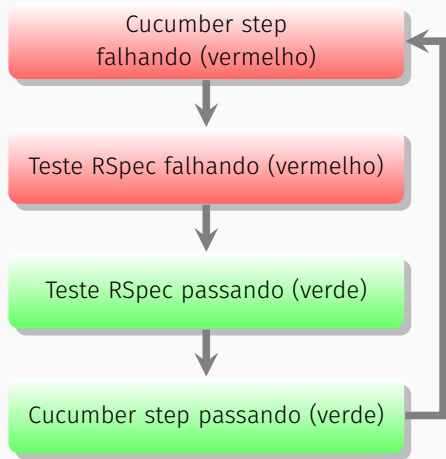
- desenvolva histórias de usuários (as funcionalidades que você quer ter) para descrever como o app irá funcionar
- usando Cucumber, histórias de usuários viram *testes de aceitação* e *testes de integração*

Desenvolvimento guiado por testes (TDD)

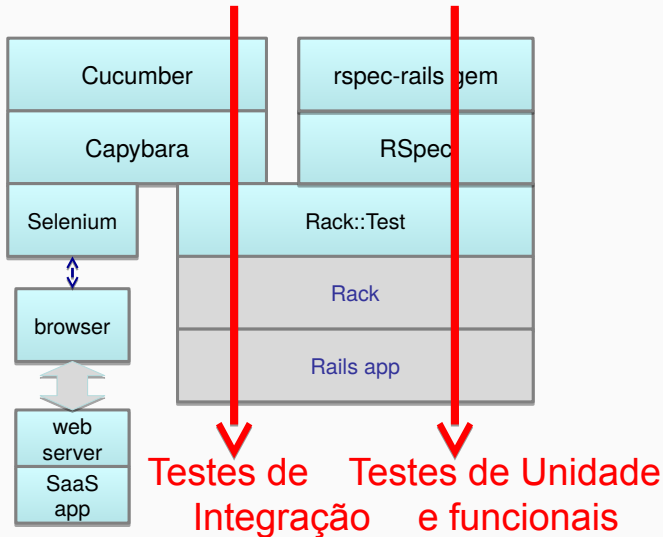
- cada *definição de passo* para uma nova história pode precisar que se desenvolva novo código
- TDD advoga que: escreva os testes de unidade & funcionais primeiro, **antes** de escrever o código
- ou seja, escreva testes para **o código que você gostaria de ter**

Cucumber & RSpec

- Cucumber descreve o *comportamento* com as funcionalidades & cenários (projeto guiado pelo comportamento)
- RSpec testa os módulos individuais que contribuem com esses comportamentos (desenvolvimento guiado por testes)



Pilha de testes



FIRST, TDD e RSpec

Testes de unidade devem ser FIRST

F ast (rápido)

I ndependent (independente)

R epeatable (repetível)

S elf-checking (autoverificável)

T imely (oportuno)

Testes de unidade devem ser FIRST

Rápido rodar (um subconjunto dos) testes deve ser rápido (já que você vai fazer isso o tempo todo)

Testes de unidade devem ser FIRST

Rápido rodar (um subconjunto dos) testes deve ser rápido (já que você vai fazer isso o tempo todo)

Independente testes não devem depender uns dos outros, você deve poder rodá-los quaisquer testes em qualquer ordem

Testes de unidade devem ser FIRST

Rápido rodar (um subconjunto dos) testes deve ser rápido (já que você vai fazer isso o tempo todo)

Independente testes não devem depender uns dos outros, você deve poder rodá-los quaisquer testes em qualquer ordem

Repetível N execuções sempre devem produzir o mesmo resultado (para ajudar a isolar bugs e permitir a automação)

Testes de unidade devem ser FIRST

Rápido rodar (um subconjunto dos) testes deve ser rápido (já que você vai fazer isso o tempo todo)

Independente testes não devem depender uns dos outros, você deve poder rodá-los quaisquer testes em qualquer ordem

Repetível N execuções sempre devem produzir o mesmo resultado (para ajudar a isolar bugs e permitir a automação)

Autoverificável testes devem poder detectar por si mesmos se foram bem sucedidos (não deve haver uma pessoa para verificar os resultados)

Testes de unidade devem ser FIRST

Rápido rodar (um subconjunto dos) testes deve ser rápido (já que você vai fazer isso o tempo todo)

Independente testes não devem depender uns dos outros, você deve poder rodá-los quaisquer testes em qualquer ordem

Repetível N execuções sempre devem produzir o mesmo resultado (para ajudar a isolar bugs e permitir a automação)

Autoverificável testes devem poder detectar por si mesmos se foram bem sucedidos (não deve haver uma pessoa para verificar os resultados)

Oportuno escrito quase que ao mesmo tempo que o código que será testado (com TDD, é escrito antes do código!)

- Linguagem específica de domínio (DSL) para testes.
 - DSL são pequenas linguagens de programação que simplificam uma tarefa, mas que normalmente são menos generalizáveis
 - Exs: migrações, expressões regulares, SQL
- Testes em RSpec são chamados de specs ou exemplos
- Para rodar os testes em um arquivo: `rspec <arquivo>`
- Ou melhor: use `guard/autotest`

Exemplo de RSpec

```
expect { k += 1.05 }.to change { k }.by( a_value_within(0.1).of(1.0) )

expect { s = "barn" }.to change { s }
  .from( a_string_matching(/foo/) )
  .to( a_string_matching(/bar/) )

expect(["barn", 2.45]).to contain_exactly(
  a_value_within(0.1).of(2.5),
  a_string_starting_with("bar")
)

expect(["barn", "food", 2.45]).to end_with(
  a_string_matching("foo"),
  a_value > 2
)

expect(["barn", 2.45]).to include( a_string_starting_with("bar") )

expect(:a => "food", :b => "good").to include(:a => a_string_matching(/foo/))
```

<https://github.com/rspec/rspec-expectations>

```

require 'ruby_intro.rb'

describe BookInStock do
  it "should be defined" do
    expect { BookInStock }.not_to raise_error
  end

  describe 'getters and setters' do
    before(:each) { @book = BookInStock.new('isbn1', 33.8) }
    it 'sets ISBN' do
      expect(@book.isbn).to eq('isbn1')
    end
    it 'sets price' do
      expect(@book.price).to eq(33.8)
    end
    it 'can change ISBN' do
      @book.isbn = 'isbn2'
      expect(@book.isbn).to eq('isbn2')
    end
    it 'can change price' do
      @book.price = 300.0
      expect(@book.price).to eq(300.0)
    end
  end
end

expect { lambda }.to(assertion)
expect(expression).to(assertion)

```

Pergunta

Quais tipos de código podem ser testados de forma **Repetível e Independente**?

1. Código que depende de aleatoriedade (ex: misturar um baralho de cartas)
2. Código que depende do horário do dia (ex: faz backup todo domingo à meia-noite)

Resposta:

- só (1)
- só (2)
- tanto (1) quanto (2)
- nem (1) nem (2)

RSpec-Rails & o ciclo TDD: Vermelho-Verde-Refatore

Testes de unidade devem ser FIRST

F ast (rápido)

I ndependent (independente)

R epeatable (repetível)

S elf-checking (autoverificável)

T imely (oportuno)

Rápido rodar (um subconjunto dos) testes deve ser rápido (já que você vai fazer isso o tempo todo)

Independente testes não devem depender uns dos outros, você deve poder rodá-los quaisquer testes em qualquer ordem

Repetível N execuções sempre devem produzir o mesmo resultado (para ajudar a isolar bugs e permitir a automação)

Autoverificável testes devem poder detectar por si mesmos se foram bem sucedidos (não deve haver uma pessoa para verificar os resultados)

Oportuno escrito quase que ao mesmo tempo que o código que será testado (com TDD, é escrito antes do código!)

Desenvolvendo o teste primeiro

- Pense naquilo que seu código *deveria* fazer
- Capture a ideia em um teste, que irá falhar
- Escreva o código mais simples possível que faria o código do teste passar
- Refatore. Simplifique o que for comum a vários testes
- Continue com a próxima coisa que o código deveria fazer

Vermelho–Verde–Refatore

Tente “sempre ter código funcionando”

O que é específico de Rails?

- Métodos adicionais *mixed*¹ no RSpec para permitir testes de coisas específicas do Rails:
 - ex: **get**, **post**, **put** para testar controladores
 - objeto **response** também pros controladores
- *Matchers* para testar o comportamento dos apps Rails:
 - `expect(response).to render_template("movies/index")`
- Permite criar vários dublês (*doubles*) para testes em métodos mais complicados

¹*Mixins* em Ruby são uma forma de emular herança múltipla

Exemplo

- Imagine uma nova funcionalidade: adicionar filme usando info importada do TMDb
- Como os passos da história de usuário deveriam se comportar?

```
When I fill in "Search Terms" with "Inception"  
And I press "Search TMDb"  
Then I expect to be on the RottenPotatoes homepage  
...
```

Lembre-se que em Rails

Adicionar uma nova funcionalidade implica em adicionar uma nova rota + novo método de controlador + nova visão.

Como testar alguma coisa “de forma isolada”
se ele tiver *dependências* que podem afetar os
testes?

O código que gostaríamos de ter

O que o método do controlador deveria fazer quando receber os dados do formulário de busca?

1. chamar um método que irá procurar no TMDb pelo filme especificado
2. se o filme for encontrado, selecione (nova) visão “Search Results” para mostrar os dados do filme
3. se não for encontrado, redirecionar para a página principal com uma mensagem de erro

O código que gostaríamos de ter

```
require 'rails_helper'

describe MoviesController do
  describe 'searching TMDb' do
    it 'calls the model method that performs TMDb search'
    it 'selects the Search Results template for rendering'
    it 'makes the TMDb search results available to that template'
  end
end
```

TDD para a ação do controlador

- Adicione uma rota a `config/routes.rb`:
`post '/movies/search_tmdb` (convenção ao invés de configuração fará o rails mapear a rota para `MoviesController#search_tmdb`)
- Crie uma visão vazia:
`$ touch app/views/movies/search_tmdb.html.haml`
- Crie um método em `movies_controller.rb` vazio:

```
def search_tmdb
end
```

E o método do modelo?

- Chamar o TMDb é responsabilidade do modelo... mas ainda não existe um método de modelo!
- Sem problemas... nós vamos fazer uma chamada “falsa” ao método que nós gostaríamos de ter, `Movie.find_in_tmdb`
- Esquema geral:
 - Simular o **POST** de um formulário de busca para a ação do controlador
 - Verificar que a ação do controlador irá tentar chamar `Movie.find_in_tmdb` com os dados desse formulário de busca
 - O teste irá **falhar** porque o controlador não vai chamar o método `find_in_tmdb` (ele ainda está vazio!)
 - Arrumar a ação do controlador e fazer o teste ficar **verde**

```
require 'rails_helper'

describe MoviesController do
  describe 'searching TMDb' do
    it 'calls the model method that performs TMDb search' do
      expect(Movie).to receive(:find_in_tmdb).with('hardware')
      post :search_tmdb, {:search_terms => 'hardware'}
    end
    it 'selects the Search Results template for rendering'
    it 'makes the TMDb search results available to that template'
  end
end
```

Emendas

Emendas ("seams")

- Um lugar onde você pode mudar o comportamento do app sem mudar seu código fonte (ideia do livro , *Working Effectively With Legacy Code* de Michael Feathers)
- Útil para testes: *isola* o comportamento de um código do comportamento de outro código do qual ele depende
- `expect( ).to receive` usa as classes abertas de Ruby para criar uma emenda que irá isolar a ação do controlador do comportamento (vazio ou bugado) de `Movie.find_in_tmdb`
- RSpec reinicializa todos os *mocks* e `stubs` depois de cada exemplo (para manter os testes Independentes)

Como tornar esse RSpec verde?

- A expectativa diz que a ação do controlador deveria chamar o método `Movie.find_in_tmdb`
- Então basta chamá-lo:

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

- Dizemos que o spec *incitou a criação do método* do controlador para fazer o teste passar
- Mas será que o método não deveria devolver alguma coisa?

Expectativas

Onde estamos & para onde vamos: desenvolvimento “de fora pra dentro”

- Foco: escreva expectativas que incitem o desenvolvimento do método do controlador
 - Acabamos descobrindo que devemos *colaborar* com um método do modelo
 - Use essa ideia recursivamente: crie um *stub* para o método do modelo no teste, escreva-o depois
- **Ideia principal:** **quebre a dependência** entre o método que está sendo testado e seus colaboradores
- **Conceito básico:** **emenda** — lugar onde você pode mudar o comportamento do código sem modificá-lo

O código que gostaríamos de ter

O que o método do controlador deveria fazer quando receber os dados do formulário de busca?

1. chamar um método que irá procurar no TMDb pelo filme especificado
2. se o filme for encontrado, selecione (nova) visão “Search Results” para mostrar os dados do filme
3. se não for encontrado, redirecionar para a página principal com uma mensagem de erro

When I fill in "Search Terms" with "Inception"
And I press "Search TMDb"

- Resolvido: a interação com o TMDb é responsabilidade de um método fictício `Movie.find_in_tmdb`
- Resolvido: a ação do controlador que lida com a submissão do formulário de busca deve chamar esse método
- Conseguimos: o spec do controlador que verifica isso (usando o `to receive`) não depende do modelo
- Falta fazer: specs do controlador para os comportamentos restantes da ação do controlador

Inserindo uma emenda para find_in_tmdb

```
it 'calls the model method that performs TMDb search' do
  expect(Movie).to receive(:find_in_tmdb).with('hardware')
  post :search_tmdb, {:search_terms => 'hardware'}
end
```

- enquanto isso, no controlador:

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

- Dizemos que o spec *incitou a criação do método* do controlador para fazer o teste passar
- Mas será que o método `find_in_tmdb` não deveria devolver alguma coisa?

it 'selects the Search Results template for rendering'

Na verdade há 2 specs aqui:

1. *It renders Search Results view*

- mais importante ainda se o sistema tiver visões diferentes que devem ser usadas dependendo do resultado

2. *It makes list of matches available to view*

- construção básica de uma expectativa: `expect(obj).to`
`matched-condition`
- use um dos *matchers* embutidos, ou defina seu próprio

- Depois de chamar `post :search_tmdb`, o método `response()` devolverá ao controlador um objeto do tipo *response*.
- o *matcher* `render_template` pode verificar qual visão o controlador tentou renderizar

```

require 'spec_helper'

describe MoviesController do
  describe 'searching TMDb' do
    before :each do
      @fake_results = [mock('movie1'), mock('movie2')]
    end
    it 'should call the model method that performs TMDb search' do
      Movie.should_receive(:find_in_tmdb).with('hardware').
        and_return(@fake_results)
      post :search_tmdb, {:search_terms => 'hardware'}
    end
    it 'should select the Search Results template for rendering' do
      Movie.stub(:find_in_tmdb).and_return(@fake_results)
      post :search_tmdb, {:search_terms => 'hardware'}
      response.should render_template('search_tmdb')
    end
    it 'should make the TMDb search results available to that template'
  end
end

```

Note que:

- a visão precisa existir (e precisará de dados para funcionar)
- **post :search_tmdb** irá exercitar todo o fluxo MVC, inclusive a renderização. Teste funcional (e não unitário)

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
```

Novo conceito de emenda: mock/dublê

- **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*.

Receitas de RSpec #1

- 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 vs. Funcionais em apps SaaS

- **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)