FIXTURES & FÁBRICAS

ENGENHARIA DE SISTEMAS DE INFORMAÇÃO

Daniel Cordeiro

17 de outubro de 2017

Escola de Artes, Ciências e Humanidades | EACH | USP

AVISO

20/out não haverá aula (professor no International Symposium on Computer Architecture and High Performance Computing — SBAC-PAD)

ARMADILHA: O PANDEMÔNIO DE MOCKS

- 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

FIXTURE

- (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

```yaml
# 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
```

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

require 'rails_helper.rb'

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
```

FÁBRICAS

- 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)
- Gem: `FactoryGirl`

EXEMPLO: GEMA FACTORYGIRL

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

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

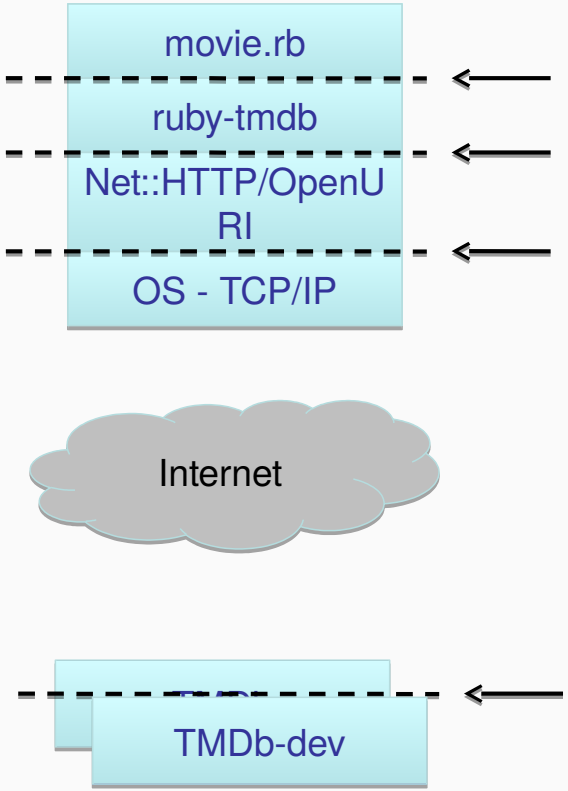
```ruby
# in spec/models/movie_spec.rb
describe Movie do
  it 'should include rating and year in full name' do
    # 'build' creates but doesn't save object; 'create' also saves it
    movie = FactoryGirl.build(:movie, :title => 'Milk', :rating => 'R')
    expect(movie.name_with_rating).to eq 'Milk (R)'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

TDD para o modelo & Stubs para a Internet

REQUISITOS EXPLÍCITOS VS. IMPLÍCITOS

- `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 [1]

[1] Veja o artigo How Amazon Web Services Uses Formal Methods em `http://dx.doi.org/10.1145/2699417`

MEDINDO A COBERTURA

```ruby
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

LEVANDO AO PÉ DA LETRA

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

LEVANDO AO PÉ DA LETRA

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

LEVANDO AO PÉ DA LETRA

- ✗ 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

LEVANDO AO PÉ DA LETRA

- ✗ 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

LEVANDO AO PÉ DA LETRA

- ✗ 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

LEVANDO AO PÉ DA LETRA

- ✗ 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

PERGUNTA

Qual afirmação é um mau conselho sobre TDD?

1. Crie mocks & stubs frequentemente e desde cedo nos testes de unidade
2. Tenha como objetivo ter uma grande cobertura de testes de unidade
3. Algumas vezes não há problemas em usar stubs & mocks em testes de integração
4. Testes de unidade dão mais confiança de que o sistema está correto do que testes de integração

Outros conceitos sobre testes; Testes vs. Depuração

INDEPENDÊNCIA NO RSPEC

- 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 **I**ndependentes
  - 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 `printf`s para imprimir variáveis a cada execução do programa	Teste pequenos pedaços de código usando expectativas
Pare o debugger 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

TESTES & TDD

☺

- 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

COBERTURA DE TESTES, QUALIDADE DE CÓDIGO

☺

- 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

SUMÁRIO SOBRE TDD

- 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

Testes de software na perspectiva Planeje-e-Documente

TESTES EM P-E-D?

- BDD/TDD escreve os testes antes do código
  - quando os desenvolvedores de P-e-D escrevem testes?
- BDD/TDD começa com histórias de usuário
  - por onde os desenvolvedores de P-e-D começam?
- BDD/TDD faz os desenvolvedores escrever código & teste
  - P-e-D usa pessoas diferentes para escrever teste e código?
- Qual a cara da documentação dos testes?

GERENTE DE PROJETO P-E-D

- P-e-D depende dos Gerentes de Projeto
- Documenta o plano de gerenciamento do projeto
- Cria o *Software Requirements Specification* (SRS)
  - pode ter centenas de páginas
  - padrão IEEE
- Precisa documentar o Plano de Testes
  - outro padrão IEEE

ABORDAGEM P-E-D PARA TESTES

- Gerente divide o SRS em unidades de programação
- Desenvolvedores escrevem o código das unidades
- Desenvolvedores fazer testes de unidade
- Uma equipe separada de *Quality Assurance* (QA) faz os testes de alto nível:
    - Módulo, integração, sistema, aceitação

3 opções de integração pelo QA

1. Integração top-down
   - começa no topo do grafo de dependência das unidades
   - funções de alto nível (UI) funcionam logo no começo
   - uso de muitos *stubs* para fazer o app “funcionar”
2. Integração bottom-up
   - começa na parte de baixo do grafo de dependências
   - não precisa de *stubs*, tudo é integrado em módulos
   - não dá para ver o app funcionando até que todo o código tenha sido escrito e integrado
3. Integração *sandwich*
   - melhor dos dois mundos?
   - reduz o uso de *stubs* ao integrar algumas unidades de forma bottom-up
   - tenta fazer a UI funcionar integrando algumas unidades top-down

EQUIPE DE TESTES DE QA

- A próxima equipe de QA faz o teste de sistema
  - o app completo deve funcionar
  - testes de requisitos não funcionais (desempenho) + requisitos funcionais (descritos no SRS)
- Quando o teste de sistema termina em P-e-D?
  - depende da política da organização:
    - ex: nível de cobertura de testes (todas as expressões)
    - ex: todas as entradas foram testadas com dados bons e dados ruins
- Etapa final: testes de aceitação do cliente ou usuário — validação vs. verificação

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

***Edsger W. Dijkstra**, Notes On Structured Programming*

Comece com uma especificação formal e prove que o comportamento do programa segue essa especificação:

1. Um humano escreve a prova
2. Um computador, usando provadores automáticos de teoremas:
   - usa inferência + axiomas de lógica para produzir provas a partir do zero
3. Um computador, usando verificação de modelos
   - verifica algumas propriedades selecionadas usando busca exaustiva em todos os estados possível que o sistema pode assumir durante a execução

MÉTODOS FORMAIS

- Computacionalmente caro, portanto use em:
  - algumas funções pequenas
  - casos onde arrumar é muito caro e testar é muito difícil
  - ex: protocolos de rede, SW crítico para segurança
- Maior projeto verificado: núcleo de um SO de 10k LOC ao custo de $ 500 / LOC
- Neste curso temos SW que muda com muita frequência (SaaS), fácil de consertar, fácil de testar $\Rightarrow$ não vamos aplicar métodos formais

Tarefas	*No Planeje e Documente*	*Em Métodos Ágeis*
Documentação e Plano de Testes	Documentação de Teste de Software, ex: norma IEEE 829-2008	Histórias de Usuários
Ordem de codificação e teste	1. Código 2. Teste Unitário 3. Teste Funcional 4. Teste de Integração 5. Teste de Sistema 6. Teste de Aceitação	1. Teste de Aceitação 2. Teste de Integração 3. Teste Funcional 4. Teste Unitário 5. Código
Testadores	Desenvolvedores para testes unitários; QA para teste funcional, integração, sistema e aceitação	Desenvolvedores
Quando Terminam os Testes	Política da empresa (ex: cobertura de expressões, entradas de caminhos tristes e felizes)	Todos os testes passarem (verde)