Testes de software na perspectiva Planeje-e-Documente

SIN5005 — Tópicos em Engenharia de Software

Daniel Cordeiro

16 de outubro de 2019

Escola de Artes, Ciências e Humanidades | EACH | USP

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

Limites de Testes

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

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

Métodos formais

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

Testes de software

Tarefas	*No Planeje e Documente*	*Em Métodos Ágeis*
Documentação e Plano de Testes	Documentação de Teste de Software, ex: norma ISO/IEC/IEEE 29119	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)

Manutenção de software

O que faz de um código ser “legado” e como método Ágil pode ajudar?

Código legado é importante

Já que manutenção de software consume 60% dos custos com o software, essa é provavelmente a fase mais importante do ciclo de vida do sofware...

> *"Old hardware becomes obsolete; old software goes into production every night."*
>
> Robert Glass, *Facts & Fallacies of Software Engineering* (fato #41)

O que podemos fazer
para entender e modificar (com segurança) um código legado?

Manutenção ≠ correção de bugs

- Melhorias: 60% do custo de manutenção
- Correção de bugs: 17% do custo de manutenção

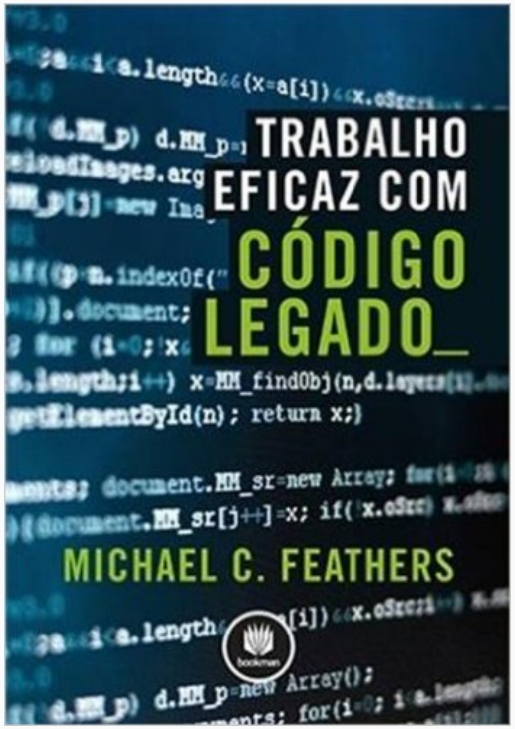
Daí a regra de “60/60”

- 60% do custo de software é manutenção
- 60% do custo de manutenção é melhoria

O que faz de um código ser “legado”?

Ele ainda faz o que o cliente precisa, mas além disso:

- você não o escreveu e ele está mal documentado
- ou você o escreveu, mas há muito, muito tempo atrás (e ele está mal documentado)
- ou ele não tem bons testes (independentemente de quem o escreveu) — Feathers, 2004



2 modos de encarar a modificação de código legado

2 modos de encarar a modificação de código legado

Edite & Reze
— “Tipo assim, eu meio que acho que eu provavelmente não quebrei nada.”

2 modos de encarar a modificação de código legado

Edite & Reze
— “Tipo assim, eu meio que acho que eu provavelmente não quebrei nada.”

Cubra & Modifique
— Faça com que a cobertura de testes seja seu cobertor de segurança!

Como métodos Ágeis podem ajudar?

1. Exploração: determine onde você precisa fazer as mudanças (pontos de mudanças)

Como métodos Ágeis podem ajudar?

1. Exploração: determine onde você precisa fazer as mudanças (pontos de mudanças)
2. Refatoração: o código em volta dos seus pontos de mudança são `(a)` testados? `(b)` testáveis?

Como métodos Ágeis podem ajudar?

1. Exploração: determine onde você precisa fazer as mudanças (pontos de mudanças)
2. Refatoração: o código em volta dos seus pontos de mudança são `(a)` testados? `(b)` testáveis?
    - se `(a)` é verdadeiro: “bora” mexer

Como métodos Ágeis podem ajudar?

1. Exploração: determine onde você precisa fazer as mudanças (pontos de mudanças)
2. Refatoração: o código em volta dos seus pontos de mudança são `(a)` testados? `(b)` testáveis?
   - se `(a)` é verdadeiro: “bora” mexer
   - `!(a) && (b)`: aplique ciclos de BDD+TDD para melhorar a cobertura do teste

Como métodos Ágeis podem ajudar?

1. Exploração: determine onde você precisa fazer as mudanças (pontos de mudanças)
2. Refatoração: o código em volta dos seus pontos de mudança são `(a)` testados? `(b)` testáveis?
   - se `(a)` é verdadeiro: “bora” mexer
   - `!(a) && (b)`: aplique ciclos de BDD+TDD para melhorar a cobertura do teste
   - `!(a) && !(b)`: **refatore**

Como métodos Ágeis podem ajudar?

1. Exploração: determine onde você precisa fazer as mudanças (pontos de mudanças)
2. Refatoração: o código em volta dos seus pontos de mudança são `(a)` testados? `(b)` testáveis?
   - se `(a)` é verdadeiro: “bora” mexer
   - `!(a) && (b)`: aplique ciclos de BDD+TDD para melhorar a cobertura do teste
   - `!(a) && !(b)`: **refatore**
3. Adicione testes para melhorar a cobertura conforme for preciso

Como métodos Ágeis podem ajudar?

1. Exploração: determine onde você precisa fazer as mudanças (pontos de mudanças)
2. Refatoração: o código em volta dos seus pontos de mudança são `(a)` testados? `(b)` testáveis?
   - se `(a)` é verdadeiro: “bora” mexer
   - `!(a) && (b)`: aplique ciclos de BDD+TDD para melhorar a cobertura do teste
   - `!(a) && !(b)`: **refatore**
3. Adicione testes para melhorar a cobertura conforme for preciso
4. Faça mudanças usando os testes como sua *referência base* (*ground truth*)

Como métodos Ágeis podem ajudar?

1. Exploração: determine onde você precisa fazer as mudanças (pontos de mudanças)
2. Refatoração: o código em volta dos seus pontos de mudança são `(a)` testados? `(b)` testáveis?
   - se `(a)` é verdadeiro: "bora" mexer
   - `!(a) && (b)`: aplique ciclos de BDD+TDD para melhorar a cobertura do teste
   - `!(a) && !(b)`: **refatore**
3. Adicione testes para melhorar a cobertura conforme for preciso
4. Faça mudanças usando os testes como sua *referência base* (*ground truth*)
5. Refatore ainda mais; deixe o código melhor do que você o encontrou

Isso sim é "abraçar as mudanças" em longo prazo

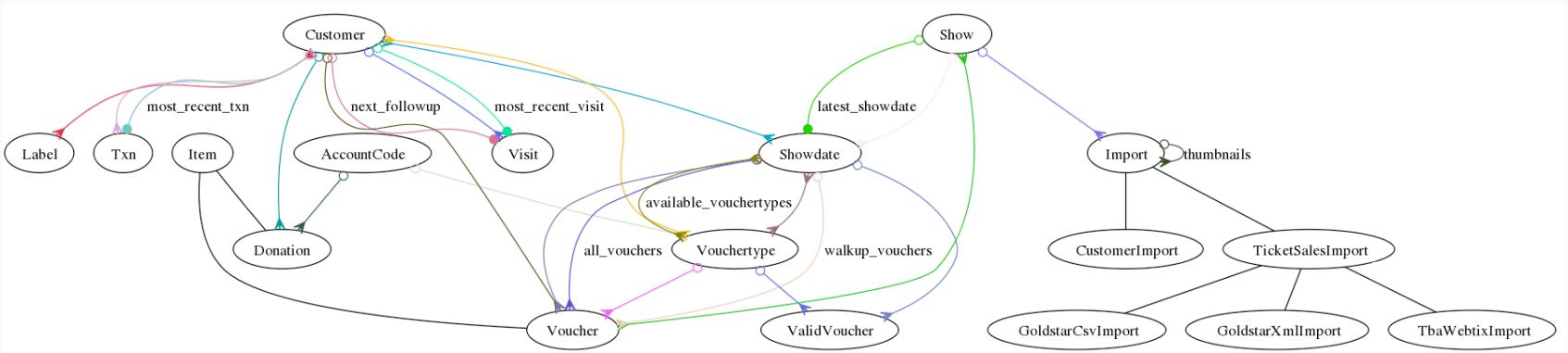
Abordagem e exploração de código legado

Faça o código rodar em desenvolvimento

- Faça o *check out* de um *branch* de rascunho (que não será enviado novamente pro repositório) e faça ele rodar:
  - com uma configuração parecida com a produção ou com o ambiente de desenvolvimento
  - idealmente com algo que se pareça com uma cópia do banco de dados de produção
- Aprenda algumas histórias de usuário: converse com o cliente e peça para ele explicar o que ele faz com o software

Entenda o esquema do banco de dados & as classes importantes

- Inspecione o esquema do banco de dados (`rake db:schema:dump`)
- Crie um diagrama de interação automaticamente (`gem install railroady`) ou inspecione o código manualmente
- Quais são as *classes* principais (as mais conectadas), suas *responsabilidades* e seus *colaboradores*?



Base de código & documentos “informais”

- Percepção geral do código
  - Qualidade de código subjetiva? (`rake metrics` depois de instalar a gema `metric-fu` ou usar o CodeClimate)
  - Razão código/teste? Tamanho do código? (`rake stats`)
  - Modelos/Visões/Controladores principais?
  - Testes Cucumber & RSpec
- Documentos informais do projeto
  - Esboços de interface lo-fi e histórias de usuário
  - E-mail arquivado, newsgroup, páginas do wiki interno, posts em blogs, etc. sobre o projeto
  - Anotações sobre a revisão do projeto (ex: Campfire ou Basecamp)
  - Logs do sistema de controle de versão, documentação RDoc

RDoc

```
##
# ClassModule is the base class for objects representing either a class or a
# module.

class RDoc::ClassModule < RDoc::Context
  ##
  # Constants that are aliases for this class or module

  attr_accessor :constant_aliases

  ##
  # Comment and the location it came from.  Use #add_comment to add comments

  attr_accessor :comment_location

  attr_accessor :diagram # :nodoc:

  ##
  # Class or module this constant is an alias for

  attr_accessor :is_alias_for

  ##
  # Return a RDoc::ClassModule of class +class_type+ that is a copy
  # of module +module+. Used to promote modules to classes.
  #--
  # TODO move to RDoc::NormalClass (I think)

  def self.from_module class_type, mod
    klass = class_type.new mod.name
  ...
```

Home
Pages Classes Methods
Search

Parent
RDoc::Context

Methods
::from_module
::new
#add_comment
#ancestors
#aref
#clear_comment
#complete
#description
#direct_ancestors
#document_self_or_methods
#documented?
#each_ancestor
#find_ancestor_local_symbol
#find_class_named
#full_name
#merge
#module?
#name=
#name_for_path
#non_aliases
#parse
#path
#remove_nodoc_children
#search_record
#store=
#superclass
#superclass=
#type
#update_aliases
#update_extends
#update_includes

class RDoc::ClassModule

ClassModule is the base class for objects representing either a class or a module.

Attributes

comment_location [RW]

Comment and the location it came from. Use add_comment to add comments

constant_aliases [RW]

Constants that are aliases for this class or module

is_alias_for [RW]

Class or module this constant is an alias for

Public Class Methods

from_module(class_type, mod)

Return a RDoc::ClassModule of class `class_type` that is a copy of module `module`. Used to promote modules to classes.

new(name, superclass = nil)

Creates a new ClassModule with `name` with optional `superclass`

This is a constructor for subclasses, and must never be called directly.

Calls superclass method RDoc::Context.new

Public Instance Methods

add_comment(comment, location)

Adds `comment` to this ClassModule's list of comments at `location`. This method is preferred over comment= since it allows ri data to be updated across multiple runs.

ancestors()

Ancestors list for this ClassModule: the list of included modules (classes will add their superclass if any).

Returns the included classes or modules, not the includes themselves. The returned values are either String or RDoc::NormalModule instances (see RDoc::Mixin#module)

Resumo: Exploração

- Avalie a base de código
- Identifique as classes e relações principais
- Identifique as estruturas de dados mais importantes
- Idealmente, identifique o(s) lugar(es) onde será necessário mudar o código
- Mantenha a documentação do projeto atualizada ao mudar o código:
  - diagramas
  - wiki do GitHub
  - comentários que você inserir usando o RDoc

Estabelecendo a referência base com Testes de Caracterização

Por quê?

- Você não quer escrever código sem testes
- Você não tem os testes
- Você não pode criar testes sem entender o código

Por onde começar?

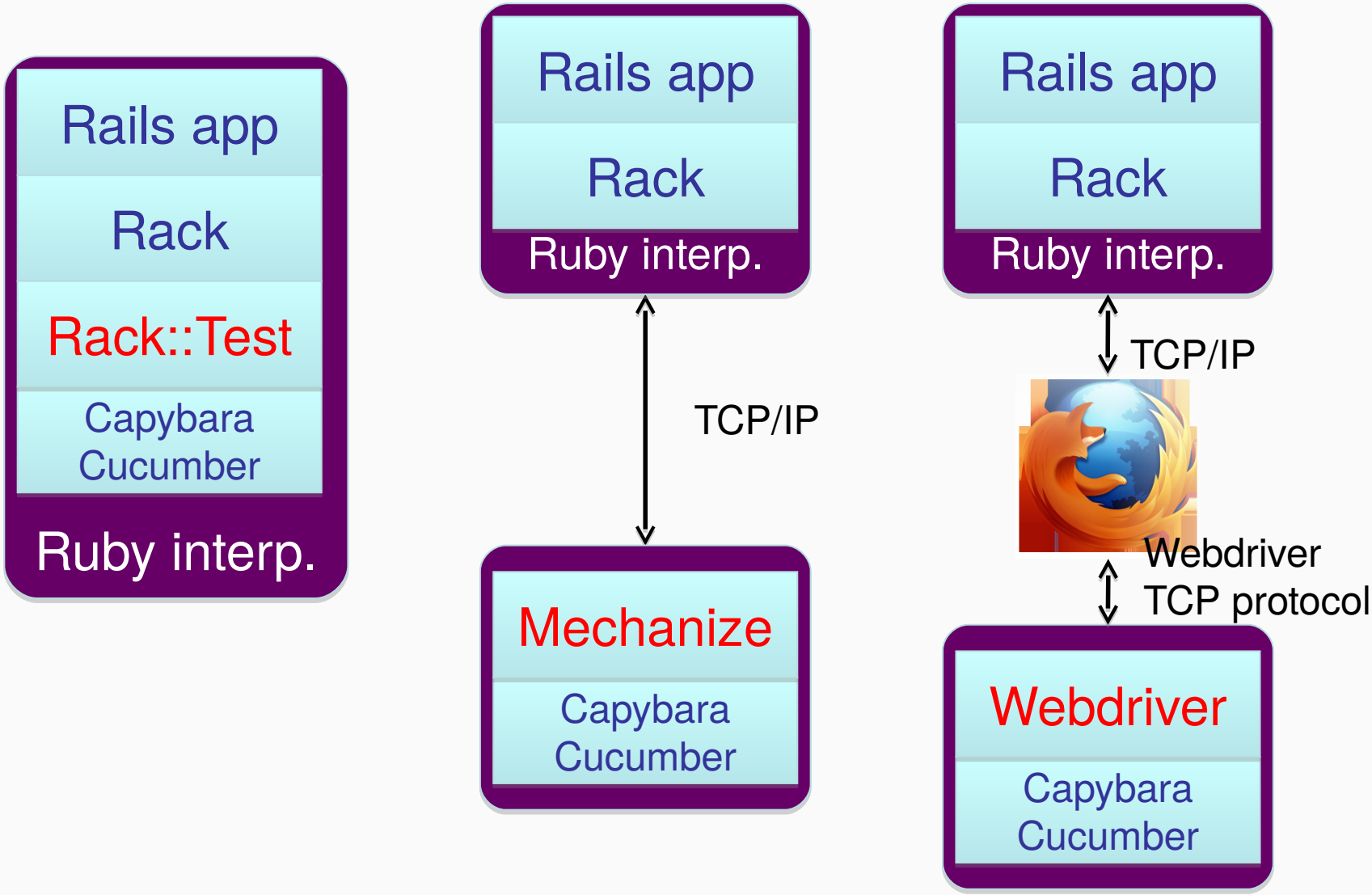
Testes de caracterização

- Estabeleça a *referência base sobre como o código funciona hoje*, como a base para a cobertura
  - Faça com que comportamentos conhecidos fiquem **R**epetíveis
  - Aumente a confiança de que você não está quebrando nada
- **Armadilha: não tente fazer melhorias nesse estágio!**

Testes de Caracterização no nível de Integração

- Primeiro passo natural: caixa-preta / nível de integração
  - não depende de entender a estrutura interna do app
- Use e abuse do Cucumber
  - *back-ends* do Capybara como o Mechanize faz com possamos automatizar quase tudo com um script
  - escreva os cenários imperativos agora
  - converta-os para declarativo ou melhore os passos `Given` depois, quando tiver um entendimento melhor de como funciona o app por dentro

In-process vs. out-of-process



Testes de Caracterização no nível de Unidade e Funcional

Escreva testes para ir aprendendo sobre o código
Veja o screencast em `http://vimeo.com/47043669`

```
class TimeSetter
  def self.convert(d)
    y = 1980
    while (d > 365) do
      if (y % 400 == 0 ||
          (y % 4 == 0 && y % 100 != 0))
        if (d > 366)
          d -= 366
          y += 1
        end
      else
        d -= 365
        y += 1
      end
    end
    return y
  end
end
```

Testes de Caracterização no nível de Unidade e Funcional

```
it "should calculate sales tax" do
  order = mock('order')
  expect(order.compute_tax).to eq -99.99
end
# object 'order' received unexpected message 'get_total'

it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  expect(order.compute_tax).to eq -99.99
end
# expected compute_tax to be -99.99, was 8.45

it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  expect(order.compute_tax).to eq 8.45
end
```

Testes de Caracterização no nível de Unidade e Funcional

```
it "should calculate sales tax" do
  order = mock('order')
  expect(order.compute_tax).to eq -99.99
end
# object 'order' received unexpected message 'get_total'
```

```
it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  expect(order.compute_tax).to eq -99.99
end
# expected compute_tax to be -99.99, was 8.45
```

```
it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  expect(order.compute_tax).to eq 8.45
end
```

Testes de Caracterização no nível de Unidade e Funcional

```
it "should calculate sales tax" do
  order = mock('order')
  expect(order.compute_tax).to eq -99.99
end
# object 'order' received unexpected message 'get_total'

it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  expect(order.compute_tax).to eq -99.99
end
# expected compute_tax to be -99.99, was 8.45

it "should calculate sales tax" do
  order = mock('order', :get_total => 100.00)
  expect(order.compute_tax).to eq 8.45
end
```

Manutenção de software: Comentários

Comentários devem descrever coisas que não são óbvias no código: “o porquê”, não “o quê”
— John Ousterhout

Comentários ruins

```
// Adiciona 1 a i
i++;


// Lock para proteger contra acesso concorrente
SpinLock mutex;


// Esta função troca os painéis
void swap_panels(Panel* p1, Panel* p2) {...}
```

Comentários

Comentários devem ser escritos em um nível de abstração maior do que o código

```
# Percorre o vetor para ver se o símbolo existe
```

E não...

```
# Em um laço para todo índice do array, pega
# o terceiro valor da lista do conteúdo para
# determinar se ela tem o símbolo que estamos
# procurando. Define o resultado como sendo
# o símbolo, se ele for encontrado.
```

Um exemplo extremo

```
def workaround_rails_bug_2298!
  # Rails Bug 2298: when a db txn fails, the id's of the instantiated objects
  # that were not saved are NOT reset to nil, which causes problems when
  # successfully saved later on (eg when transaction is rerun).  Also,
  #  new_record is not correctly reset to true.
  # the fix is based on a patch shown here:
  # http://s3.amazonaws.com/activereload-lighthouse/assets/fe67deaf98bb15d58218acdbbdf7d4f166
  # If any of the saves was on a record that had already been saved previously,
  #   we can just reload it instead; but we have to force trying this since we can't
  #   trust @new_record to tell us this fact.
  # If reload fails, and it really was a new record, we have to apply the fix.
```

Identificando o que está errado: Métricas, Cheiros de Código e SOFA

Além da corretude

Como dar feedback sobre a beleza do código?

- Existem diretrizes para o que é bonito?
- Avaliações qualitativas?
- Avaliações quantitativas?
- Se existem, funcionam?
- O Rails tem ferramentas para apoiar isso?

Qualitativo: Cheiros de Código

A sigla **SOFA** captura sintomas que normalmente indicam esses cheiros de código:

- O código é curto (**S**hort)?
- Faz uma única tarefa (**O**ne thing)
- Tem poucos argumentos (**F**ew arguments)
- Mantém um nível consistente de **A**bstração?

A ferramenta `reek` do Rails ajuda a encontrar cheiros de código.

Único nível de abstração

- Tarefas complexas precisam de uma estratégia dividir para conquistar
- Alerta amarelo para “encapsule essa tarefa em um método”
- Como em uma notícia de jornal, classes & métodos devem poder ser lidos de cima para baixo
  - Bom: comece com um resumo de alto nível sobre os pontos chave, depois discuta cada ponto em detalhe
  - Bom: cada parágrafo descreve um tópico
  - Ruim: vagueie sobre o código, mude os “níveis de abstração” toda hora ao invés de refiná-los progressivamente

Por que ter muitos argumentos é ruim?

- Difícil de conseguir uma boa cobertura de testes
- Difícil de criar mocks/stubs enquanto testa
- Argumentos booleanos devem acender a luz amarela:
  - se uma função se comporta de forma diferente baseado no valor de um argumento booleano, talvez você devesse ter duas funções
- Se argumentos “andam sempre em bando”, talvez você devesse extraí-los para uma nova classe
  - mesmo conjunto de argumentos para um monte de métodos

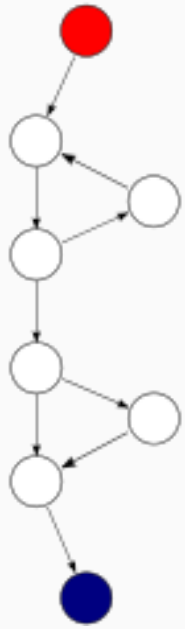
Quantitativo: Complexidade ABC

- Conta o número de **A**tribuições, ramos (***B****ranches*) e **C**ondições
- Pontuação = $\sqrt{A^2 + B^2 + C^2}$
- NIST (Natl. Inst. Stds. & Tech.) recomenda ≤ 20 / método
- A ferramenta `flog`, pro Rails, verifica a complexidade ABC

Quantitativo: complexidade ciclomática

- Número de caminhos linearmente independentes do código = $E - N + 2P$, onde E são as arestas, N são os nós e P são os componentes conexos do digrafo

```
def meu_metodo
  while(...)
    ....
  end
  if (...)
    faca_algo
  end
end
```



- No exemplo, E=9, N=8, P=1 $\Rightarrow$ CC = 3
- NIST recomenda que seja ≤ 10 / módulo
- Calculado pela ferramenta `saikuro`, para Rails

Quantitativo: Métricas

Métrica	Ferramenta	Meta de Pontuação
Razão código/testes	`rake stats`	$\leq 1:2$
C0 (expressão) cobertura	`SimpleCov`	$\geq 90\%$
Pontuação ABC	`flog`	< 20 por método
Complexidade ciclomática	`saikuro`	< 10 por método

- "Hotspots": lugares onde múltiplas métricas fizeram a luz vermelha acender
  - adicione `require 'metric_fu'` ao `Rakefile`
  - `rake metrics:all`
- Não leve as métricas ao pé da letra:
  - assim como com cobertura, elas são melhores para identificar onde melhorias são necessárias do que para garantir algo

Melhorando o estilo na base da palmatória

Pontuação do `flog`: 18,8

```ruby
def combine_anagrams(words)
   output_array = Array.new(0)
   words.each do |w1|
      temp_array = []
      words.each do |w2|
         if (w2.downcase.split(//).sort
             == w1.downcase.split(//).sort)
             temp_array.push(w2)
         end
      end
      output_array.push(temp_array)
   end
   return output_array.uniq
end
```

Pontuação do `flog`: 5.2

```ruby
def combine_anagrams(words)
   words.group_by { |word|
       word.chars.sort }.values
end
```

Nota: `flog` significa punir severamente com uma vara ou chicote. Descrição do seu site: "*Flog shows you the most torturous code you wrote. The more painful the code, the higher the score*".

Refatoração no nível de Métodos

Quantitativo: Métricas

Métrica	Ferramenta	Meta de Pontuação
Razão código/testes	`rake stats`	$\leq 1:2$
C0 (expressão) cobertura	`SimpleCov`	$\geq 90\%$
Pontuação ABC	`flog`	< 20 por método
Complexidade ciclomática	`saikuro`	< 10 por método

- "Hotspots": lugares onde múltiplas métricas fizeram a luz vermelha acender
  - adicione `require 'metric_fu'` ao `Rakefile`
  - `rake metrics:all`
- Não leve as métricas ao pé da letra:
  - assim como com cobertura, elas são melhores para identificar onde melhorias são necessárias do que para garantir algo

Qualitativo: Cheiros de Código

A sigla **SOFA** captura sintomas que normalmente indicam esses cheiros de código:

- O código é curto (**S**hort)?
- Faz uma única tarefa (**O**ne thing)
- Tem poucos argumentos (**F**ew arguments)
- Mantém um nível consistente de **A**bstração?

O CodeClimate usa métricas tanto qualitativas como quantitativas.

Exemplo: encorajar o cliente a “opt-in”

```ruby
# Objetivo: quando um cliente se logar pela primeira vez, verificar se
# ele optou por não receber e-mails. Se optou, mostrar uma mensagem
# encorajando ele a mudar de ideia.
# self.current_user devolve o usuário atualmente logged-in
# (uma instância de modelo ActiveRecord)

# em CustomersController

def show
  if self.current_user.e_blacklist?  &&
      self.current_user.valid_email_address?  &&
      !(m = Option.value(:encourage_email_opt_in)).blank?
    m << 'Clique na aba Endereço de Cobrança para atualizar suas preferências.'
    flash[:notice] ||= m
  end
end
```

- mistura diferentes níveis de abstração
- expõe detalhes de implementação de como calcular se o cliente precisa ver a mensagem
- como saber o que há em `flash[:notice]`? Se ele não for `nil`, isso nunca fará nada (mas a gente precisa saber disso)
- o que a gente realmente quer é que isso apareça uma vez por login

Exemplo: encorajar o cliente a “opt-in”

```ruby
# em ApplicationController

def login_message
  encourage_opt_in_message if self.current_user.has_opted_out_of_email?
end
#
# ....
#
def encourage_opt_in_message
  m = Option.value(:encourage_email_opt_in)
  m << 'Clique na aba Endereço de Cobrança para atualizar suas preferências.'
       unless m.blank?
  return m
end


# em customer.rb

def has_opted_out_of_email?
  e_blacklist? && valid_email_address?
end

# na ação de gestão de Login

flash[:notice] = login_message || "Usuário autenticado com sucesso"
```

Refatoração: ideia

- Comece com o código que tem 1 ou mais problemas / mau cheiros
- Usando uma série de pequenos passos, mude o código para o mau cheiro sumir
- Proteja cada passo com testes
- Minimize o tempo durante o qual os testes ficam vermelhos

História e contexto

- Fowler et al. desenvolveram o catálogo definitivo de refatorações
  - adaptado para várias linguagens
  - refatoração em nível de método e classe
- Cada refatoração consiste de:
  - Nome
  - Resumo do que ele faz / quando usar
  - Motivação (qual problema ele resolve)
  - Mecânica: receita passo a passo
  - Exemplo(s)


zune

Refatorando o TimeSetter

```
class TimeSetter

  def convert(d)
    y = 1980
    while (d > 365) do
      if (y % 400 == 0 ||
          (y % 4 == 0 && y % 100 == 0))
        if (d > 366)
          d -= 366
          y += 1
        end
      else
        d -= 365
        y += 1
      end
    end
    return y
  end

end
```

Refatorando o TimeSetter

```
# Refatoração aplicada: Renomeação de Variável

class DateCalculator

  def convert(days)
    year = 1980
    while (days > 365) do
      if (year % 400 == 0 ||
          (year % 4 == 0 && year % 100 == 0))
        if (days > 366)
          days -= 366
          year += 1
        end
      else
        days -= 365
        year += 1
      end
    end
    return year
  end

end
```

Refatorando o TimeSetter

```ruby
# Refatoração aplicada: Extração de Método

class DateCalculator

  def convert(days)
    year = 1980
    while (days > 365) do
      if leap_year?(year)
        if (days > 366)
          days -= 366
          year += 1
        end
      else
        days -= 365
        year += 1
      end
    end
    return year
  end

  # método extraído
  def leap_year?(year)
    (year % 400 == 0 ||
      (year % 4 == 0 && year % 100 != 0))
  end

end

describe DateCalculator do
  describe 'leap years' do
    before(:each) do ; @calc = DateCalculator.new ; end
    it 'should occur every 4 years' do
      @calc.leap_year?(2004).should be_true
    end
    it 'but not every 100th year' do
      @calc.leap_year?(1900).should_not be_true
    end
    it 'but YES every 400th year' do
      @calc.leap_year?(2000).should be_true
    end
  end
end
```

Refatorando o TimeSetter

```ruby
# Refatoração aplicada: decomposição de condicional
class DateCalculator
  attr_accessor :days, :year
  def initialize(days)
    @days = days
    @year = 1980
  end
  def convert
    while (@days > 365) do
      if leap_year?
        add_leap_year
      else
        add_regular_year
      end
    end
    return @year
  end
  # métodos extraídos
  def leap_year?
    (@year % 400 == 0 ||
      (@year % 4 == 0 && @year % 100 != 0))
  end
  def add_leap_year
    if (@days > 366)
      @days -= 366
      @year += 1
    end
  end
  def add_regular_year
    @days -= 365
    @year += 1
  end
end

describe DateCalculator do
  describe 'leap years' do
    before(:each) do
      @calc = DateCalculator.new(0)
    end
    def test_leap_year(year)
      @calc.year = year
      @calc.leap_year?
    end
    it 'should occur every 4 years' do
      test_leap_year(2004).should be_true
    end
    it 'but not every 100th year' do
      test_leap_year(1900).should_not be_true
    end
    it 'but YES every 400th year' do
      test_leap_year(2000).should be_true
    end
  end
end
```

Refatorando o TimeSetter

```ruby
class DateCalculator
  attr_accessor :days, :year
  def initialize(days)
    @days = days
    @year = 1980
  end

  def convert
    while (@days > 365) do
      if leap_year?
        add_leap_year
      else
        add_regular_year
      end
    end
    return @year
  end

  # extracted methods
  def leap_year? ... end
  def add_leap_year ... end
  def add_regular_year ... end
end

describe DateCalculator do
  describe 'leap years' do
    before(:each) do
      @calc = DateCalculator.new(0)
    end
    def test_leap_year(year)
      @calc.year = year
      @calc.leap_year?
    end
```

```ruby
    it 'should occur every 4 years' do
      test_leap_year(2004).should be_true
    end
    it 'but not every 100th year' do
      test_leap_year(1900).should_not be_true
    end
    it 'but YES every 400th year' do
      test_leap_year(2000).should be_true
    end
  end

  describe 'adding a leap year' do
    it 'shouldnt peel off leap year if not enough days left' do
      @calc = DateCalculator.new(225)
      @calc.year = 2008
      expect { @calc.add_leap_year }.not_to change { @calc.year }
    end
    it 'should peel off leap year if >1 year of days left' do
      @calc = DateCalculator.new(400)
      @calc.year = 2008
      expect { @calc.add_leap_year }.to change { @calc.year }.by(1)
    end
    it 'should peel off leap year if exactly 1 year of days left' do
      @calc = DateCalculator.new(366)
      @calc.year = 2008
      # will fail given original code!
      expect { @calc.add_leap_year }.to change { @calc.year }.by(1)
    end
  end

end
```

Refatorando o TimeSetter

- Corrija nomes ruins
- Extrair método
- Extrair método, encapsular classe
- Teste os métodos extraídos
- Sobre testes de unidade:
  - teste caixa branca pode ser útil quando refatorar
  - abordagem clássica: teste os valores críticos e alguns valores não críticos que sejam representativos

Resumo do que foi feito

- O calculador de datas ficou mais fácil de ler e entender usando refatorações simples
- Encontramos um erro
- Observação: se o método fosse desenvolvido com TDD, provavelmente teria sido mais fácil
- Melhoramos a pontuação do `flog` e `reek`

Outros mau cheiros & Remédios

Mal cheiro	Refatoração que pode resolver
Classe grande	Extrair classe, subclasse ou módulo
Método longo	Decompor condicional Substituir laço por método de coleção Extrair método Extrair método externo com yield() Substituir variável temporária por consulta Substituir método por objeto método
Lista de parâmetros longa	Substituir parâmetro por método Extrair classe
Intimidade inapropriada e *shotgun surgery* Comentários em excesso	Mover método/campo cara recuperar itens relacionados em um único (DRY) lugar Extrair método Introduzir asserção Substituir por comentários
Níveis inconsistentes de abstração	Extrair métodos & classes

Pergunta

Qual item abaixo não é um objetivo de refatoração em nível de método?

1. Reduzir a complexidade do código
2. Eliminar mau cheiros de código
3. Eliminar bugs
4. Melhorar a testabilidade