# MANUTENÇÃO DE SOFTWARE

## ACH2006 – ENGENHARIA DE SISTEMAS DE INFORMAÇÃO
## SIN5005 – TÓPICOS EM ENGENHARIA DE SOFTWARE

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

# 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 (ex: 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<br>2. Teste Unitário<br>3. Teste Funcional<br>4. Teste de Integração<br>5. Teste de Sistema<br>6. Teste de Aceitação | 1. Teste de Aceitação<br>2. Teste de Integração<br>3. Teste Funcional<br>4. Teste Unitário<br>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) |

# 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

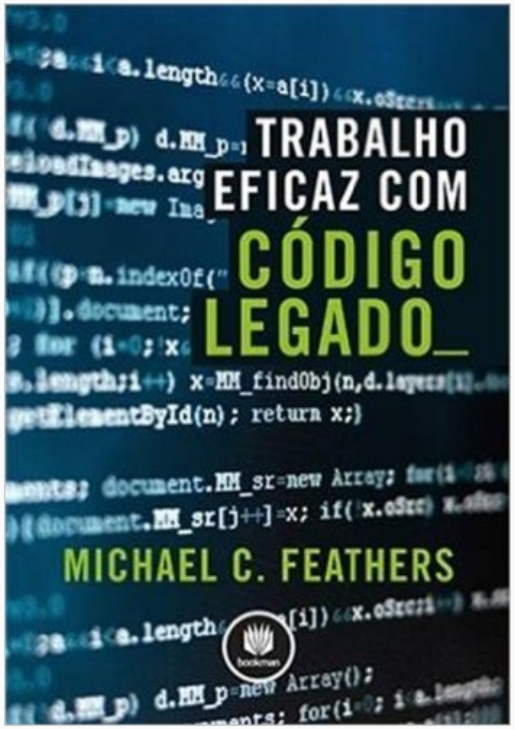
**Daí a regra de “60/60”**

- 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**

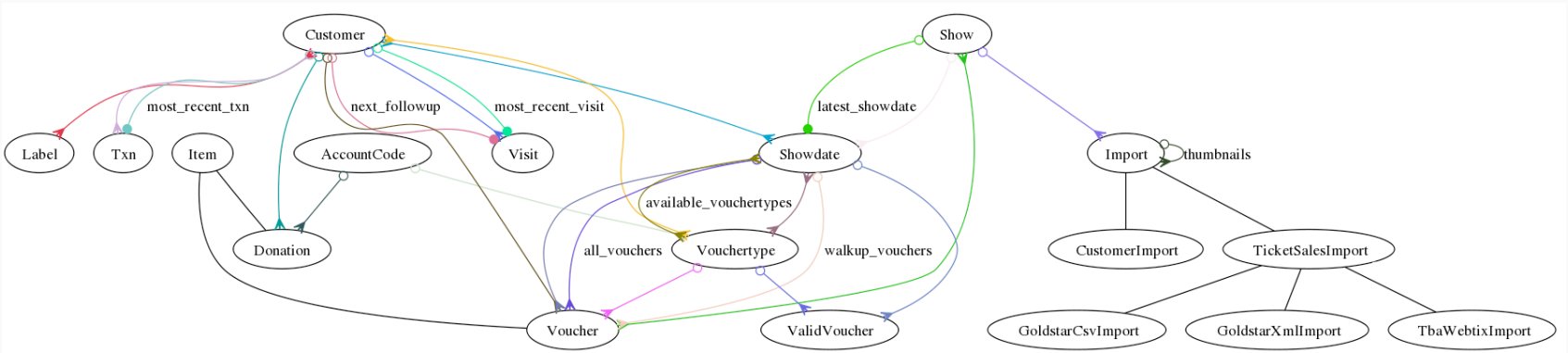
# Abordagem e exploração de código legado

# 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)

*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 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.
```

# 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?**

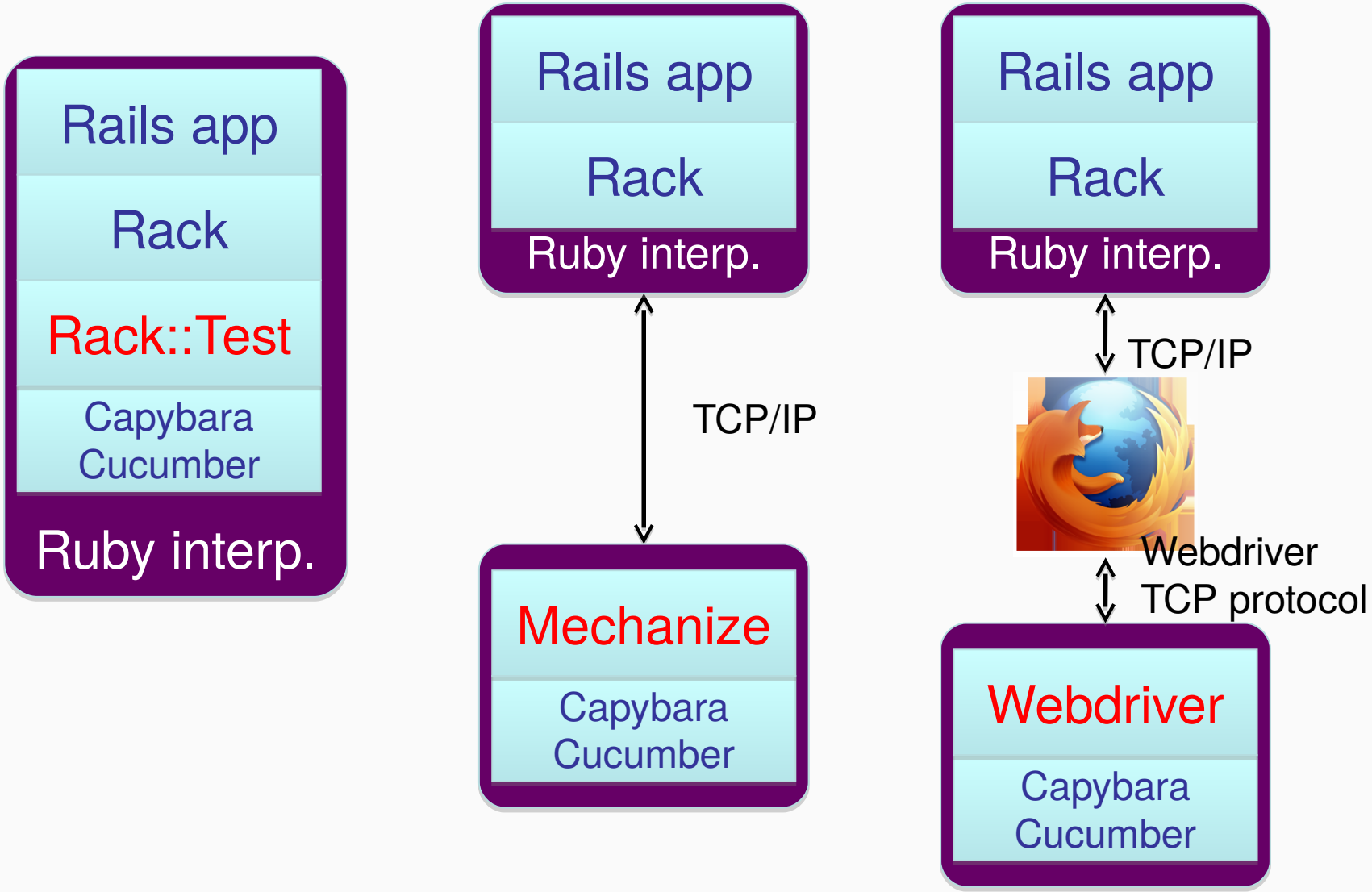
# TESTES DE CARACTERIZAÇÃO

- 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

```ruby
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

```ruby
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

```ruby
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
```

Escreva testes para ir aprendendo sobre o código
Veja o screencast em https://youtu.be/8QwvqtMp5QM

```
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
```