MANUTENÇÃO DE SOFTWARE: MÉTRICAS, CHEIROS DE CÓDIGO, SOFA E REFATORAÇÃO

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

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

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

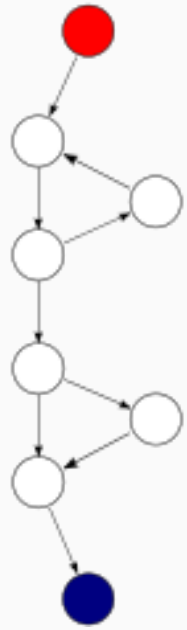
QUANTITATIVO: COMPLEXIDADE ABC

- 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*".

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 decidir se o cliente precisa ver a mensagem (i.e., o que o "opted out" significa)
- como saber o que há em `flash[:notice]`? Se ele não for `nil`, isso nunca fará nada (mas precisamos 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"
```

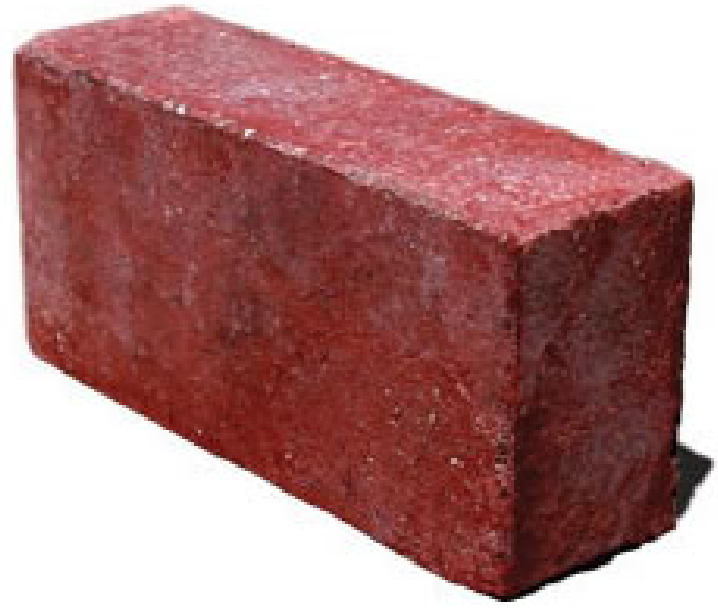
Fonte: https://pastebin.com/AFQAKxbR

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

- 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
      # este teste falharia com o código original!
      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

Mau 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 para recuperar itens relacionados em um único (DRY) lugar Extrair método Introduzir asserção Substituir por comentários
Níveis inconsistentes de abstração	Extrair métodos & classes

Perspectiva Planeje-e-Documente na Manutenção de Software

MANUTENÇÃO P-E-D?

- Quanto é gasto em desenvolvimento P-e-D em relação à manutenção P-e-D?
  - quanto é isso comparado com um método Ágil?
- Desenvolvedores ágeis mantém o código
  - P-e-D usa as mesmas pessoas ou usa gente diferente para a manutenção?
- Qual a cara da documentação de manutenção de P-e-D?

MANUTENÇÃO P-E-D

- P-e-D gasta 1/3 em desenvolvimento, 2/3 em manutenção
  - clientes gastam 10% / ano em taxas de manutenção de SW
- Equipe de Desenvolvimento ≠ Equipe de Manutenção
  - Gerentes de manutenção
  - Engenheiro de manutenção de software
  - (em geral, são menos prestigiados)

GERENTE DE MANUTENÇÃO

Tal como um gerente de desenvolvimento:

- estima riscos, mantém cronograma, avalia riscos e os supera
- recruta a equipe de manutenção
- avalia o desempenho dos engenheiros de software (o que define seus salários)
- Documenta o plano de manutenção do projeto (mantém os documentos e código)
  - padronizado pela IEEE
- Culpado se o upgrade demora muito tempo ou se torna muito caro

PROCESSO DE MANUTENÇÃO P-E-D

Diferenças em relação ao processo de desenvolvimento:

1. Software funcionando em produção
   - novos lançamentos não podem quebrar as funcionalidades
2. Colaboração com o cliente
   - trabalha com o cliente para melhorar o próximo lançamento (vs. respeitar a especificação do contrato)
3. Respostas às mudanças
   - clientes enviam requisições de mudanças, os quais os engenheiros de software devem priorizar
   - formulários de requisições de mudanças são rastreados com tickets

COMITÊ DE CONTROLE DE MUDANÇAS

- Comitê (não o gerente) decide
- Gerente estima custo/tempo por pedido de mudança
- Equipe de QA estima o custo de testar a mudança, incluindo testes de regressão + novos testes
- Equipe de documentação estima os custos de atualização dos documentos
- Grupo de atendimento ao cliente decide se é urgente ou *workaround*

REQUISIÇÃO URGENTE DE MUDANÇA

- Quando não há tempo para atualizar docs, planos e código
  - o software falha (e morre)
  - novas leis em vigor afetam o produto
  - buraco na segurança $\Rightarrow$ dados vulneráveis
  - novos lançamentos de S.O. ou bibliotecas necessárias
  - precisa bater a nova funcionalidade do concorrente
- Sincronizar depois da emergência?
  - as emergências podem ser muito frequentes para dar tempo de sincronizar
- Tempo para refatorar código e melhorar a manutenibilidade
  - pode ser considerado muito caro pelo comitê de controle de mudanças

HORA DE REPROJETAR?

- Hora de refatorar para melhorar a manutenibilidade?
  - refatoração continua durante o desenvolvimento & manutenção
- Reprojetar para melhorar vs. Substituir? Use ferramentas automáticas para atualizar a medida que o SW envelhecer (e a manutenção ficar mais difícil)
  - mude o schema do banco de dados
  - melhore a documentação fazendo engenharia reversa
  - ferramentas de análise de código para apontar código ruim
  - ferramentas de tradução de linguagem de programação

MANUTENÇÃO: P-E-D VS. ÁGIL

Tarefas	*Em Planeje-e-Documente*	*Em Ágil*
Requisições de mudanças do cliente	Formulário de requisição de mudança	História de usuário em cartões A5 no formato Connextra
Estimativa de custo/tempo da mudança	Pelo Gerente de Manutenção	Pontos, pela Equipe de Desenvolvimento
Priorização das mudanças	Comitê de Controle de Mudanças	Time de Desenvolvimento com a participação do cliente
Papéis	Gerente de manutenção	∅
	Eng. de software de manutenção Time de QA Time de documentação Grupo de atendimento ao cliente	Time de Desenvolvimento

FALÁCIAS E ARMADILHAS

"A maior parte do tempo gasto com o projeto, codificação e teste será usada com refatorações."

Falácia Será mais rápido jogar esse código fora e começar tudo de novo

Armadilha Misturar refatoração com implementação de melhorias

Armadilha Aderência rígida às métricas ou "alergia" a cheiros de código

Armadilha *Dívida Técnica*: esperar muito tempo para fazer uma "grande refatoração" (vs. refatoração contínua)

PROVOCAÇÃO

Se 2/3 do custo de um produto são relacionados à fase de manutenção, por que não usar um mesmo processo de desenvolvimento (que seja compatível com a manutenção) em todo o ciclo de desenvolvimento (Ágil) ao invés de usar um processo (e uma equipe) separado para desenvolvimento e um outro para manutenção?