

PADRÕES, ANTIPADRÕES E SOLID

ENGENHARIA DE SISTEMAS DE INFORMAÇÃO

Daniel Cordeiro

24 de novembro de 2017

Escola de Artes, Ciências e Humanidades | EACH | USP

Motivação¹: minimizar o custo de mudanças

- Single Responsibility principle
- Open/Closed principle
- Liskov substitution principle
- Injection of dependencies
 - também chamado de Interface Segregation principle
- Demeter principle

¹Propostos por Robert C. Martin, coautor do Manifesto Ágil

Motivação¹: minimizar o custo de mudanças

- Princípio da Responsabilidade Única
- Princípio Aberto/Fechado
- Princípio da Substituição de Liskov
- Princípio da Injeção de Dependência
 - também chamado de Princípio da Segregação de Interface
- Princípio de Demeter

¹Propostos por Robert C. Martin, coautor do Manifesto Ágil

- Uma classe deve ter *uma e apenas uma* razão para mudar
 - cada *responsabilidade* é um *eixo de mudança* possível
 - mudanças em um eixo não deve afetar os outros
- Qual a responsabilidade desta classe, em ≤ 25 palavras?
 - parte de modelar um projeto OO é definir as responsabilidades e depois segui-las à risca
- Modelos com muitos conjuntos de comportamentos
 - ex: um usuário é um espectador de filmes e um membro de rede social e um usuário a ser autenticado, ...
 - classes muito grandes são uma dica de que algo está errado

PRINCÍPIO ABERTO/FECHADO

- Classes devem ser *abertas para extensão*, mas fechadas para modificação no código

```
class Report
  def output_report
    case @format
    when :html
      HtmlFormatter.new(self).output
    when :pdf
      PdfFormatter.new(self).output
```

- Classes devem ser *abertas para extensão*, mas fechadas para modificação no código

```
class Report
  def output_report
    case @format
    when :html
      HtmlFormatter.new(self).output
    when :pdf
      PdfFormatter.new(self).output
```

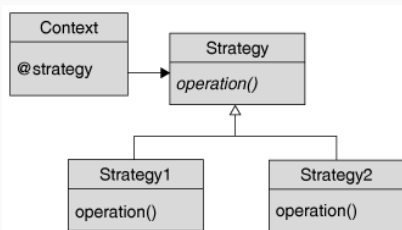
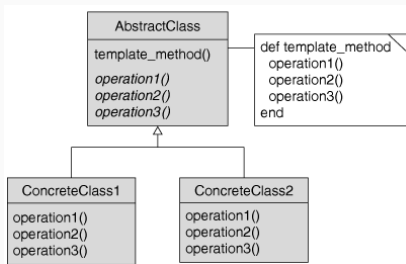
- Não é possível estender (adicionar novos tipos de relatórios) sem mudar a classe base **Report**
 - não é tão ruim quanto em linguagens estaticamente tipadas... mas é feio

- Como evitar uma violação do princípio aberto/fechado no construtor de **Report**, se o tipo da saída não é conhecido até o momento da execução?
- Em linguagens estaticamente tipadas: padrão *abstract factory*
- Ruby tem uma implementação particularmente simples para esse padrão

Veja: <http://pastebin.com/p3AHMqHZ>

PADRÃO TEMPLATE METHOD & STRATEGY

- *Template Method*: o **conjunto de passos** é o mesmo, mas a implementação dos passos é diferente
 - **herança**: subclasses sobreescrevem os métodos abstratos dos “passos”
- *Strategy*: a **tarefa** é a mesma, mas há muitas formas de fazê-la
 - **composição**: classes componentes implementam toda a tarefa

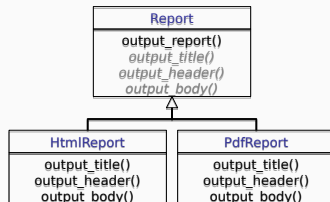


GERAÇÃO DO RELATÓRIOS USANDO TEMPLATE

```
class Report
  attr_accessor :title, :text
  def output_report
    output_title
    output_header
    output_body
  end
end
```

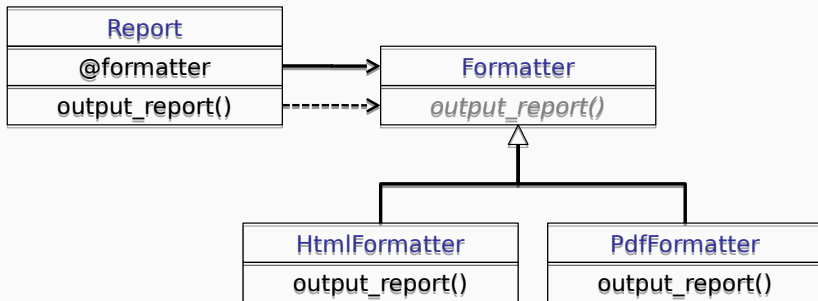
```
class HtmlReport < Report
  def output_title ... end
  def output_header ... end
end

class PdfReport < Report
  def output_title ... end
  def output_header ... end
end
```



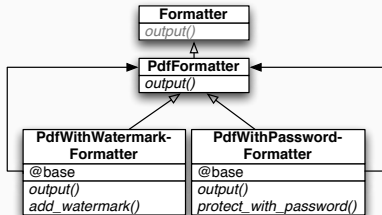
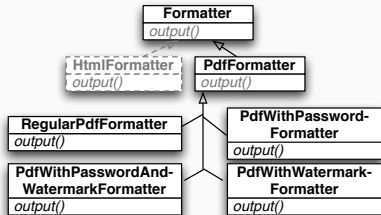
GERAÇÃO DE RELATÓRIO USANDO STRATEGY

```
class Report
  attr_accessor :title, :text, :formatter
  def output_report
    formatter.output_report
  end
end
```



Prefira composição à herança!

PADRÃO DECORATOR: TIRANDO AS REPETIÇÕES DOS PONTOS DE EXTENSÃO



Exemplo em Rails: escopos de ActiveRecord

```
Movie.for_kids.with_good_reviews(3)
```

```
Movie.with_many_fans.recently_reviewed
```

Outro exemplo de composição ao invés de herança!

- Você não pode se fechar contra *todos os tipos* de mudanças, então você tem que escolher (e você ainda assim pode estar errado)
- A metodologia Ágil pode ajudar a expôr os tipos de mudanças mais importantes o mais cedo possível
 - Projeto guiado por cenários com funcionalidades priorizadas
 - Iterações curtas
 - Desenvolvimento com testes primeiro
- Então você pode tentar fechar contra *esses tipos* de mudanças

O PRINCÍPIO DA SUBSTITUIÇÃO DE LISKOV

SUBSTITUIÇÃO DE LISKOV: SUBTIPOS PODEM SUBSTITUIR A CLASSE PAI

- Formulação atribuída a ganhadora do Prêmio Turing Barbara Liskov

“Um método que age sobre também poder agir sobre qu
- Tipo/subtipo ≠ classe/subclasse. Com tipagem “pato”, o conceito de *substitutividade* depende de como os colaboradores interagem com o objeto



EXEMPLO

```
class Rectangle
  attr_accessor :width, :height, :top_left_corner
  def new(width,height,top_left) ... ; end
  def area ... ; end
  def perimeter ... ; end
end
```

Quadrado é um caso especial de retângulo, certo?

EXEMPLO

```
class Rectangle
  attr_accessor :width, :height, :top_left_corner
  def new(width,height,top_left) ... ; end
  def area ... ; end
  def perimeter ... ; end
end
```

Quadrado é um caso especial de retângulo, certo?

```
class Square < Rectangle
  # mas... um quadrado precisa ter largura = altura
  attr_reader :width, :height, :side
  def width=(w) ; @width = @height = w ; end
  def height=(w) ; @width = @height = w ; end
  def side=(w) ; @width = @height = w ; end
end
```

EXEMPLO

Será que um quadrado realmente é um retângulo?

```
def make_twice_as_wide_as_high(r, dim)
```

```
    r.width = 2*dim
```

```
    r.height = dim
```

```
end
```

```
# viola PSL se esse método for parte do seu "contrato"!
```

EXEMPLO

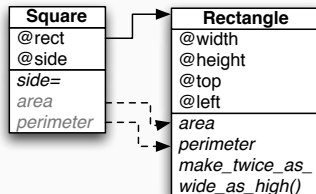
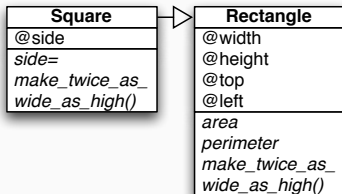
Será que um quadrado realmente é um retângulo?

```
def make_twice_as_wide_as_high(r, dim)
  r.width = 2*dim
  r.height = dim
end
# viola PSL se esse método for parte do seu "contrato"!
```

Solução compatível com o PSL: substituir herança com delegação. Tipagem pato do Ruby permite você usar quadrado na maior parte dos lugares onde retângulo seria usado, mas não como uma subclasse propriamente dita.

```
class Square
  def initialize(side, top_left_corner)
    @rect = Rectangle.new(side, side, top_left_corner)
  end
  def area      ; @rect.area      ; end
  def perimeter ; @rect.perimeter ; end
  def side=(s) ; @rect.width = @rect.height = s ; end
end
```

- Composição vs. (mau uso de) herança
- Se não puder expressar hipóteses consistentes sobre o “contrato” entre a classe e seus colaboradores, provavelmente é uma violação do PSL
 - sintoma: mudança na subclasse requer mudanças na superclasse (*shotgun surgery*)



- (a) Em linguagens com tipagem pato, violações ao PSL podem acontecer mesmo quando herança não é usada
 - (b) Em linguagens com tipagem estática, se o compilador não relatar nenhum erro/warning, então não há violações ao PSL
-
- 1. Apenas (a) é verdadeira
 - 2. Apenas (b) é verdadeira
 - 3. Tanto (a) quanto (b) são verdadeiras
 - 4. Nem (a) nem (b) são verdadeiras

PRINCÍPIO DE DEMETER

- Fale apenas com seus amigos... não fale com estranhos
- Você pode chamar os métodos:
 - que são seus
 - de suas variáveis de instância (se aplicável)
- Mas não nos resultados devolvidos por elas

Soluções:

- trocar método por delegação
- separar a computação transversal (padrão *Visitor*)
- estar ciente de eventos importantes sem conhecer seus detalhes de implementação (padrão *Observer*)

EXEMPLO

Imagine um sistema² onde um entregador de jornal cobra seus clientes, que guardam dinheiro em uma carteira

```
class Wallet
  attr_accessor :cash
end
class Customer
  has_one :wallet
end
class Paperboy
  def collect_money(customer, due_amount)
    if customer.wallet.cash < due_ammount
      raise InsufficientFundsError
    else
      customer.wallet.cash -= due_amount
      @collected_amount += due_amount
    end
  end
end
```

²Fonte: <http://www.dan-manges.com/blog/37>

EXEMPLO

Imagine um sistema² onde um entregador de jornal cobra seus clientes, que guardam dinheiro em uma carteira

```
class Wallet
  attr_accessor :cash
end
class Customer
  has_one :wallet
end
class Paperboy
  def collect_money(customer, due_amount)
    if customer.wallet.cash < due_ammount
      raise InsufficientFundsError
    else
      customer.wallet.cash -= due_amount
      @collected_amount += due_amount
    end
  end
end
```

- O entregador de jornal não deveria tirar o dinheiro diretamente da carteira do cliente!
- Quem deveria tratar o erro de fundos insuficientes? Paperboy ou Wallet?

²Fonte: <http://www.dan-manges.com/blog/37>

EXEMPLO

Um pouco melhor: nós **delegamos** o atributo `cash` via `Customer`.
Assim `Paperboy` só “fala com” `Customer`

```
class Customer
  def cash
    self.wallet.cash
  end
end

class Paperboy
  def collect_money(amount)
    if customer.cash >= amount
      customer.cash -= due_amount
      @collected_amount += due_amount
    else
      raise InsufficientFundsError
    end
  end
end
```

EXEMPLO

Essa solução é ainda melhor, agora o **comportamento** que é delegado. A implementação do comportamento pode ser mudada sem afetar **Paperboy**

```
class Wallet
  attr_reader :cash # não é mais um attr_accessor!
  def withdraw(amount)
    raise InsufficientFundsError if amount > cash
    cash -= amount
    amount
  end
end

class Customer
  # behavior delegation
  def pay(amount)
    wallet.withdraw(amount)
  end
end

class Paperboy
  def collect_money(customer, due_amount)
    @collected_amount += customer.pay(due_amount)
  end
end
```