PRINCÍPIOS SOLID

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

PRINCÍPIOS SOLID PARA POO

Motivação[1]: minimizar o custo de mudanças

- **S**ingle Responsibility principle
- **O**pen/Closed principle
- **L**iskov substitution principle
- **I**njection of dependencies
  - também chamado de Interface Segregation principle
- **D**emeter principle

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

Motivação[1]: 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

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

PRINCÍPIO DA RESPONSABILIDADE ÚNICA

- 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

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
```

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
```

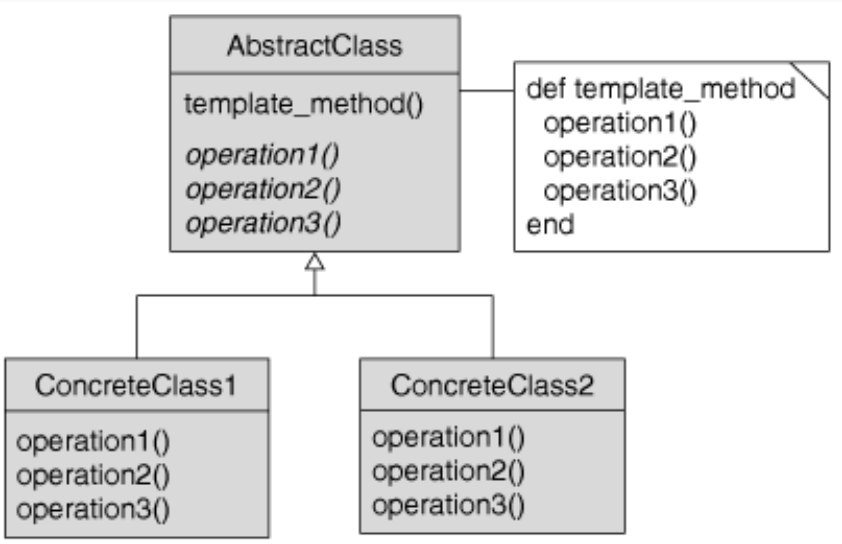
- Não é possível estender (adicionar novos tipos de relatórios) sem mudar a classe base `Report`
  - não é tão ruim quanto em linguagens estaticamente tipadas... mas é feio

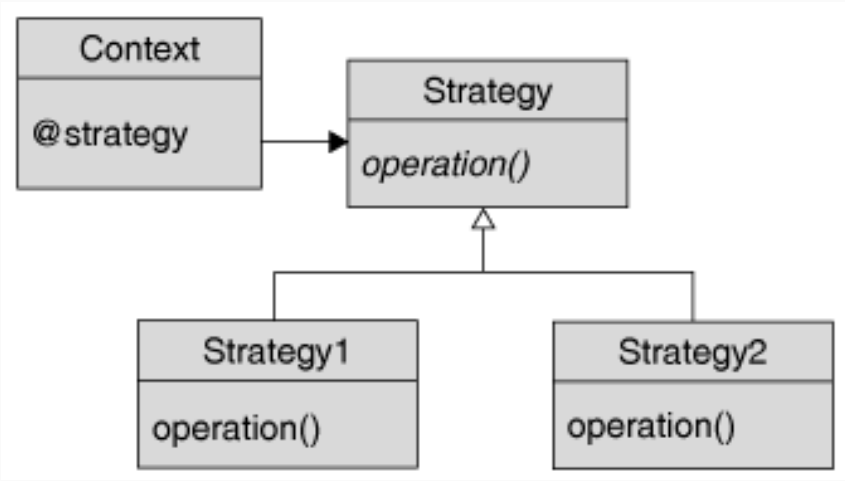
PADRÃO ABSTRACT FACTORY

- 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 (como Java): padrão *abstract factory*

PADRÃO TEMPLATE METHOD & STRATEGY

- *Template Method*: o conjunto de passos é o mesmo, mas a implementação dos passos é diferente
  - herança: subclasses sobrescrevem 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

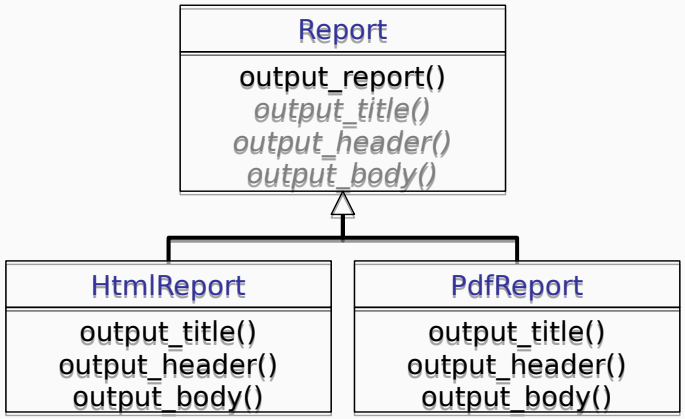




GERAÇÃO DE RELATÓRIOS USANDO TEMPLATE

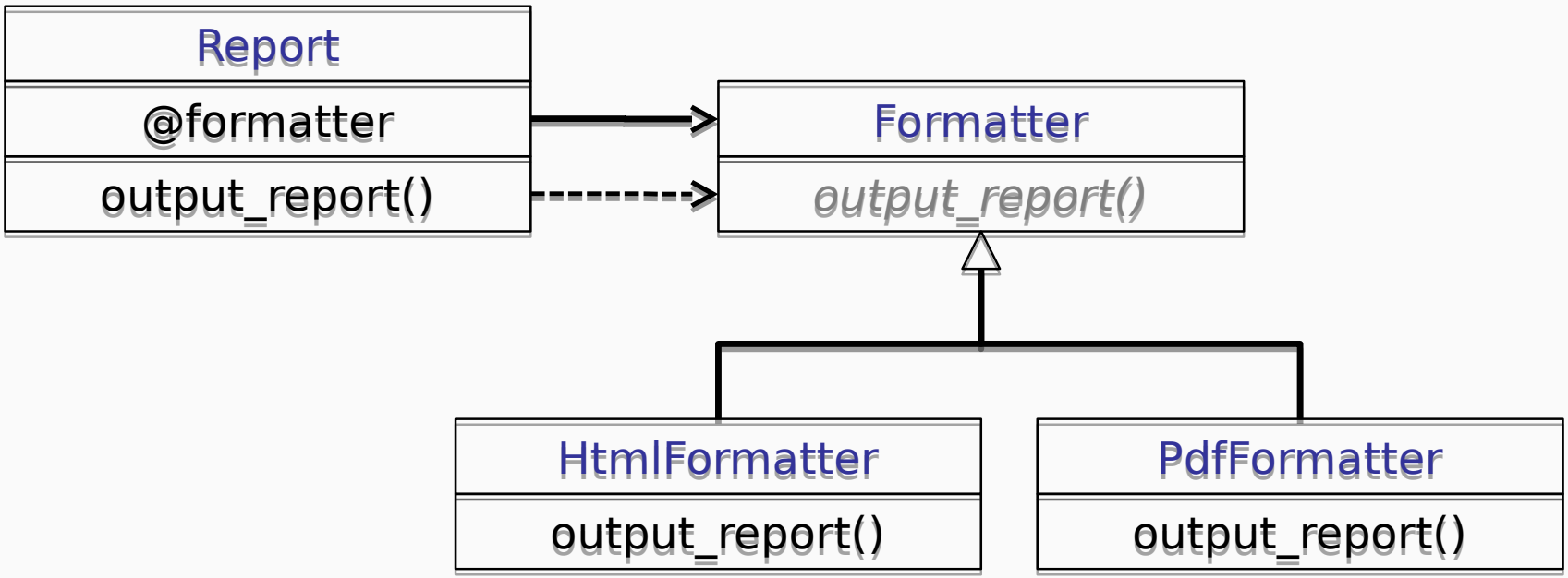
```ruby
class Report
 attr_accessor :title, :text
 def output_report
  output_title
  output_header
  output_body
 end
end

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



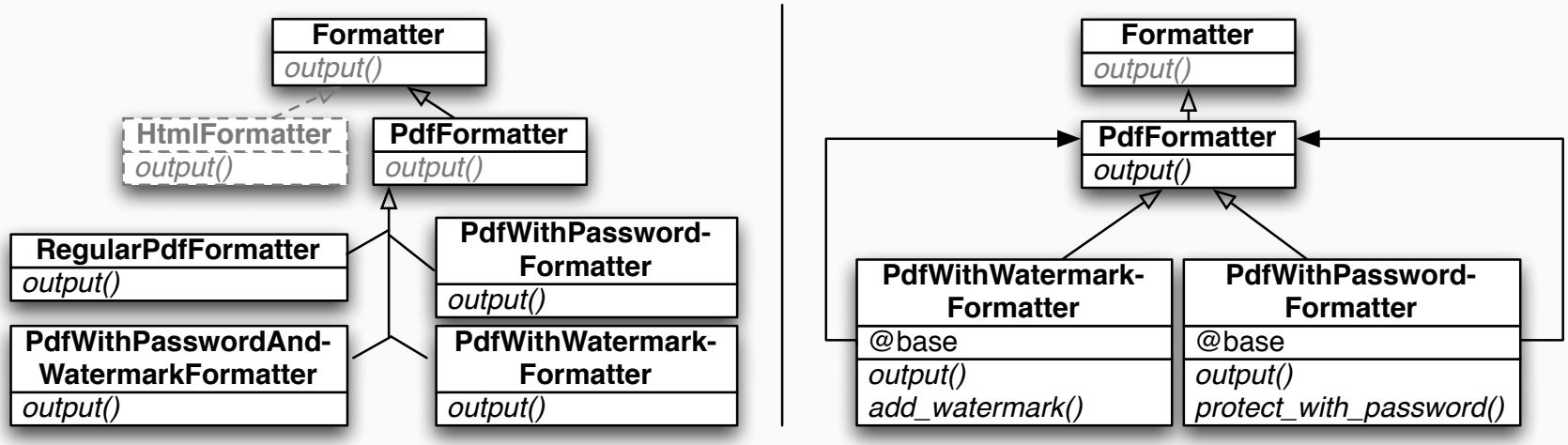
GERAÇÃO DE RELATÓRIOS USANDO STRATEGY

```
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



Outro exemplo de composição ao invés de herança!

PRINCÍPIO ABERTO/FECHADO NA PRÁTICA

- Você não pode se fechar contra *todos os tipos* de mudanças, então você tem que escolher (e você ainda assim pode estar errado)
- Evoluir o código aos poucos com *feedback* contínuo dos usuários (usando alguma metodologia Ágil[2]) pode ajudar a expôr os tipos de mudanças mais importantes o mais cedo possível. Mas, para isso, é preciso:
  - definir cenários de uso do programa, com funcionalidades priorizadas
  - iterações curtas
  - desenvolvimento com testes primeiro e muita refatoração
- Então você pode tentar se fechar contra *esses tipos* de mudanças

[2] Vocês verão mais sobre isso em ESI.

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 uma instância de tipo T deve também poder agir sobre qualquer subtipo de T"*

- Tipo/subtipo ≠ classe/subclasse. Com tipagem "pato", o conceito de *substitutividade* depende de como os colaboradores interagem com o objeto

EXEMPLO

```ruby
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

```ruby
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

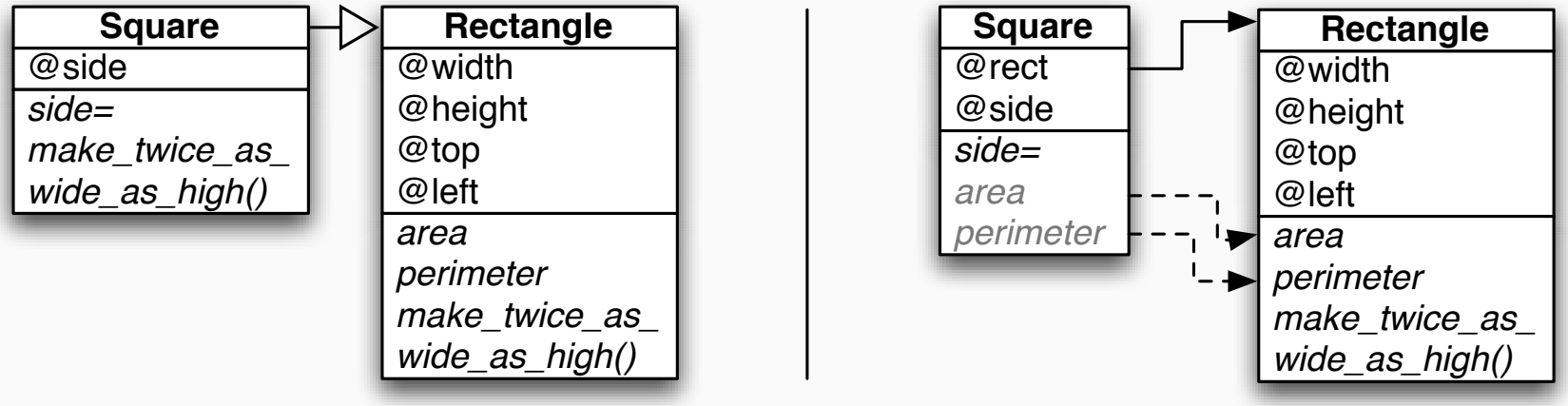
Será que um quadrado realmente é um retângulo?

```
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*)



Princípio de Demeter

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 sistema[3] 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
```

[3] Fonte: http://www.dan-manges.com/blog/37

EXEMPLO

Imagine um sistema[3] 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?

[3] Fonte: `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

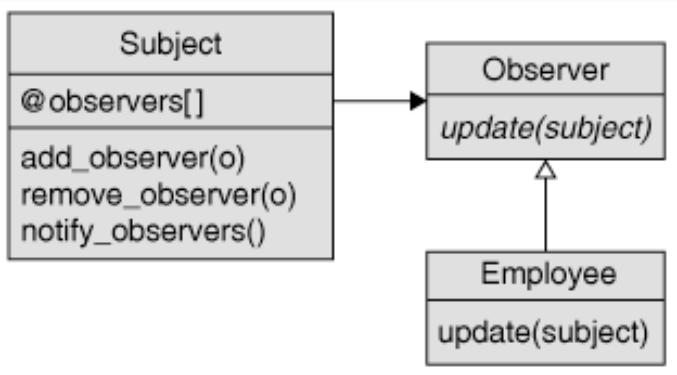
Essa solução é ainda melhor, agora o **comportamento** que é delegado. A implementação do comportamento pode ser mudada sem afetar **Paperboy**

```
class Wallet
  attr_reader :cash # agora é um atributo só de leitura!
  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
```

OBSERVER

- Problema: entidade O (“observador”) quer saber sobre certas coisas que podem acontecer com uma entidade S (“sujeito”)
- Problemas de projeto:
  - agir na ocorrência dos eventos é um problema de O — não queremos poluir S
  - qualquer tipo de objeto pode ser um observador ou um sujeito — herança seria esquisito

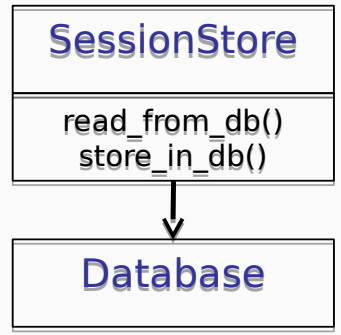
- Exemplos de casos de uso:
  - um indexador de textos quer ser notificado sobre novos posts
  - um auditor quer saber sobre quaisquer ações “sensíveis” realizadas por um admin

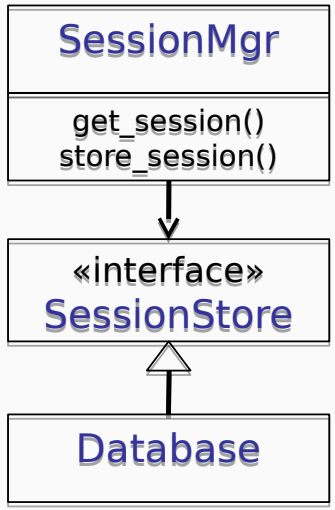


Princípio de Injeção de Dependência

INVERSÃO DE DEPENDÊNCIA & INJEÇÃO DE DEPENDÊNCIA

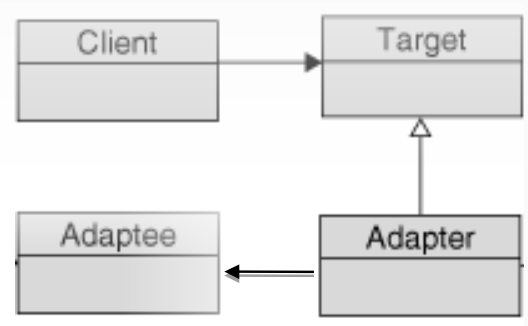
- Problema: **a** depende de **b**, mas a interface e a implementação de **b** podem mudar, mesmo que a funcionalidade já esteja estável
- Solução: "injetar" uma *interface abstrata* que será usada por **a** e **b**
  - se não houver uma correspondência exata, usar Adapter/Façade
  - "inversão": agora **b** (e **a**) depende da interface vs. **a** depende de **b**

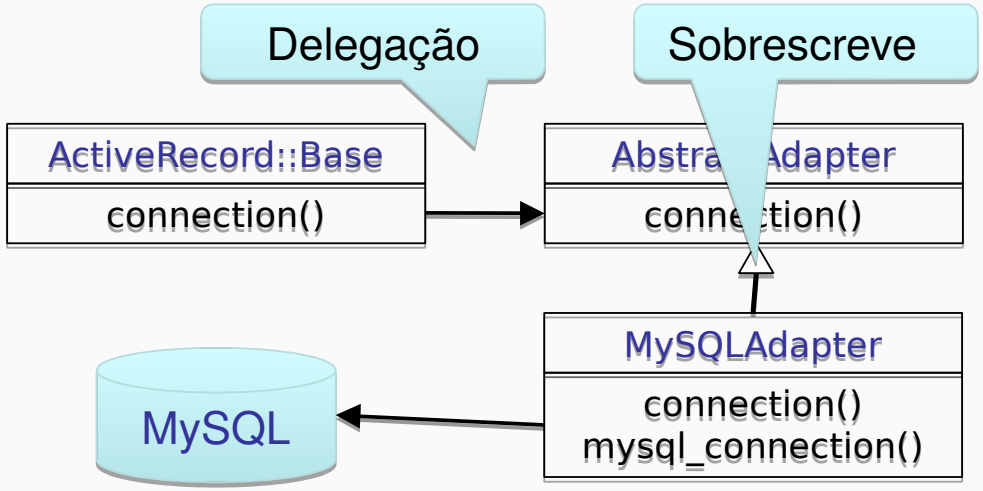




INJEÇÃO DE DEPENDÊNCIAS COM O PADRÃO ADAPTER

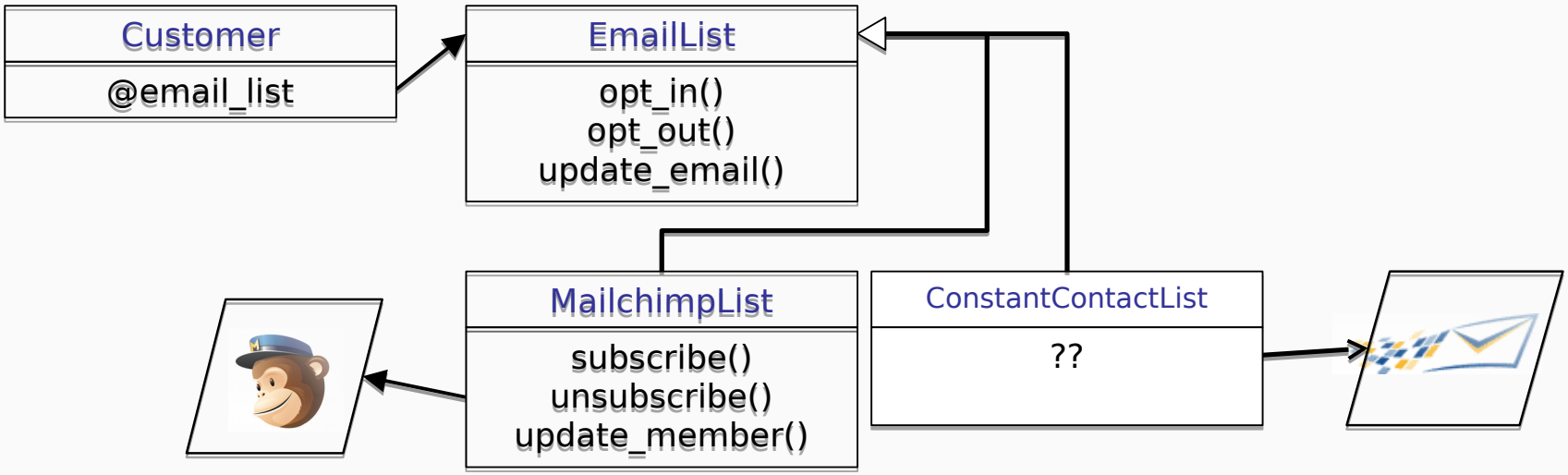
- Problema: cliente quer usar um “serviço”
  - serviço geralmente permite as operações necessárias
  - mas a API não é aquilo que o cliente espera
  - e/ou cliente precisa interoperar com múltiplos (mas ligeiramente diferentes) serviços
- Exemplo no Rails: “adaptadores” de banco de dados para MySQL, Oracle, PostgreSQL, ...





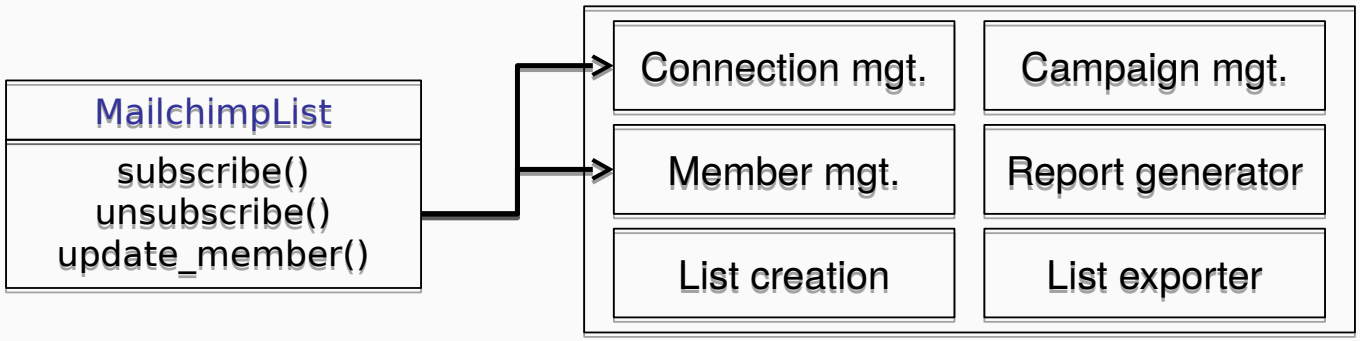
EXEMPLO: APOIO A SERVIÇOS EXTERNOS

- Suponha que você use serviços externos para enviar e-mails de marketing
- Ambos com APIs RESTful
- Ambos com funcionalidades semelhantes
  - Mantêm múltiplas listas, permitem adicionar/remover usuário(s) de lista(s), mudar preferências de inscrição de usuário, ...



RELACIONADO: FAÇADE

- Na verdade, nós usamos apenas um subconjunto de uma API muito mais elaborada
  - inicialização, gerenciamento de lista, início/fim de campanha, ...
- Então nosso adaptador também é uma *façade*
  - permite *unir* APIs distintas em uma única API simplificada



REFERÊNCIAS

- Artigo da Wikipédia sobre os Princípios SOLID (veja também o artigo de cada princípio individualmente): `https://en.wikipedia.org/wiki/SOLID`
- Avraam Piperidis. *Introduction to SOLID Design Principles for Java Developers*: `https://dzone.com/articles/a-gentle-and-easy-to-grasp-introduction-to-solid-p`
- Yiğit Kemal Erinç. *The SOLID Principles of Object-Oriented Programming Explained in Plain English*: `https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/`