PADRÕES, ANTIPADRÕES E SOLID

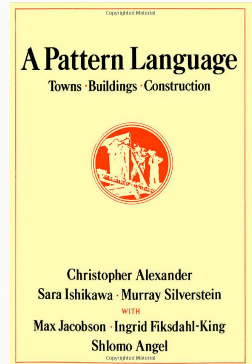
ACH2006 – ENGENHARIA DE SISTEMAS DE INFORMAÇÃO
SIN5005 – TÓPICOS EM ENGENHARIA DE SOFTWARE

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

"Um padrão descreve um problema que ocorre repetidamente, junto com uma solução testada para o problema" — Christopher Alexander, 1977

- Os 253 padrões arquiteturais (para construção civil) de Christopher Alexander vão desde a criação de cidades (2. distribuição de cidades) até problemas particulares de prédios (232. cobertura do telhado)
- Uma linguagem de padrões é uma forma organizada de abordar um problema arquitetural usando padrões
- Separa as coisas que mudam daquelas que sempre permanecem iguais (como DRY)

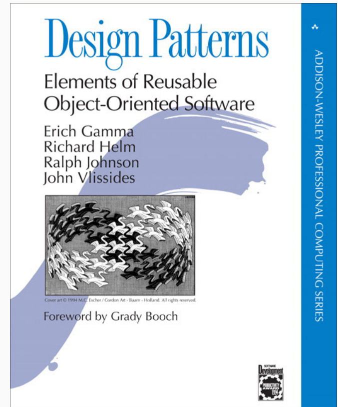


TIPOS DE PADRÕES EM SOFTWARE

- Padrões arquiteturais (“macroescala”)
  - Model–View–Controller
  - Pipe & Filter (ex: compiladores, Unix *pipeline*)
  - Event-based (ex: jogos interativos)
  - Layering (ex: pilha de tecnologias de SaaS)
- Padrões de computação
  - Transformada rápida de Fourier
  - Malhas (grids) estruturadas e não-estruturadas
  - Álgebra linear densa
  - Álgebra linear esparsa
- Padrões do GoF (Gang of Four): de criação, estrutural, comportamental

GANG OF FOUR (GOF)

- 23 padrões estruturais de projetos
- descrição das classes & objetos comunicantes
  - captura soluções comuns (e bem sucedidas) para um conjunto de problemas relacionados
  - pode ser personalizada para resolver um (novo) problema específico dessa categoria
- Padrão ≠
  - classes ou bibliotecas individuais (listas, *hash*, etc.)
  - projeto completo — está mais para um *blueprint* (desenho técnico) para o projeto



PADRÕES DE CRIAÇÃO

- São padrões relacionados à instanciação de classes
- Podem ser divididos em:
  - padrões de criação de classes (usam herança)
  - padrões de criação de objetos (usam delegação)

- Abstract Factory
- Builder
- Factory Method
- Object Pool
- Prototype
- Singleton

PADRÕES ESTRUTURAIS

São padrões relacionados à composição de classes (com herança) e objetos.

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Private Class Data
- Proxy

PADRÕES COMPORTAMENTAIS

São padrões que tratam de problemas relacionados à comunicação entre objetos.

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template method
- Visitor

Separa as coisas que mudam daquelas que permanecem as mesmas

1. Programar para uma Interface, não para uma implementação
2. Preferir composição & delegação à herança
    - delegação trata de compartilhar uma interface, herança trata de compartilhar implementação

- Código que parece que provavelmente segue algum padrão de projeto `#sqn`
- Geralmente é resultado de muita dívida técnica acumulada
- Sintomas:
    - Viscosidade (mais fácil corrigir usando um *hack* do que fazer a Coisa Certa)
    - Imobilidade (não dá para extrair uma funcionalidade porque ela faz parte do âmago do app)
    - Repetição desnecessária (consequência da imobilidade)
    - Complexidade desnecessária (generalidade inserida antes da necessidade)

Veja uma extensa lista de antipadrões em:
`http://wiki.c2.com/?AntiPatternsCatalog`

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

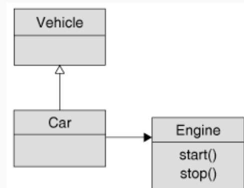
[1]Propostos por Robert C. Martin, coautor do Manifesto Ágil

Métodos dentro de uma classe	Relações entre classes
Mau cheiros de código	Mau cheiros de projetos
Muitos catálogos de cheiros de código & refatorações	Muitos catálogos de cheiros de projetos & padrões de projetos
Algumas refatorações são supérfluas em Ruby	Alguns padrões de projetos são supérfluos em Ruby
Métricas: ABC & Complexidade Ciclomática	Métricas: *Lack of Cohesion of Methods* (LCOM)
Refatore extraindo métodos e movendo código dentro de uma classe	Refatore extraindo classes e movendo código entre classes
SOFA: métodos são: **S**hort, do **O**ne thing, have **F**ew arguments, single level of **A**bstraction	SOLID: **S**ingle responsibility per class, **O**pen/closed principle, **L**iskov substitutability, **I**njection of dependencies, **D**emeter principle

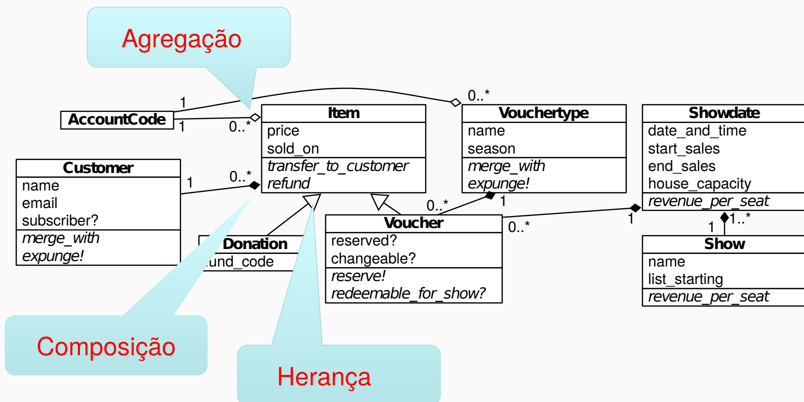
- Iremos ver exemplos de princípios SOLID, mostrando como os padrões de projeto podem ajudar
- **Não é necessário** lembrar de cor toda sintaxe, etc.
- Mas **é necessário** saber a ideia geral de cada princípio SOLID para que você fique atento para saber se está seguindo ou não

Um pouco sobre UML

- *Unified Modeling Language*: notação para descrever vários artefatos em sistemas orientados a objetos
- Um tipo de diagrama UML é o *diagrama de classe*, que mostra as relações e principais métodos de uma classe:

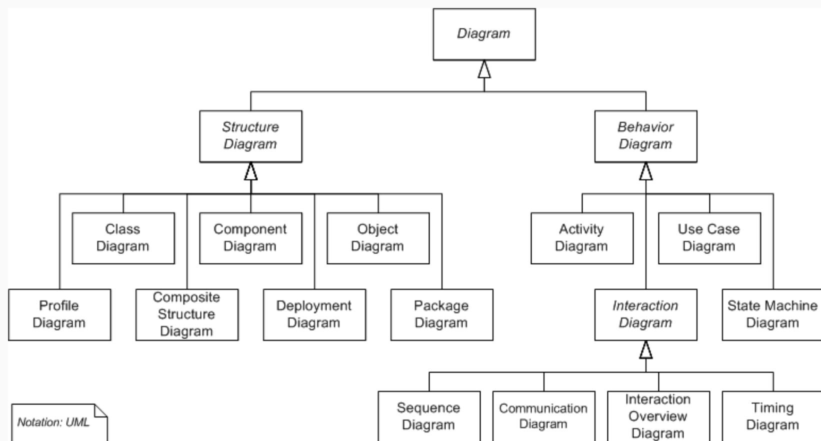


- `Car` é uma subclasse de `Vehicle`
- `Engine` é um componente de `Car`
- A classe `Engine` inclui os métodos `start()` e `stop()`



Agregação
AccountCode
1
1
0..*
0..*
Item
price
sold_on
transfer_to_customer
refund
Vouchertype
name
season
merge_with
expunge!
Showdate
date_and_time
start_sales
end_sales
house_capacity
revenue_per_seat
Customer
name
email
subscriber?
merge_with
expunge!
1
0..*
0..*
1
1
1..*
1
0..*
Donation
fund_code
Voucher
reserved?
changeable?
reserve!
redeemable_for_show?
Show
name
list_starting
revenue_per_seat
Composição
Herança

(UML EM EXCESSO)



CARTÕES CLASSE–RESPONSABILIDADE–COLABORAÇÃO (CRC)

(Proposto por Kent Beck & Ward Cunningham, OOPSLA’89)

Showing	
Responsibilities	*Collaborators*
Knows name of movie	Movie
Knows date & time	
Computes ticket availability	Ticket

Ticket	
Responsibilities	*Collaborators*
Knows its price	
Knows which showing it’s for	Showing
Computes ticket availability	
Knows its owner	Patron

Order	
Responsibilities	*Collaborators*
Knows how many tickets it has	Ticket
Computes its price	
Knows its owner	Patron
Knows its owner	Patron

```
Feature: Add movie tickets to shopping cart
  As a patron
  So that I can attend a showing of a movie
  I want to add tickets to my order

Scenario: Find specific showing
  Given a showing of "Inception" on Oct 5 at 7pm
  When  I visit the "Buy Tickets" page
  Then  the "Movies" menu should contain "Inception"
  And   the "Showings" menu should contain "Oct 5, 7pm"

Scenario: Find what other showings are available
  Given there are showings of "Inception" today at
  2pm,4pm,7pm,10pm
  When  I visit the "List showings" page for "Inception"
  Then  I should see "2pm" and "4pm" and "7pm" and "10pm"
```

- "Ter um projeto e um *schema* sólido nos salvou de muita dor de cabeça"
- "A separação de conceitos do MVC permitiu uma estrutura muito boa para o app"
- "Projetar o lado do cliente e o lado do servidor usando SOA facilitou o desacoplamento do código"
- "Algumas das técnicas para fazer stubs de SOA nos ajudaram a projetar um app cliente rico"

- "Gostaríamos de ter projetado o modelo de objetos e o *schema* com mais cuidado"

Motivação[2]: 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

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

Motivação[2]: 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

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

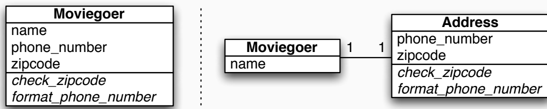
Princípio da responsabilidade única

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

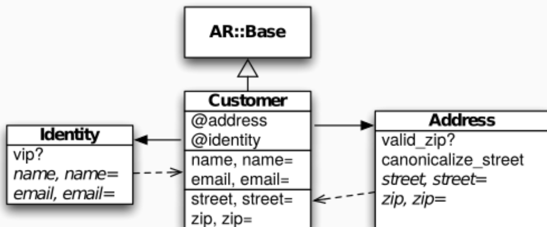
FALTA DE COESÃO DOS MÉTODOS

- Henderson–Sellers (revisado):
  LCOM = $1 - (\sum_i MV_i / M \times V)$, valor entre 0 e 1
    - M = # de métodos de instância
    - V = # de variáveis de instância
    - MV_i = # de métodos de instância que acessam a i-ésima variável de instância (excluindo *getters* e `setters` triviais)
- LCOM-4: mede o número de componentes conexos em um grafo onde métodos relacionados são ligados por uma aresta
- LCOM alto sugere possíveis violações do PRU



http://pastebin.com/bjdaTWN8

- `has_one` ou `composed_of`?
- usar composição & delegação?



http://pastebin.com/XESSSNb6

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**

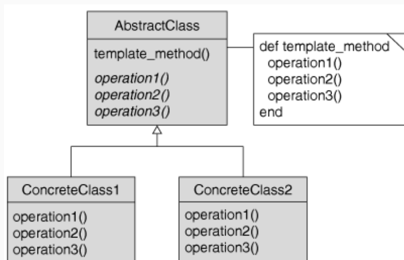
```
class Report
  def output_report
    case @format
    when :html
      HtmlFormatter.new(self).output
    when :pdf
      PdfFormatter.new(self).output
```

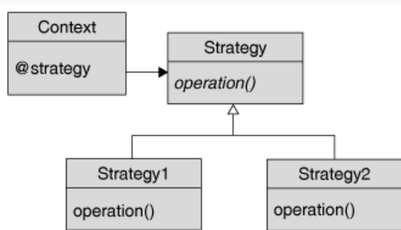
- Não é possível estender (adicionar novos tipos de relatórios) sem mudar a classe base `Report`
    - não é tão ruim quanto em linguagens estaticamente tipadas... mas é feio

- 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

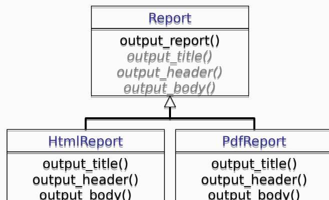
Veja: `http://pastebin.com/p3AHMqHZ`

- *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

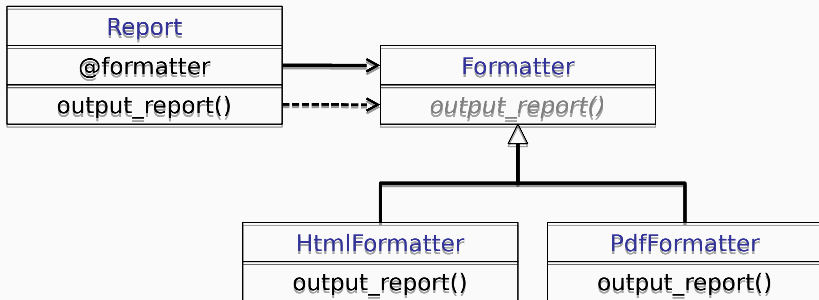




```ruby
class Report
 attr_accessor :title, :text
 def output_report
  output_title
  output_header
  output_body
 end
end

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

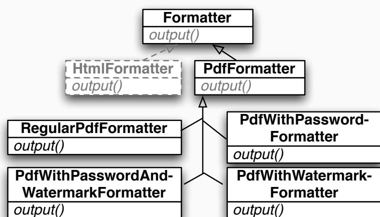


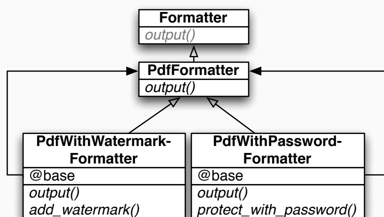
```ruby
class Report
 attr_accessor :title, :text, :formatter
 def output_report
  formatter.output_report
 end
end
```



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!

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)
- 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 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?
```

```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
```

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

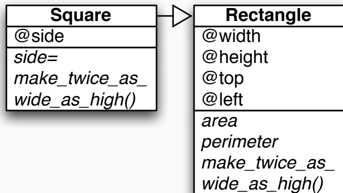
Será que um quadrado realmente é um retângulo?

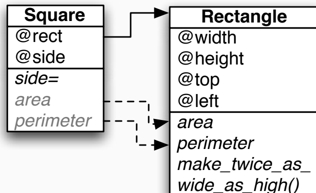
```
def make_twice_as_wide_as_high(r, dim)
  r.width = 2*dim
  r.height = dim
end
# viola PSL se esse método for parte do seu "contrato"!
```

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*)

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`

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

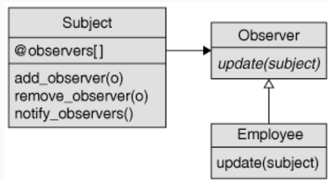
```
class Customer
  def cash
    self.wallet.cash
  end
end

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

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

```ruby
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
```

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

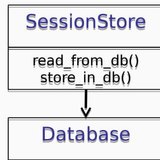


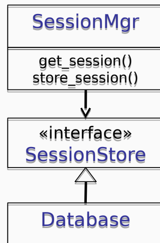
- Problema: apagar um cliente que “possui” transações prévias (ex: uma chave estrangeira aponta pra ele)
- Possível solução: substituir referências ao cliente por referências a um “cliente desconhecido”
- `ActiveRecord` provê ganchos para o padrão de projetos Observer

```
class CustomerObserver < ActiveRecord::Observer
  observe :customer # inferido se necessário (convenção)
  def before_destroy ... end
end

# em config/environment.rb
config.active_record.observers = :customer_observer
```

Princípio de 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
- Equivalente em Ruby: extraia um módulo para isolar a interface





- O que está errado com esse código em uma view?

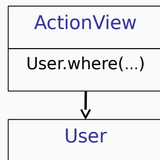
```
@vips = User.where('group="VIP"')
```

- O que está errado com esse código em uma view?

```ruby
@vips = User.where('group="VIP"')
```

- Um pouco melhor:

```ruby
@vips = User.find_vips
```



- O que está errado com esse código em uma view?

```ruby
@vips = User.where('group="VIP"')
```

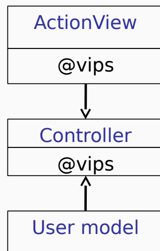
- Um pouco melhor:

```ruby
@vips = User.find_vips
```

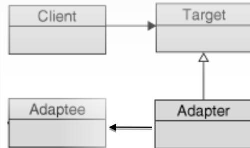
- Agora sim:

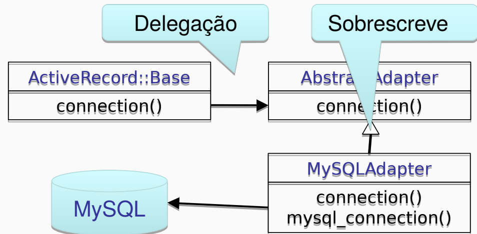
```ruby
# no controller
@vips = User.find_vips
```

Independente de como VIPs são representados no modelo!

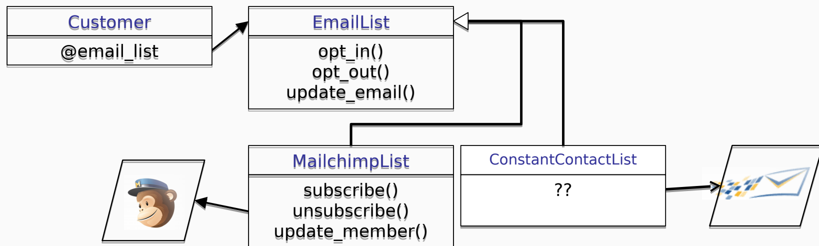


- Problema: cliente quer usar um “serviço”
  - serviço geralmente permite as operações necessárias
  - mas a API não é aquilo que o cliente espera
  - e/ou cliente precisa interoperar com múltiplos (mas ligeiramente diferentes) serviços
- Exemplo no Rails: “adaptadores” de banco de dados para MySQL,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

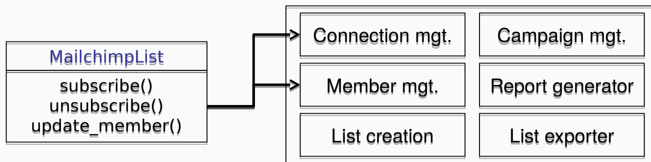




- Suponha que você use serviços externos para enviar e-mails de marketing
- Ambos com APIs RESTful
- Ambos com funcionalidades semelhantes
  - Mantêm múltiplas listas, permitem adicionar/remover usuário(s) de lista(s), mudar preferências de inscrição de usuário, ...



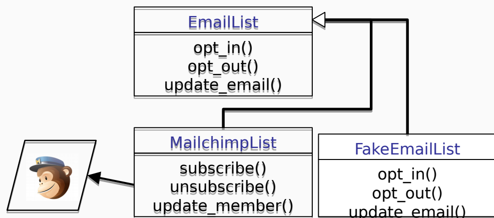
- Na verdade, nós usamos apenas um subconjunto de uma API muito mais elaborada
    - inicialização, gerenciamento de lista, início/fim de campanha, ...
- Então nosso adaptador também é uma *façade*
    - permite *unir* APIs distintas em uma única API simplificada



Mais padrões relacionados ao Adapter

- Problema: você quer *invariantes* para simplificar o projeto, mas os requisitos do app parece quebrá-los
- *Null Object*: substituto onde métodos "importantes" podem ser chamados sem consequências

```
@customer = Customer.null_customer
@customer.logged_in?  # => false
@customer.last_name   # => "ANÔNIMO"
@customer.is_vip?     # => false
```



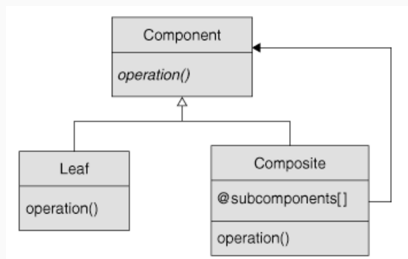
- Tecnicamente, uma classe que provê 1 instância, a qual todos podem acessar
- Ruby permite implementar Singleton de um modo bastante elegante, usando *singleton classes* (que não tem relação com implementar um singleton!)
    - uma `$variável` global? Outros poderiam mudá-la
    - uma `CONSTANTE`? Não há como controlar quando ela será inicializada
- Um objeto singleton no fundo é um membro de uma classe, mas que é imutável e único
- Veja um exemplo de implementação em `http://pastebin.com/RBuvPMkR`

- Proxy implementa os *mesmos métodos* que os oferecidos por um objeto “real”, mas “intercepta” cada chamada
  - para fazer autenticação / proteger acesso
  - adiar um trabalho (ser *lazy*)
  - exemplo do Rails: *association proxies*
- Exemplo:

```ruby
class Blog < ActiveRecord::Base
  has_many :posts
end

blog = Blog.find(:first)
blog.posts.count # os posts não serão carregados
```

- Componente no qual operações fazem sentido se aplicadas tanto em um indivíduo quanto em um conjunto
- Exemplo: ingressos comuns, ingressos VIPs ou assinaturas

- O que eles têm em comum?
  - um preço
  - podem ser adicionados a uma compra
- O que têm de diferente?
  - Ingressos comuns & VIP são para um concerto específico
  - Uma assinatura tem que manter quais ingressos ela contém



COMPOSITE

- O *single-table inheritance* do Rails armazena objetos de subclasses diferentes (mas com a mesma classe base) em uma mesma tabela
  - o Rails automaticamente gerencia uma coluna para armazenar o tipo da subclasse do Ruby
  - permite definir validações, associações, etc. separadas em cada subclasse
- Permite implementar o *Composite* e alguns outros padrões

```ruby
class Ticket < ActiveRecord::Base
class RegularTicket < Ticket
class VIPTicket < Ticket
class Subscription < Ticket
```

A perspectiva do Planeje-e-Documente sobre Padrões de Projetos

PADRÕES DE PROJETO P-E-D?

- Quais são os prós e contras para Planeje-e-Documente do ponto de vista de Padrões de Projetos?
- Em qual tipo de metodologia é mais fácil usar Padrões de Projetos? Métodos Planeje-e-Documente ou Métodos Ágeis?

- Um planejamento cuidadoso pode resultar em uma boa arquitetura de software
    - conhecida como *Big Design Up Front*)
- Quebra a Especificação de Requisitos de Software (SRS) em problemas
- Para cada problema, procura por padrões de projetos que a resolveriam
    - e, então, por padrões para os subproblemas
- Revisões de Projeto podem ajudar

- Crítica à Ágil: encoraja desenvolvedores a começar a programar antes de ter algum projeto
    - depende muito de refatoração depois
- Crítica à P-e-D: não há código até que o projeto esteja completo $\Rightarrow$ não há confiança de que o projeto seja implementável ou que case com as necessidades do cliente
    - quando a codificação começa, percebe-se que o projeto precisa mudar

- Conselho Ágil: se você já realizou algum projeto com restrições ou elementos de projeto parecidos, então tudo bem se planejar para fazer algo parecido; a experiência provavelmente vai levar a decisões de projeto razoáveis
- ex: planejamento para armazenamento de dados para um app SaaS, mesmo que num primeiro momento BDD/TDD não incitem o uso de BD
- ex: pensar em como fazer para garantir escalabilidade horizontal (mais no Cap. 12) desde o início

- GoF distingui os conceitos de padrões de projetos e arcabouços (*frameworks*)
    - padrões são mais abstratos, com um foco mais restrito e não são direcionados a um domínio específico
- Ainda assim, arcabouços são uma ótima forma de um novato aprender a usar padrões de projetos
    - ganhe experiência para criar código baseado em padrões de projeto examinando os padrões usados em arcabouços

Resumo sobre Padrões de Projeto & SOLID

- Adapter (conexão ao banco de dados)
- Abstract Factory (conexão ao banco)
- Observer (caching, ver Cap. 12)
- Proxy (associações nas coleções de AR)
- Singleton (*Inflector*)
- Decorator (escopos AR, `alias_method_chain`)
- Command (migrações)
- Iterator (em todo lugar)
- Tipagem pato (*duck typing*) simplifica a implementação da maioria desses padrões ao "enfraquecer" o acoplamento introduzido por herança

- Desenvolvido para linguagens estaticamente tipadas, por isso alguns princípios são mais importantes para elas
    - “evita mudanças que modificam o tipo da assinatura” (geralmente implica em mudanças no contrato) — mas Ruby geralmente não usa tipos para nada
    - “evita mudanças que criem a necessidade de recompilar” — mas Ruby não é compilado
- Use seu bom senso: o objetivo é *entregar rapidamente código que funciona & que pode ser mantido*

RESUMO

- Padrões de Projeto representam soluções bem sucedidas para classes de problemas
  - reuso de projeto ao invés de código/classe
  - alguns padrões voltaram a ficar importantes em Rails porque são úteis para SaaS
- Podem ser aplicados em vários níveis: arquitetura, projeto (padrões GoF), computação
- Separa o que muda daquilo que permanece o mesmo
  - programe para uma interface, não para uma implementação
  - prefira composição a herança
  - delegue!
  - todos os 3 muito mais fáceis com tipagem pato
- Há muito mais para ler & aprender — isso é só uma introdução