PADRÕES, ANTIPADRÕES E SOLID

ENGENHARIA DE SISTEMAS DE INFORMAÇÃO

Daniel Cordeiro

28 de novembro de 2017

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

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[2] 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
```

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

EXEMPLO

Imagine um sistema[2] 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?

[2] 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

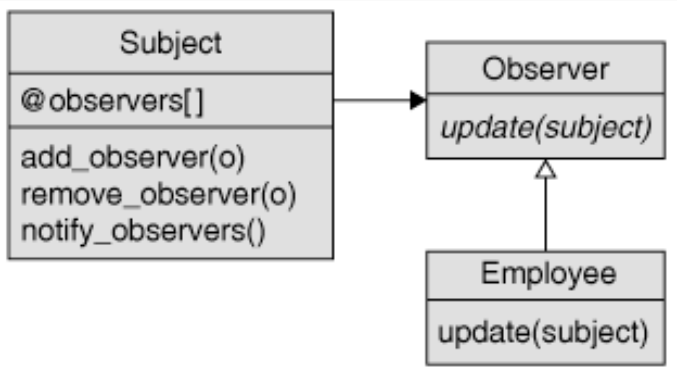
Essa solução é ainda melhor, agora o **comportamento** que é delegado. A implementação do comportamento pode ser mudada sem afetar `Paperboy`

```
class Wallet
  attr_reader :cash # 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



EXEMPLO: MANTENDO A INTEGRIDADE RELACIONAL

- 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

```ruby
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
```

PERGUNTA

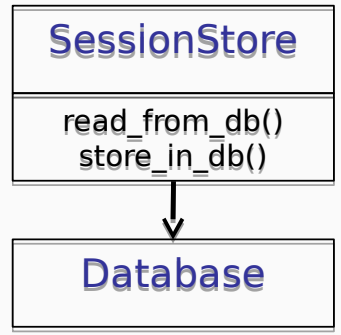
Suponha que `Order` pertença a `Customer` e a visão tem um `@order.customer.name`... isso é uma violação do princípio de Demeter?

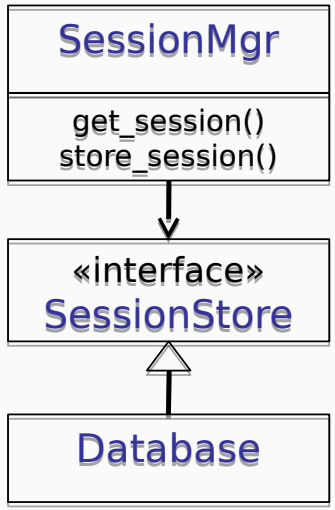
1. Sim... mas nesse caso provavelmente é razoável expor o grafo de objetos na visão
2. Sim... substitua por `Order#customer_name`, que delega para `Customer#name`
3. Sim... você pode argumentar que qualquer uma das duas soluções acima são adequadas
4. Não... de qualquer jeito, ao usar `belongs_to` nós já estamos expondo informação sobre o `Customer`

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





PID NO RAILS: EXEMPLO

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

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

PID NO RAILS: EXEMPLO

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

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

- Um pouco melhor:

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

PID NO RAILS: EXEMPLO

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

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

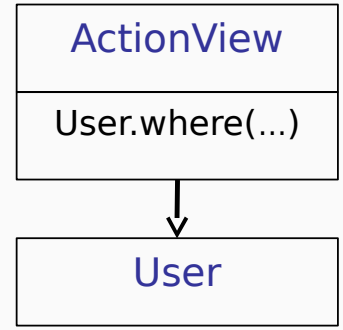
- Um pouco melhor:

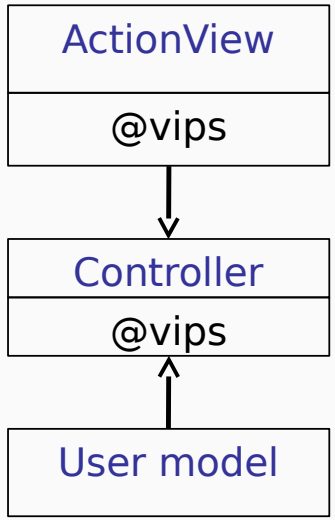
```
@vips = User.find_vips
```

- Agora sim:

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

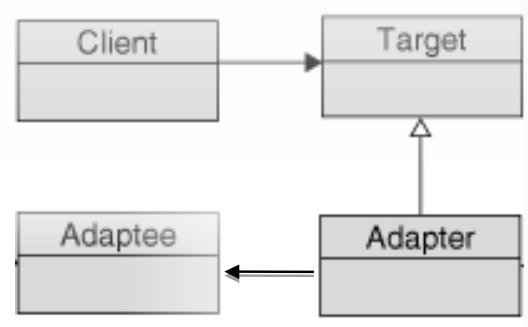
Independente de como VIPs são representados no modelo!

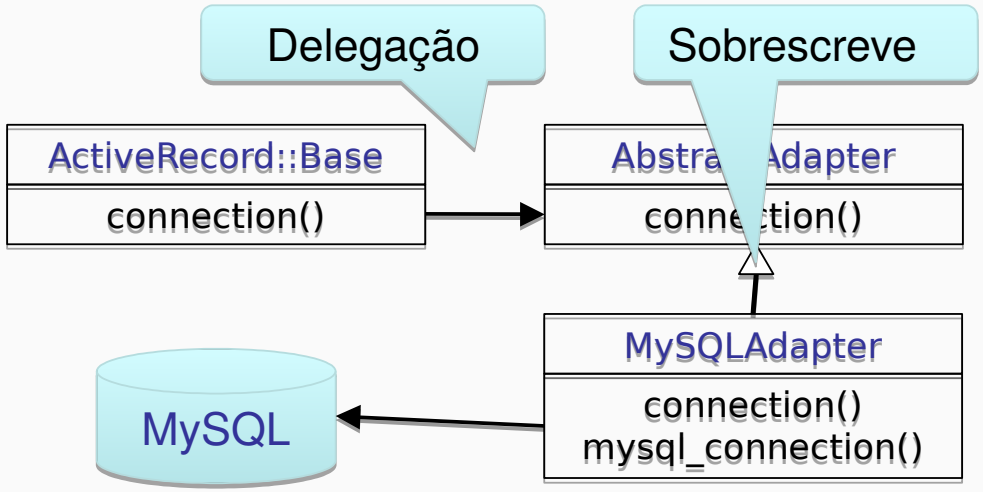




INJEÇÃO DE DEPENDÊNCIAS COM O PADRÃO ADAPTER

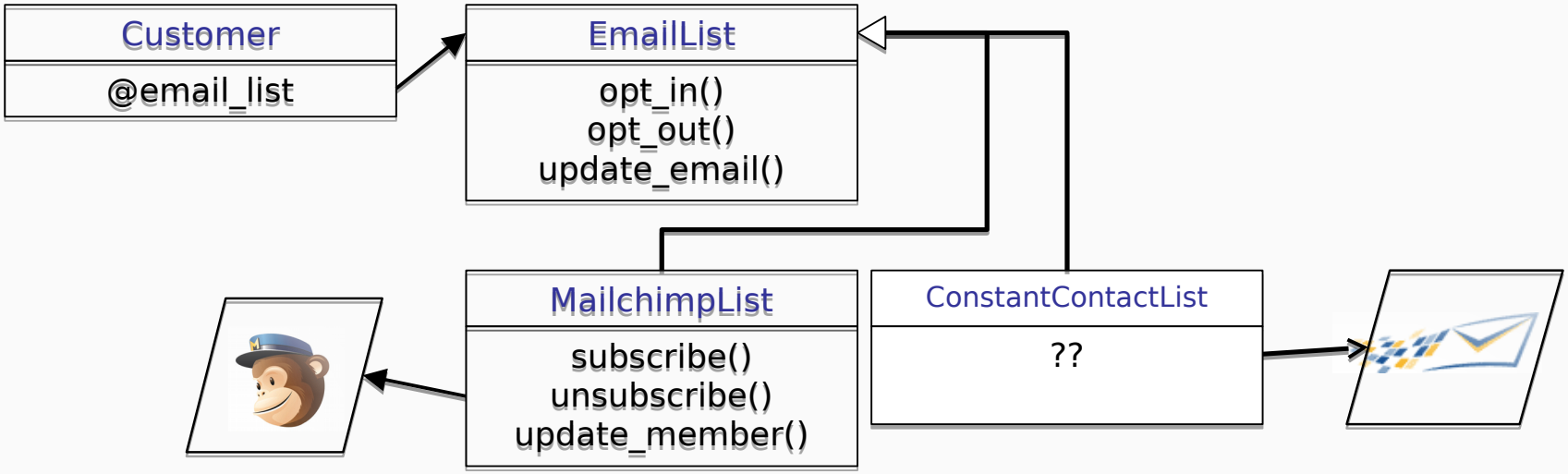
- Problema: cliente quer usar um “serviço”
  - serviço geralmente permite as operações necessárias
  - mas a API não é aquilo que o cliente espera
  - e/ou cliente precisa interoperar com múltiplos (mas ligeiramente diferentes) serviços
- Exemplo no Rails: “adaptadores” de banco de dados para MySQL,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





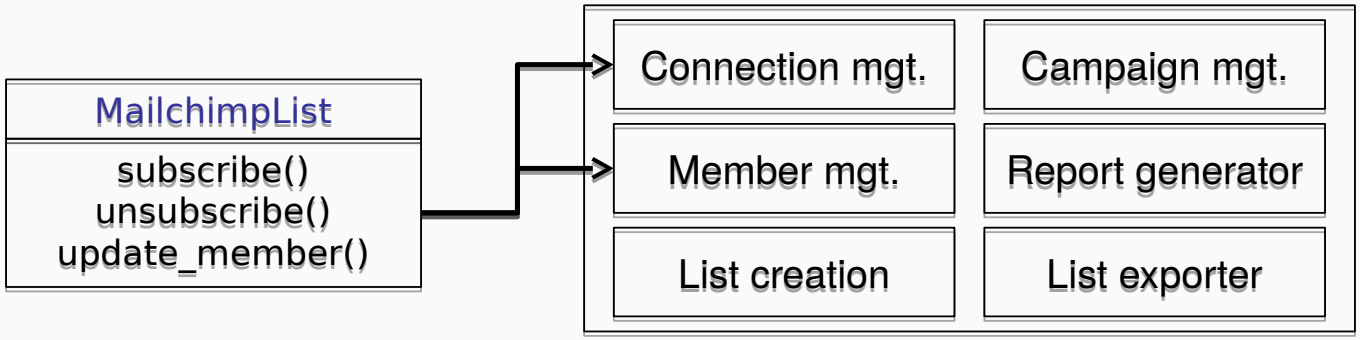
EXEMPLO: APOIO A SERVIÇOS EXTERNOS

- 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



PERGUNTA

Nos testes de controlador no RSpec, é normal criar stubs para `ActiveRecord::Base.where`, um método que é herdado. Qual afirmação está correta sobre tais testes:

(a) O teste do controlador está ligeiramente acoplado ao modelo

(b) Em uma linguagem estática, nós teríamos usado Inversão de Dependência para conseguir fazer a mesma coisa em um arcabouço de testes

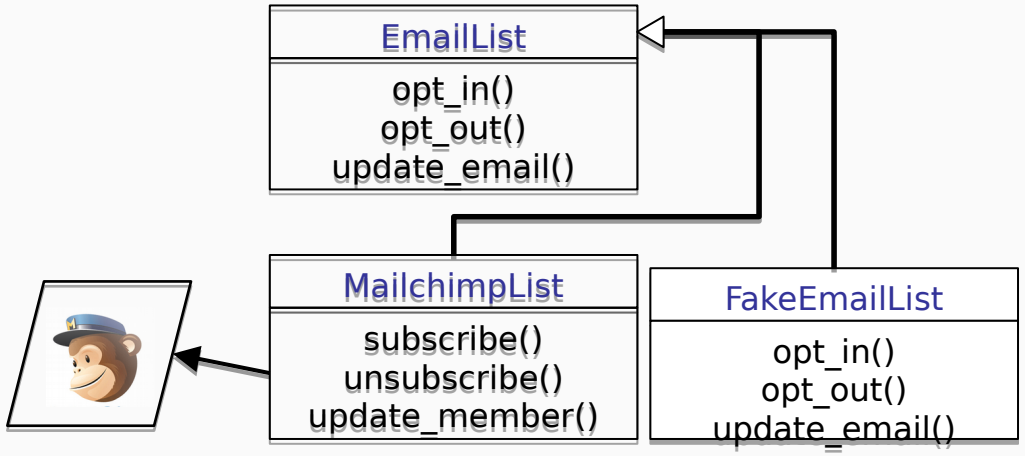
1. apenas (a)
2. apenas (b)
3. tanto (a) quanto (b)
4. nem (a) nem (b)

Mais padrões relacionados ao Adapter

NULL OBJECT

- 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
```



SINGLETON

- 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:

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

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

PERGUNTA

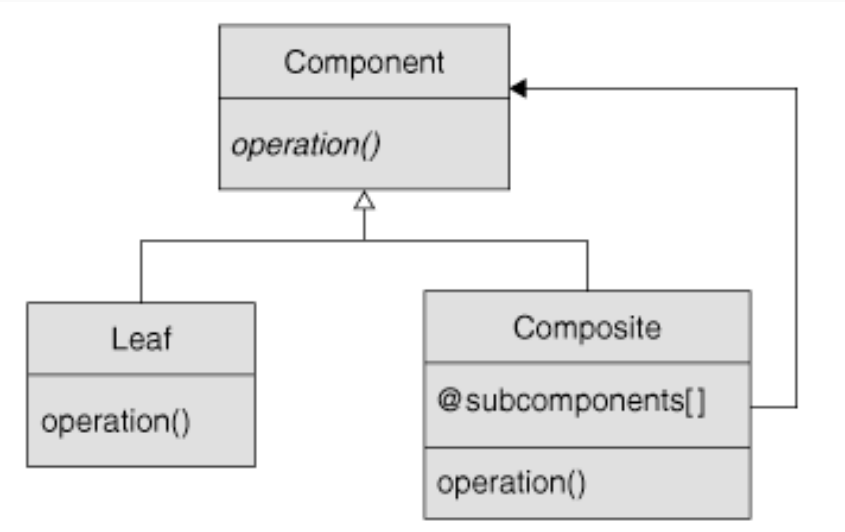
O uso do `FakeWeb` para criar um stub para requisições SOA a um serviço externo durante um teste é um exemplo de qual padrão de projeto?

- Null Object
- Proxy
- Adapter
- Façade

COMPOSITE

- 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



- 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
```

PERGUNTA

Quando usamos o padrão Composite, um risco que devemos evitar ao decidir quais comportamentos vão na superclasse e quais vão na subclasse é:

- violar o Princípio de Substituição de Liskov
- violar o Princípio Aberto/Fechado
- *pandemônio de mocks* quando for escrever os testes para a composição e para as classes folha
- não conseguir tirar proveito da herança single-table para armazenar instâncias de composições e de classes folha

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?

Abordagem P-E-D para padrões de projetos

- 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

QUANTO PROJETAR ANTECIPADAMENTE?

- 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

PERGUNTA

Qual afirmação relacionada a Padrões de Projetos é falsa (se houver)?

1. Processos P-e-D possuem uma fase de projeto explícita que é o local natural para o uso de padrões de projeto e, portanto, para garantir uma boa arquitetura de software
2. Métodos Ágeis não possuem uma fase de arquitetura, então há um risco maior de terminar com uma arquitetura ruim de software
3. Desenvolvedores Ágeis podem planejar a arquitetura do software e o uso de padrões de projeto que eles esperam usar (baseados em projetos anteriores e similares)
4. Nenhuma é falsa, todas são verdadeiras

Resumo sobre Padrões de Projeto & SOLID

ALGUNS PADRÕES VISTOS NO RAILS

- 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

CUIDADO COM SOLID

- 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

PERGUNTA

O módulo `ActiveRecord` do Rails define um `AbstractAdapter` para conexão com bancos de dados. Subclasses de `AbstractAdapter` existem para cada tipo de banco de dados e podem ser adicionados para novos SGBDs; quando o app inicia, o tipo correto é instanciado, baseado na configuração em `config/database.yml`. Qual princípio SOLID não é ilustrado nesse exemplo:

1. Princípio Aberto/Fechado
2. Princípio da Injeção de Dependência
3. Princípio de Demeter
4. Princípio da Substituição de Liskov