PADRÕES, ANTIPADRÕES E SOLID

ENGENHARIA DE SISTEMAS DE INFORMAÇÃO

Daniel Cordeiro

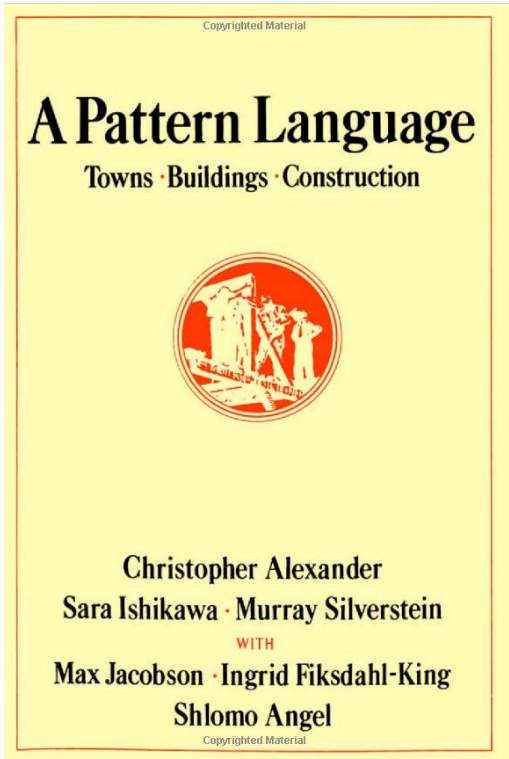
21 de novembro de 2017

Escola de Artes, Ciências e Humanidades | EACH | USP

PADRÕES DE PROJETO PROMOVEM REUSABILIDADE

"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)

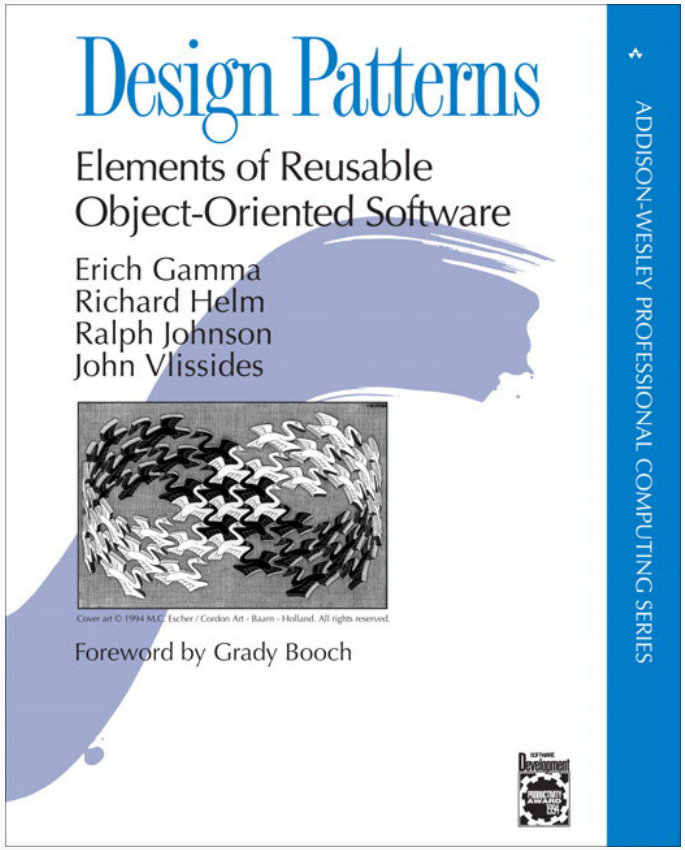


TIPOS DE PADRÕES EM SOFTWARE

- 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

META-PADRÕES

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

ANTIPADRÃO

- 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

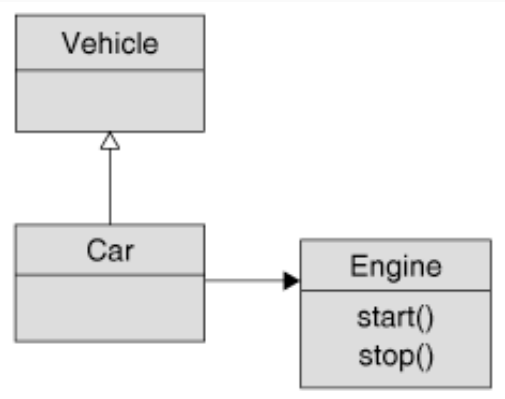
A SEGUIR

- 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

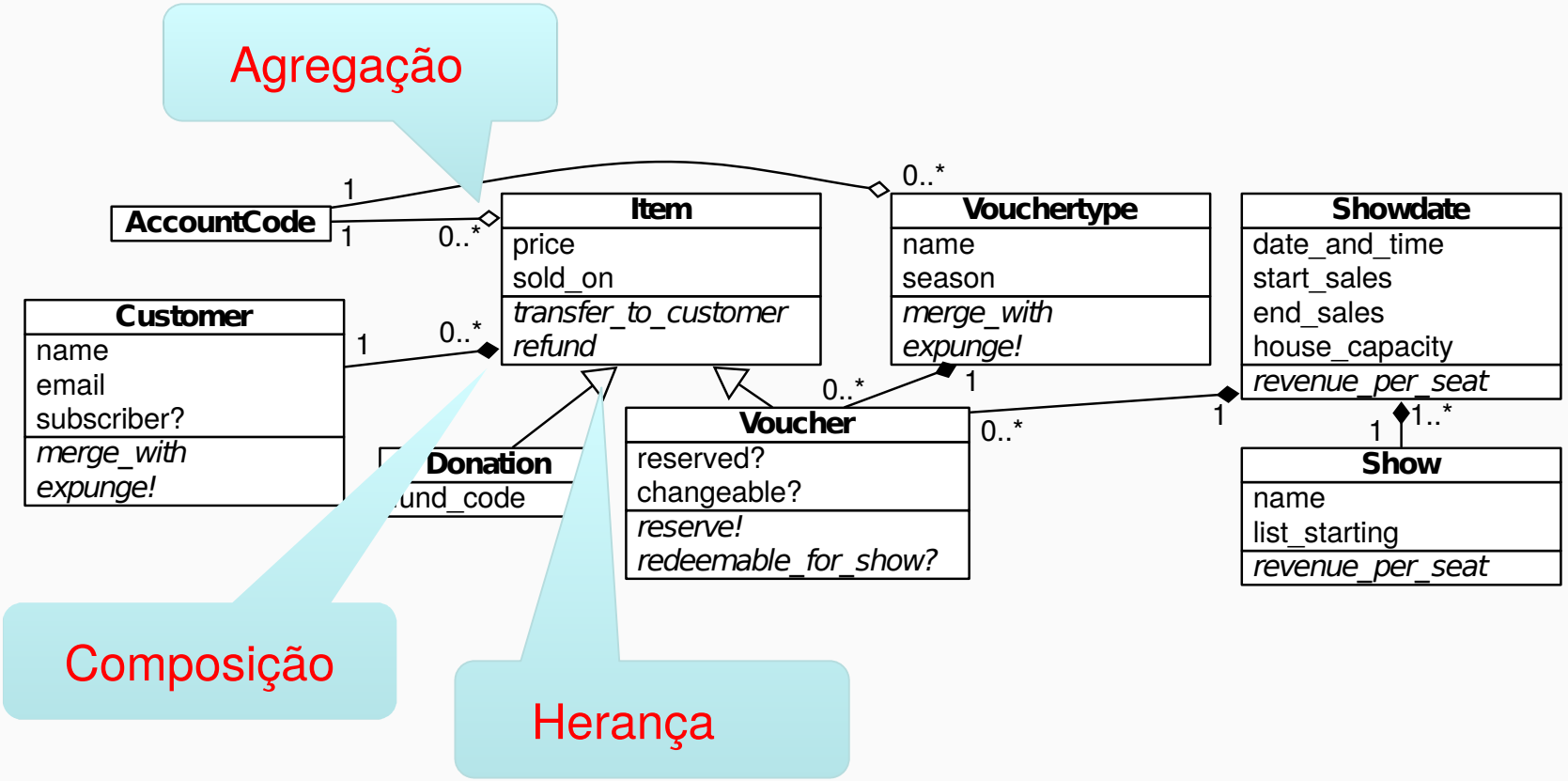
O MÍNIMO NECESSÁRIO SOBRE UML

- *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:

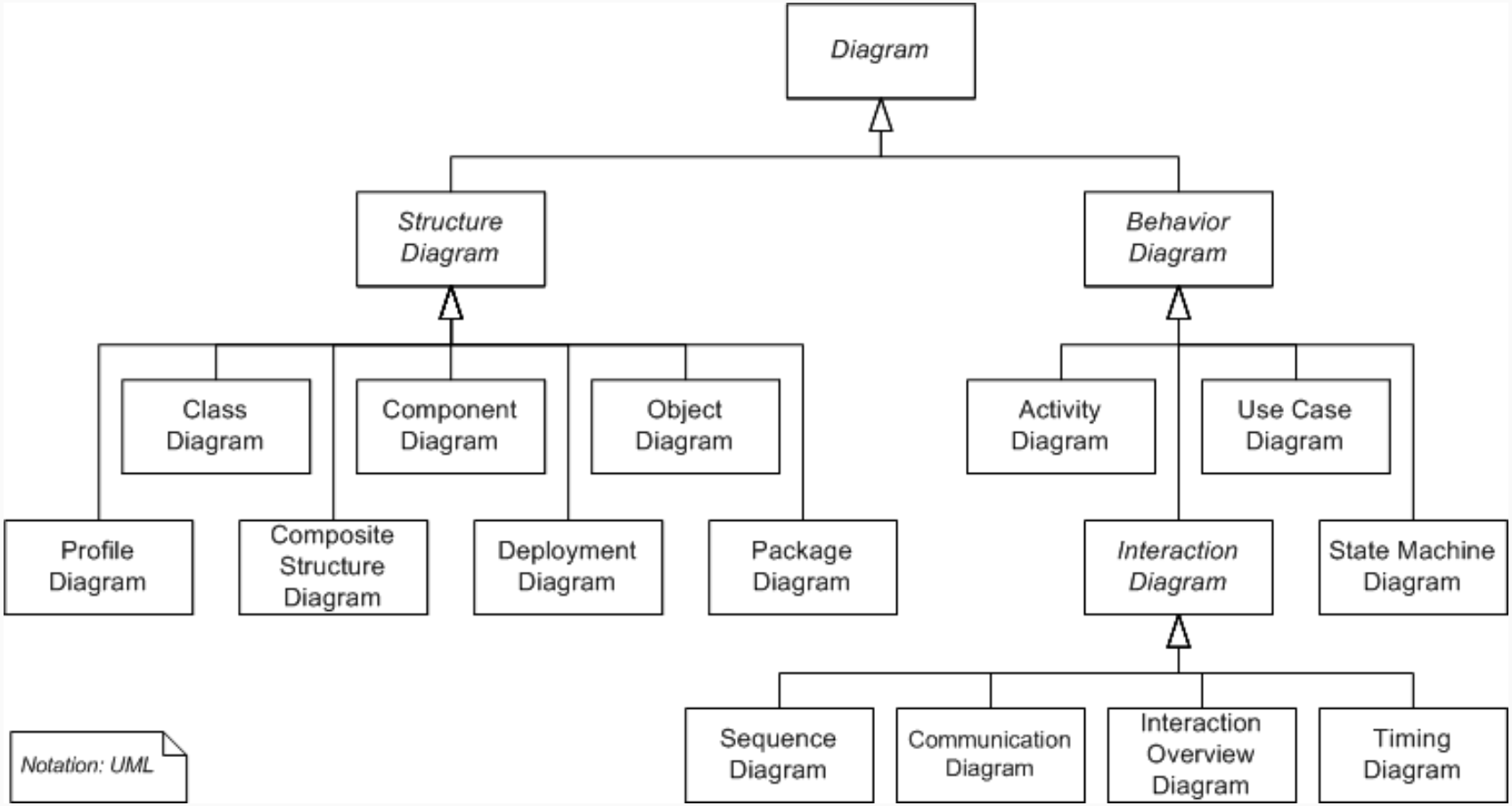


- `Car` é uma subclasse de `Vehicle`
- `Engine` é um componente de `Car`
- A classe `Engine` inclui os métodos `start()` e `stop()`

RELAÇÕES


Agregação
AccountCode
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!
Donation
und_code
Voucher
reserved?
changeable?
reserve!
redeemable_for_show?
Show
name
list_starting
revenue_per_seat
1
0..*
1
0..*
0..*
1
1
0..*
0..*
1
1
1..*
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

CRC & HISTÓRIAS DE USUÁRIO

```
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"
```

CARTÕES CRC / PLANEJAMENTO ANTECIPADO

☺

- “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”

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

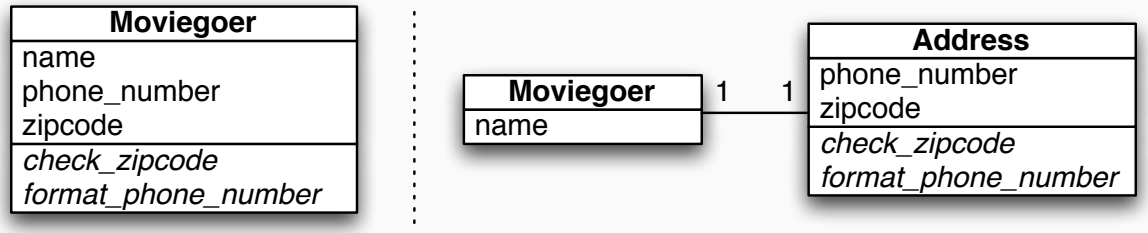
FALTA DE COESÃO DOS MÉTODOS

- 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

SERÁ QUE ACTIVERECORDS VIOLAM PRU?

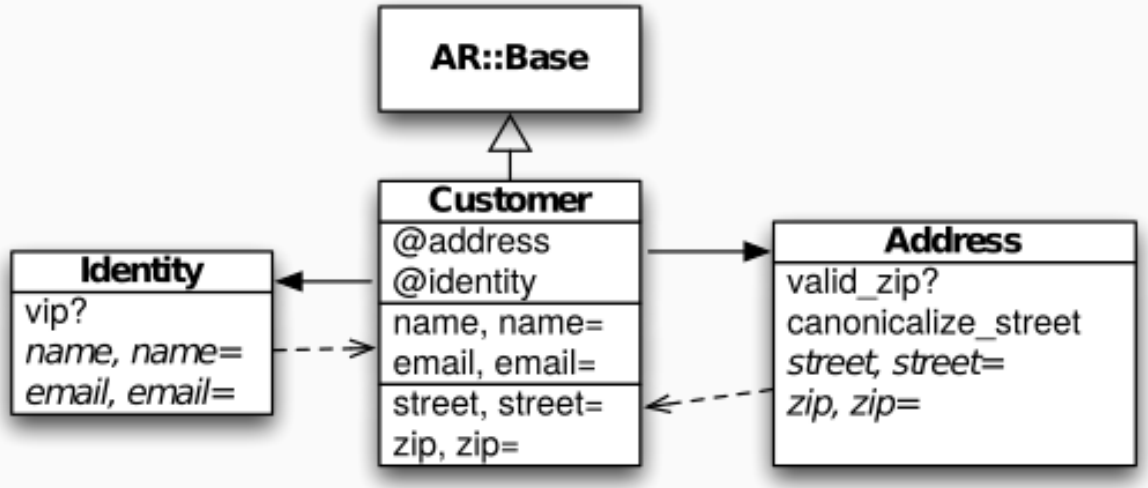
- Eles parecem misturar comportamentos em 1 classe
  - sabem como ler, gravar e apagar a si mesmos
  - sabem sobre as associações
  - sabem sobre as validações
  - isso tudo além da lógica de negócio específica do modelo
- Apesar de que a maior parte dos comportamentos são incluídos como módulos

EXTRAIA UM MÓDULO OU UMA CLASSE



http://pastebin.com/bjdaTWN8

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



http://pastebin.com/XESSSNb6

Qual afirmação é verdadeira sobre a observancia do princípio da responsabilidade única?

1. Em geral, nós esperamos ver uma correlação entre nota baixa para a coesão e uma nota baixa para as métricas SOFA
2. Pouca coesão é um possível indicador de uma oportunidade para extrair uma classe
3. Se uma classe respeita PRU, seus métodos provavelmente respeitarão SOFA
4. Se um método de classe respeita SOFA, a classe provavelmente respeita PRU