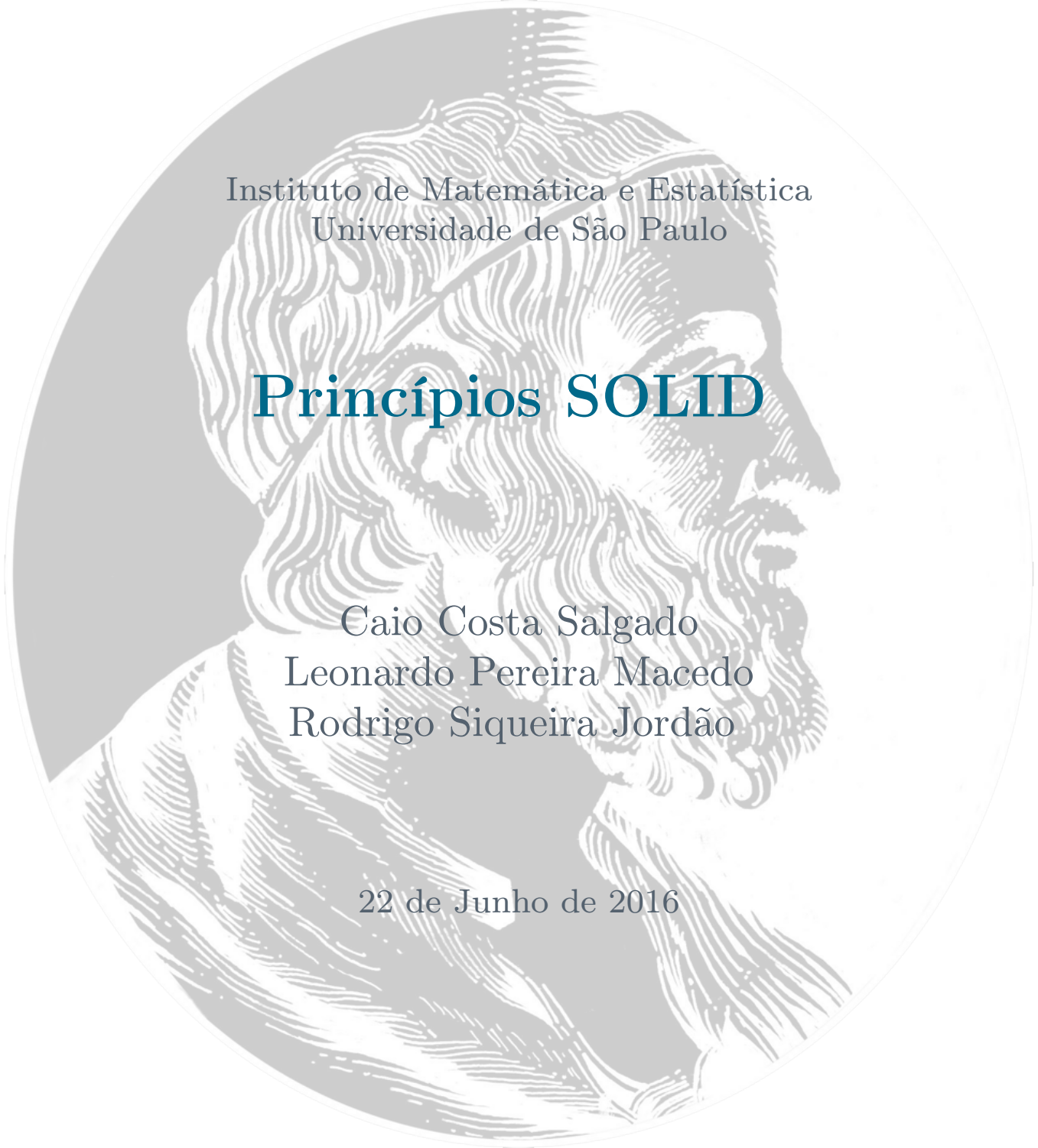


Instituto de Matemática e Estatística
Universidade de São Paulo

Princípios SOLID

Caio Costa Salgado
Leonardo Pereira Macedo
Rodrigo Siqueira Jordão

22 de Junho de 2016

Sumário

- **Padrões de projeto**
  - Solução reutilizável de um problema em design de *software*
  - Melhoram a qualidade e organização das classes

Introdução
Padrões e Antipadrões

- **Padrões de projeto**
  - Solução reutilizável de um problema em design de *software*
  - Melhoram a qualidade e organização das classes
- **Antipadrões**
  - Ineficientes, arriscados e contraprodutivos
  - Podem ser identificados pelo **mau cheiro de projeto** que criam

Princípios SOLID
Introdução

- Acrônimo para 5 princípios de *POO*
- Criado por Robert C. Martin *(Uncle Bob)* por volta do ano 2000

Princípios SOLID
Introdução

- Acrônimo para 5 princípios de *POO*
- Criado por Robert C. Martin *(Uncle Bob)* por volta do ano 2000
- Diminui o acoplamento entre classes
- Separa as responsabilidades para melhorar o código da aplicação desenvolvida

Acrônimo SOLID
Introdução

S ingle Responsibility

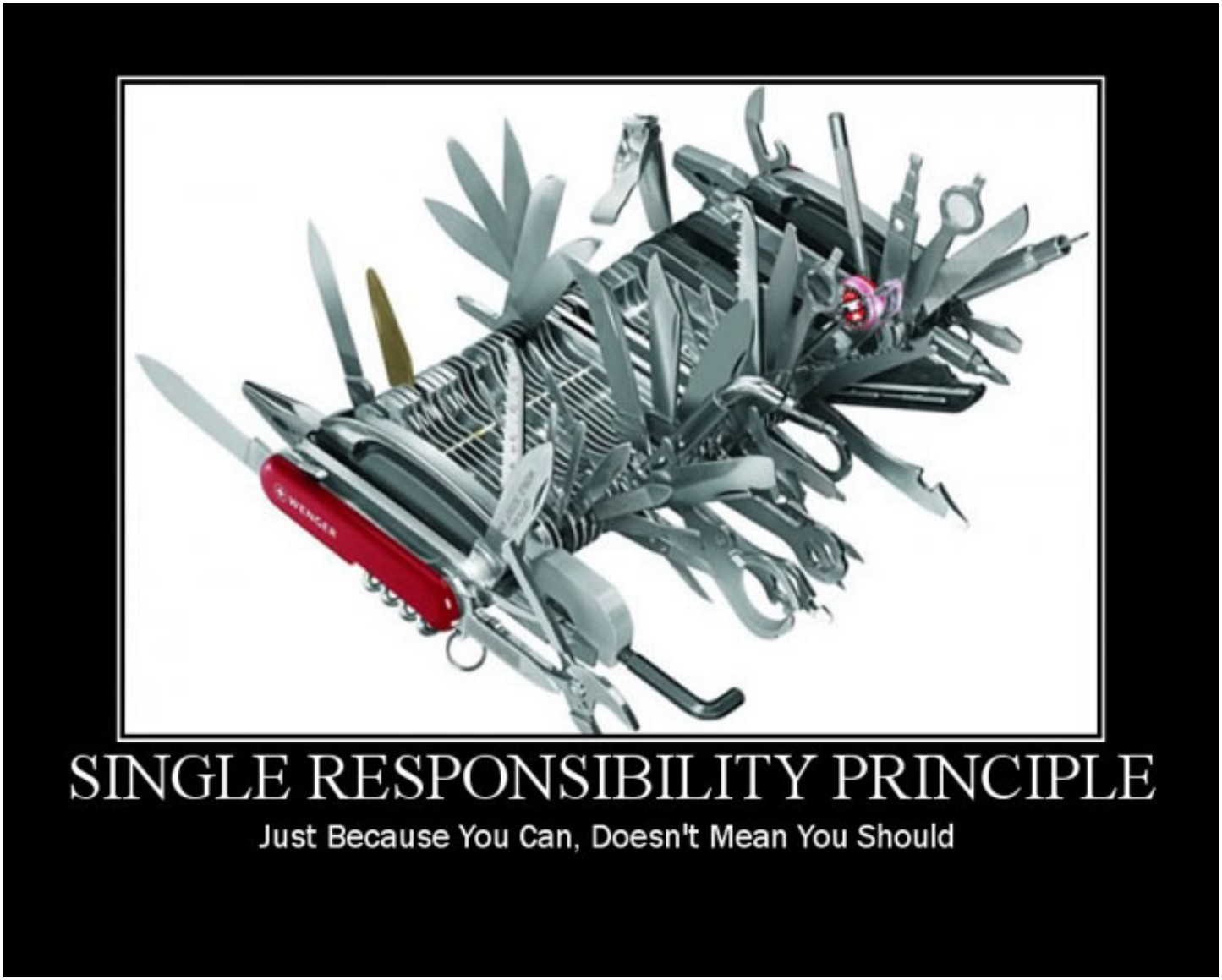
O pen/Closed

L iskov Substitution

I njection of Dependencies

D emeter Principle

SOLID: Responsabilidade Única



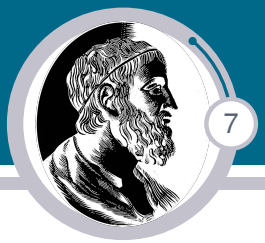
Responsabilidade Única
Definição

PRU - Princípio de Responsabilidade Única

Uma classe deve ter uma, e apenas uma, responsabilidade (isto é, apenas uma razão para mudar)

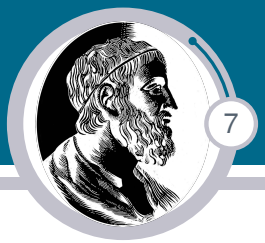
- Uma responsabilidade pode ser descrita com 25 ou menos palavras
- Falta de **coesão** pode indicar uma violação do *PRU*

Responsabilidade Única
Exemplo




```
1  class MasterClass
2    def performInitialization ...; end
3    def readFromFile          ...; end
4    def writeToFile           ...; end
5    def displayToScreen       ...; end
6    def performCalculation    ...; end
7    def validateInput         ...; end
8  end
```

Responsabilidade Única
Exemplo




```
1  class MasterClass
2    def performInitialization ...; end
3    def readFromFile          ...; end
4    def writeToFile           ...; end
5    def displayToScreen       ...; end
6    def performCalculation    ...; end
7    def validateInput         ...; end
8  end
```

MasterClass contém responsabilidades diversas e pouco relacionadas

Responsabilidade Única
Exemplo

Solução: Separar em classes de acordo com a responsabilidade

```
1 class FileInputOutput
2   def readFromFile ...; end
3   def writeToFile  ...; end
4 end
```

```
1 class UserInputOutput
2     def displayToScreen ...; end
3     def validateInput   ...; end
4 end
```

```
1 class Logic
2     def performInitialization ...; end
3     def performCalculation    ...; end
4 end
```

Responsabilidade Única
LCOM

- **LCOM:** *Lack of Cohesion of Methods*
- Analisa a coesão de uma classe, medindo se ela consiste em múltiplos "aglomerados"
- Duas variantes principais:

Variante	Pontuação	Interpretação
Henderson-Sellers	**0** a **1**	Quanto mais próximo de 1, mais variáveis de instância são acessadas por apenas um método
LCOM-4	**1** a **n**	Se n > 1, então n-1 responsabilidades podem ser extraídas para suas próprias classes

Responsabilidade Única
LCOM-4

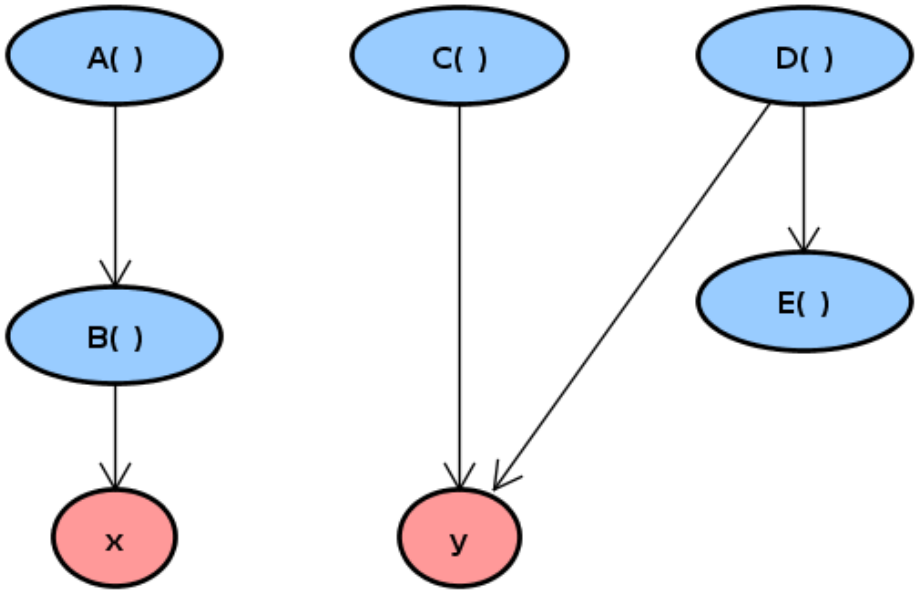
- Dois métodos estão relacionados se:
  - Acessam a mesma variável de instância/classe
  - Um chama o outro

Responsabilidade Única
LCOM-4

- Dois métodos estão relacionados se:
  - Acessam a mesma variável de instância/classe
  - Um chama o outro
- LCOM-4 conecta métodos relacionados em um grafo
- Se o número **n** de grafos resultantes for maior que 1, pode-se dividir em classe em partes menores

Responsabilidade Única
LCOM-4

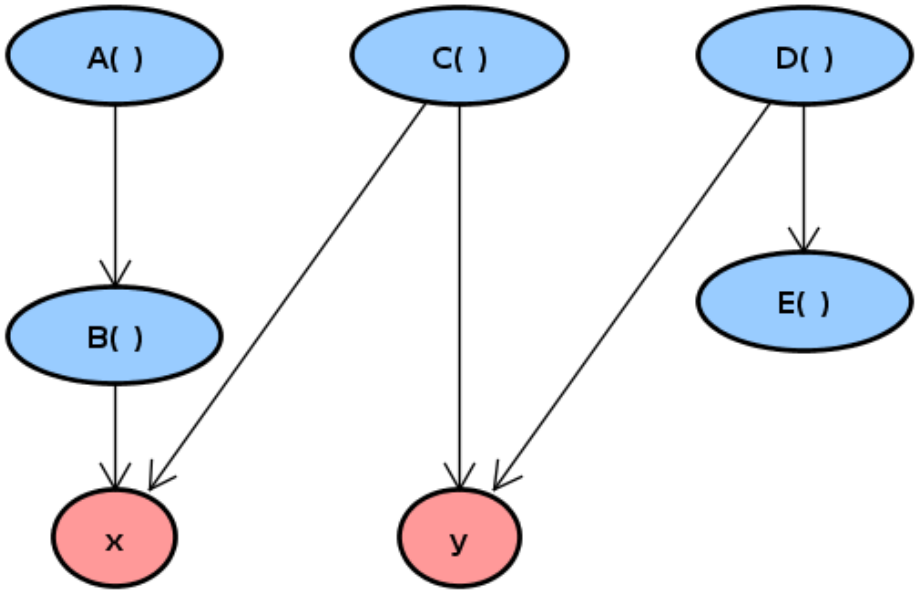
- Dois métodos estão relacionados se:
  - Acessam a mesma variável de instância/classe
  - Um chama o outro
- LCOM-4 conecta métodos relacionados em um grafo
- Se o número **n** de grafos resultantes for maior que 1, pode-se dividir em classe em partes menores



Exemplo 1: LCOM-4 = 2

Responsabilidade Única
LCOM-4

- Dois métodos estão relacionados se:
  - Acessam a mesma variável de instância/classe
  - Um chama o outro
- LCOM-4 conecta métodos relacionados em um grafo
- Se o número **n** de grafos resultantes for maior que 1, pode-se dividir em classe em partes menores



Exemplo 2: LCOM-4 = 1

SOLID: Aberto/Fechado

Aberto/Fechado
Definição

PAF - Princípio Aberto/Fechado

Uma classe deve ser aberta para extensão, mas fechada para modificação

- Deseja-se estender o comportamento de classes sem modificar código existente na qual dependem

Aberto/Fechado
Exemplo


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

Aberto/Fechado
Exemplo

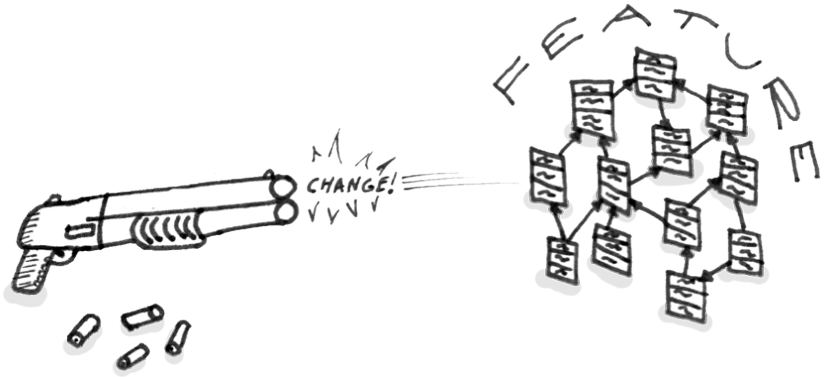

```
class Report
  def output
    formatter =
      case @format
      when :html
        HtmlFormatter.new(self)
      when :pdf
        PdfFormatter.new(self)
      end
  end
end
```

- Cheiro de código *Comando Case*...

Aberto/Fechado
Exemplo


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

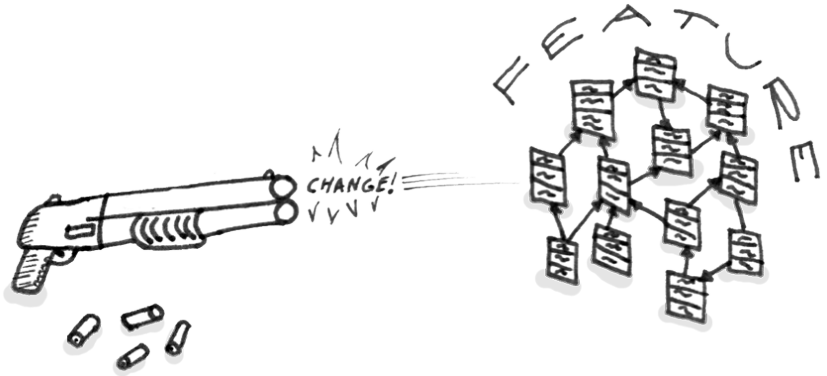
- Cheiro de código *Comando Case*...
- **Cirurgia de Espingarda**: Adicionar um novo tipo de *Formatter* exigirá mudanças em um ou mais métodos



Aberto/Fechado
Exemplo


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

- Cheiro de código *Comando Case*...
- **Cirurgia de Espingarda**: Adicionar um novo tipo de *Formatter* exigirá mudanças em um ou mais métodos

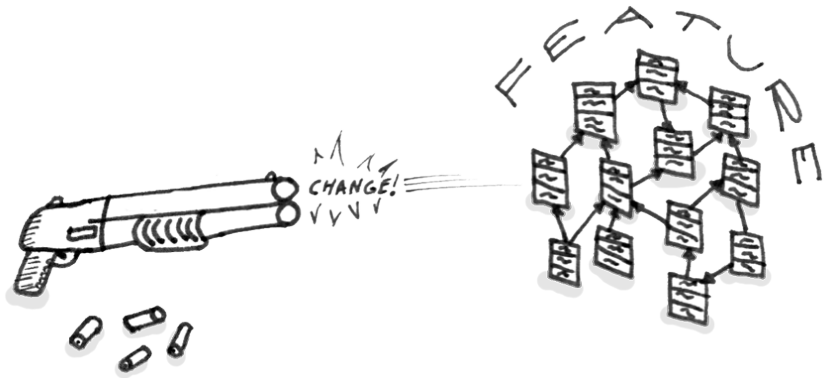


Como resolver?

Aberto/Fechado
Exemplo


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

- Cheiro de código *Comando Case*...
- **Cirurgia de Espingarda**: Adicionar um novo tipo de *Formatter* exigirá mudanças em um ou mais métodos



Como resolver? **Padrões de projeto!**

Aberto/Fechado
Solução com *Fábrica Abstrata*


```ruby
class Report
  def output
    formatter_class =
      begin
        @format.to_s.classify.constantize
      rescue NameError
        # Handle "invalid formatter type"
      end
    formatter = formatter_class.send(:new, self)
  end
end
```

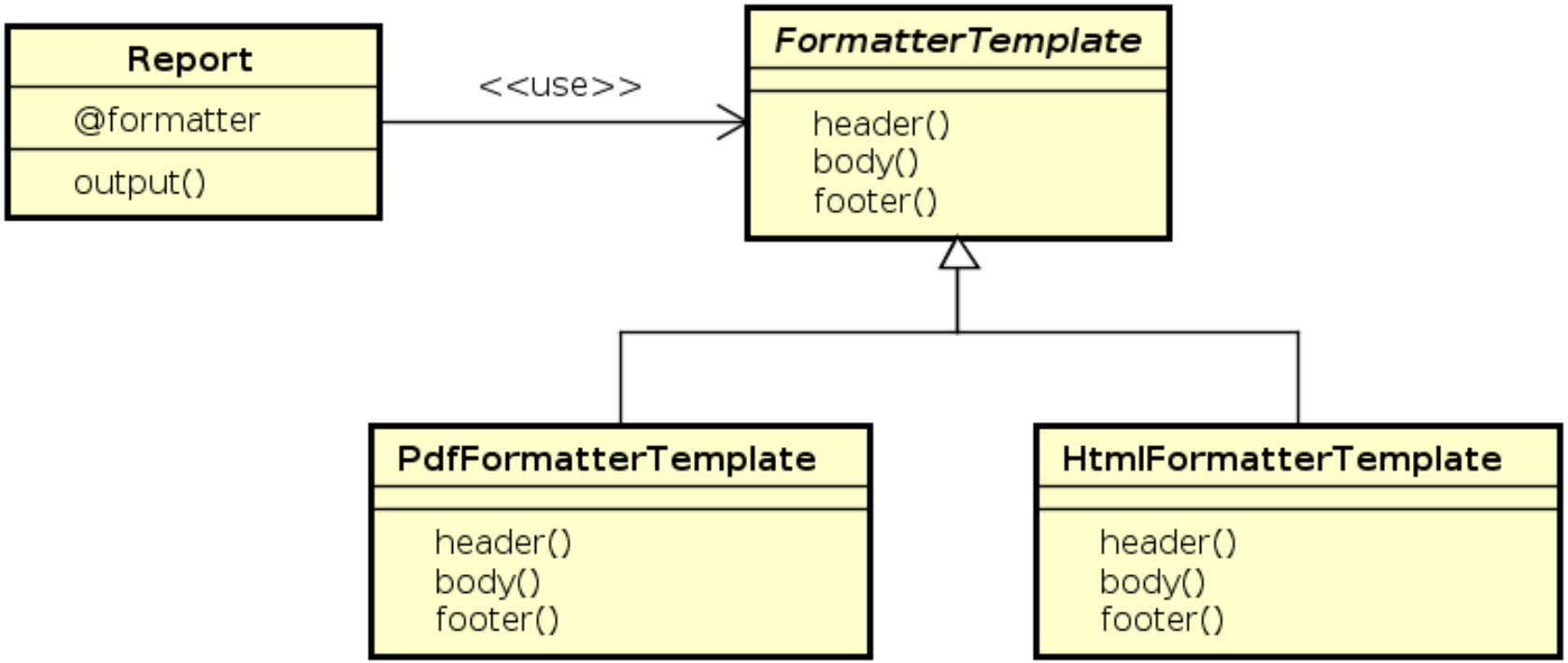
- **@format** se refere a um símbolo (:pdf, :html)

Aberto/Fechado
Solução com *Fábrica Abstrata*


```ruby
class Report
  def output
    formatter_class =
      begin
        @format.to_s.classify.constantize
      rescue NameError
        # Handle "invalid formatter type"
      end
    formatter = formatter_class.send(:new, self)
  end
end
```

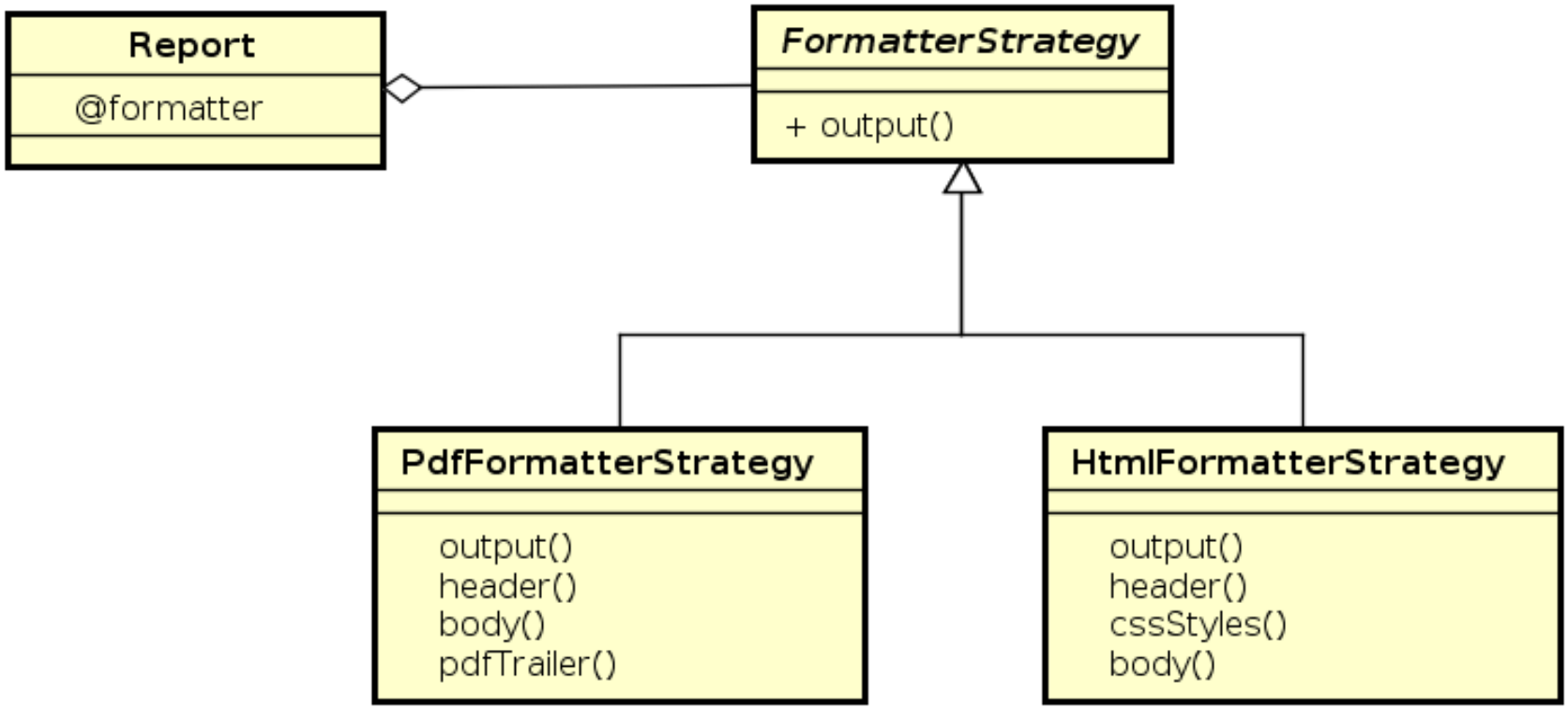
- **@format** se refere a um símbolo (:pdf, :html)
- ***Duck Typing:*** A partir do símbolo, adquirimos uma referência para a classe desejada e chamamos seu construtor

Aberto/Fechado
Solução com *Template*



- Os passos (métodos) da tarefa (formatação) são os mesmos para todas as variantes de *Formatter*
- Implementação dos métodos de cada subclasse podem divergir

Aberto/Fechado
Solução com *Strategy*



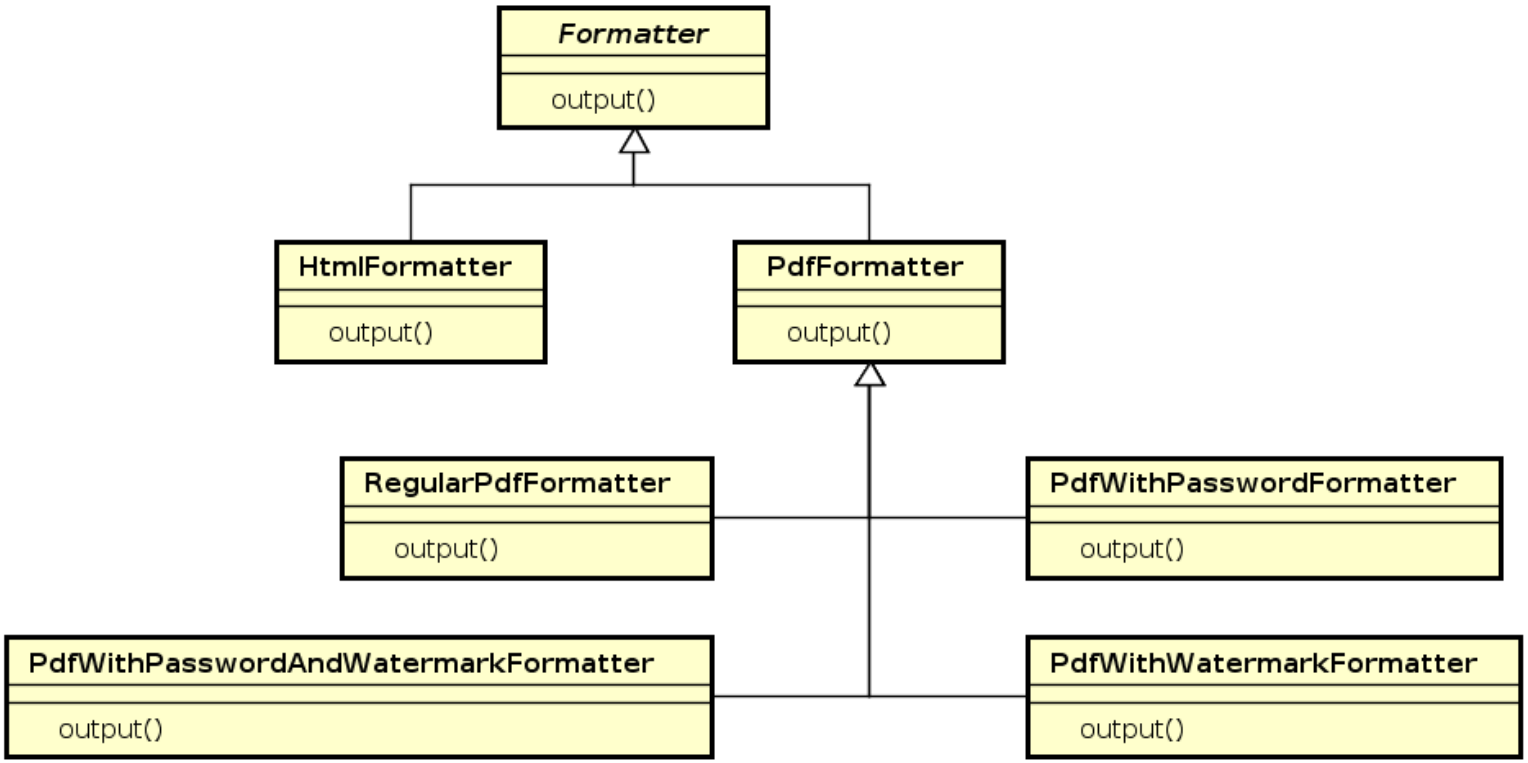
- Usando **Strategy**, os passos podem ser diferentes em cada subclasse

Aberto/Fechado
Queremos Novos Comportamentos!

- E se quisermos adicionar um novo comportamento em uma classe existente? Por exemplo, arquivos PDF:
  - Com/sem proteção de senha
  - Com/sem uma marca d'água "rascunho"

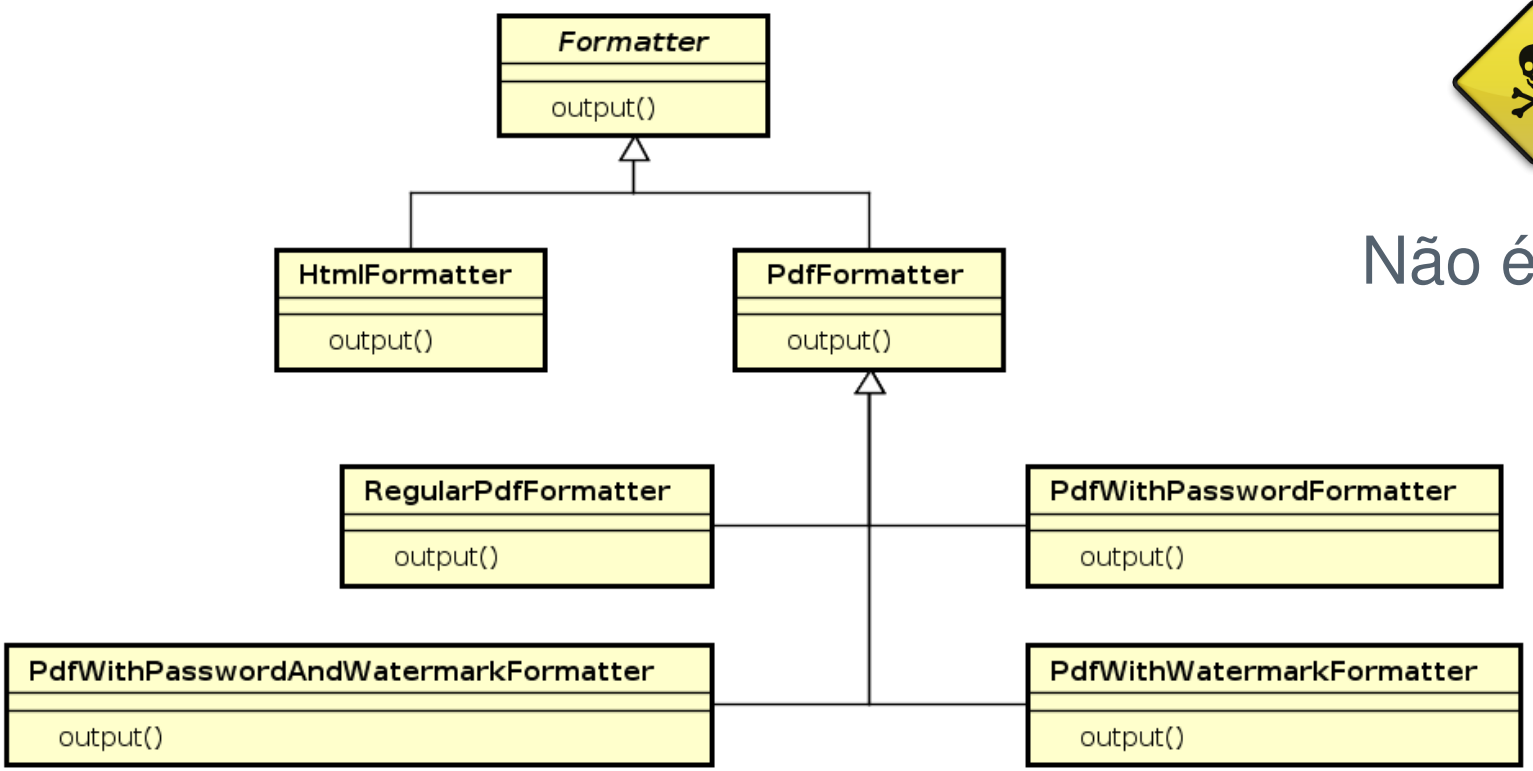
Aberto/Fechado
Queremos Novos Comportamentos!

- E se quisermos adicionar um novo comportamento em uma classe existente? Por exemplo, arquivos PDF:
  - Com/sem proteção de senha
  - Com/sem uma marca d'água "rascunho"
- Uma possível ideia:



Aberto/Fechado
Queremos Novos Comportamentos!

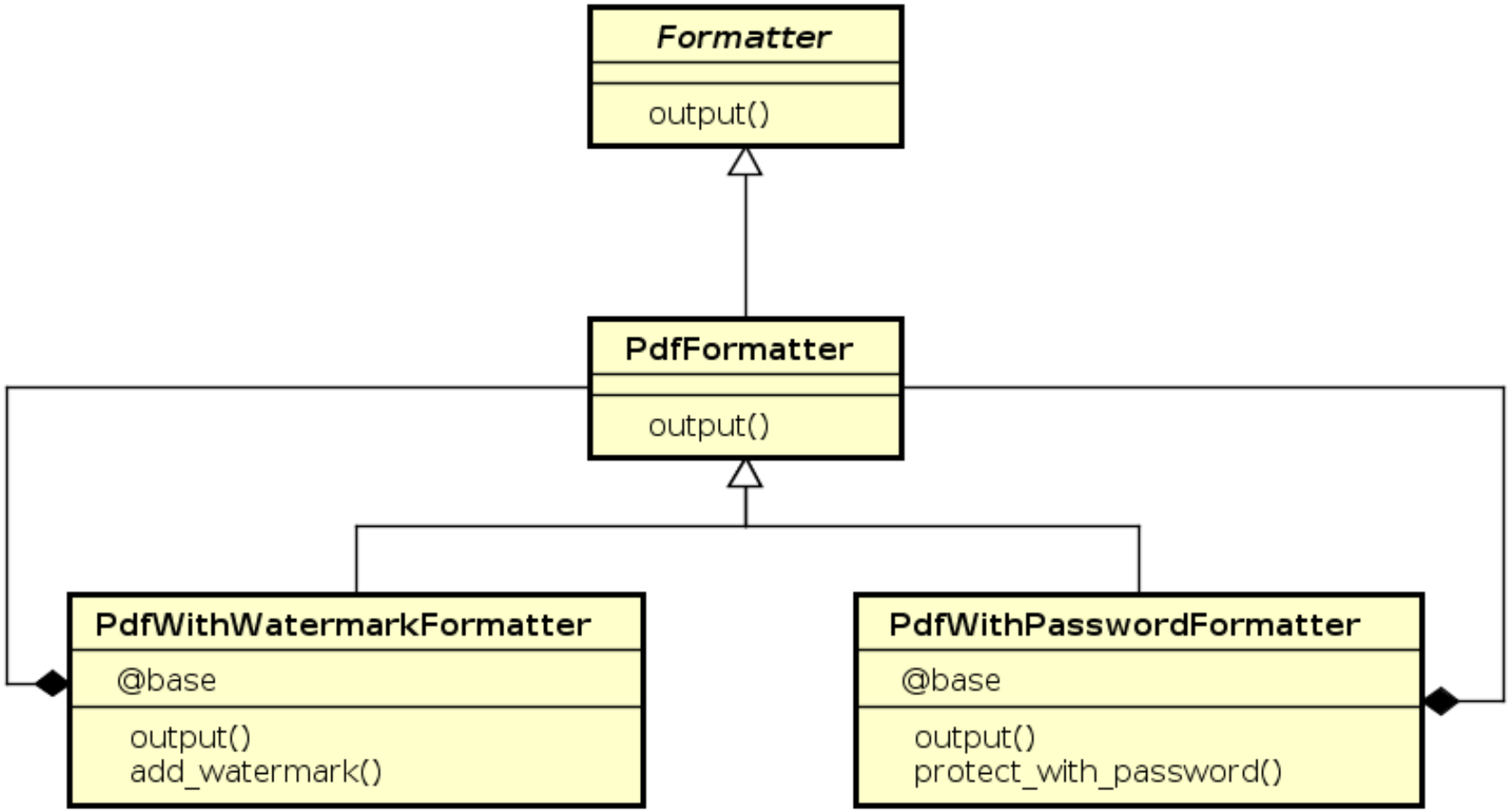
- E se quisermos adicionar um novo comportamento em uma classe existente? Por exemplo, arquivos PDF:
  - Com/sem proteção de senha
  - Com/sem uma marca d'água "rascunho"
- Uma possível ideia:



Não é **DRY**!

Aberto/Fechado
Padrão *Decorator*

Padrão Decorator: "Decoramos" a classe ao envolvê-la em uma versão melhorada, com mesma interface



Código fonte

SOLID: Substituição de Liskov

Substituição de Liskov
Definição

Princípio de Liskov

Um método projetado para trabalhar em um objeto de tipo T deve também trabalhar em um objeto de qualquer subtipo de T

- Herança é muito util para a reutilização de código, mas não é só por isso que ela deve ser usada
- A herança é um compartilhamento de implementação. Se a subclasse não ganha vantagem com a implementação herdada, talvez ela não devesse ser uma subclasse

Substituição de Liskov
Exemplo


```
1  class Retangulo
2    attr_accessor :largura, :altura, :canto_inf_esq
3    def new(largura, altura, canto_inf_esq) ... ; end
4    def area ... ; end
5    def dobrar_altura_sobre_a_largura(dim)
6      self.largura = 2*dim
7      self.altura = dim
8    end
9  end
10 class Quadrado < Retangulo
11   attr_reader :largura, :altura, :lado
12   def largura=(w)  ; @largura = @altura = w ; end
13   def altura=(w)   ; @largura = @altura = w ; end
14   def lado=(w)     ; @largura = @altura = w ; end
15 end
```

Substituição de Liskov
Cheiro

Mau cheiro

- Modificação do funcionamento do método herdado
- Nesse caso, não faz sentido dobrar altura sobre a largura para um quadrado
- Método da superclasse jogado fora

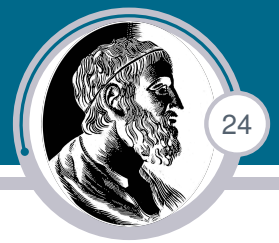
Substituição de Liskov
Refatoração


```
1  class Quadrado
2    attr_accessor :ret
3    def initialize(:lado, :canto_inf_esq)
4      @ret = Retangulo.new(lado, lado, canto_inf_esq)
5    end
6
7    def area
8      ret.area
9    end
10
11   def lado=(s)
12     rect.width = rect.height = s
13   end
14 end
```

Substituição de Liskov
Refatoração


```
1  class Quadrado
2    attr_accessor :ret
3    def initialize(:lado, :canto_inf_esq)
4      @ret = Retangulo.new(lado, lado, canto_inf_esq)
5    end
6
7    def area
8      ret.area
9    end
10
11   def lado=(s)
12     rect.width = rect.height = s
13   end
14 end
```

Trocando herança por composição: implementação é delegada, e não herdada



Delegação em Ruby: **Forwardable**

Módulo em Ruby que implementa a delegação

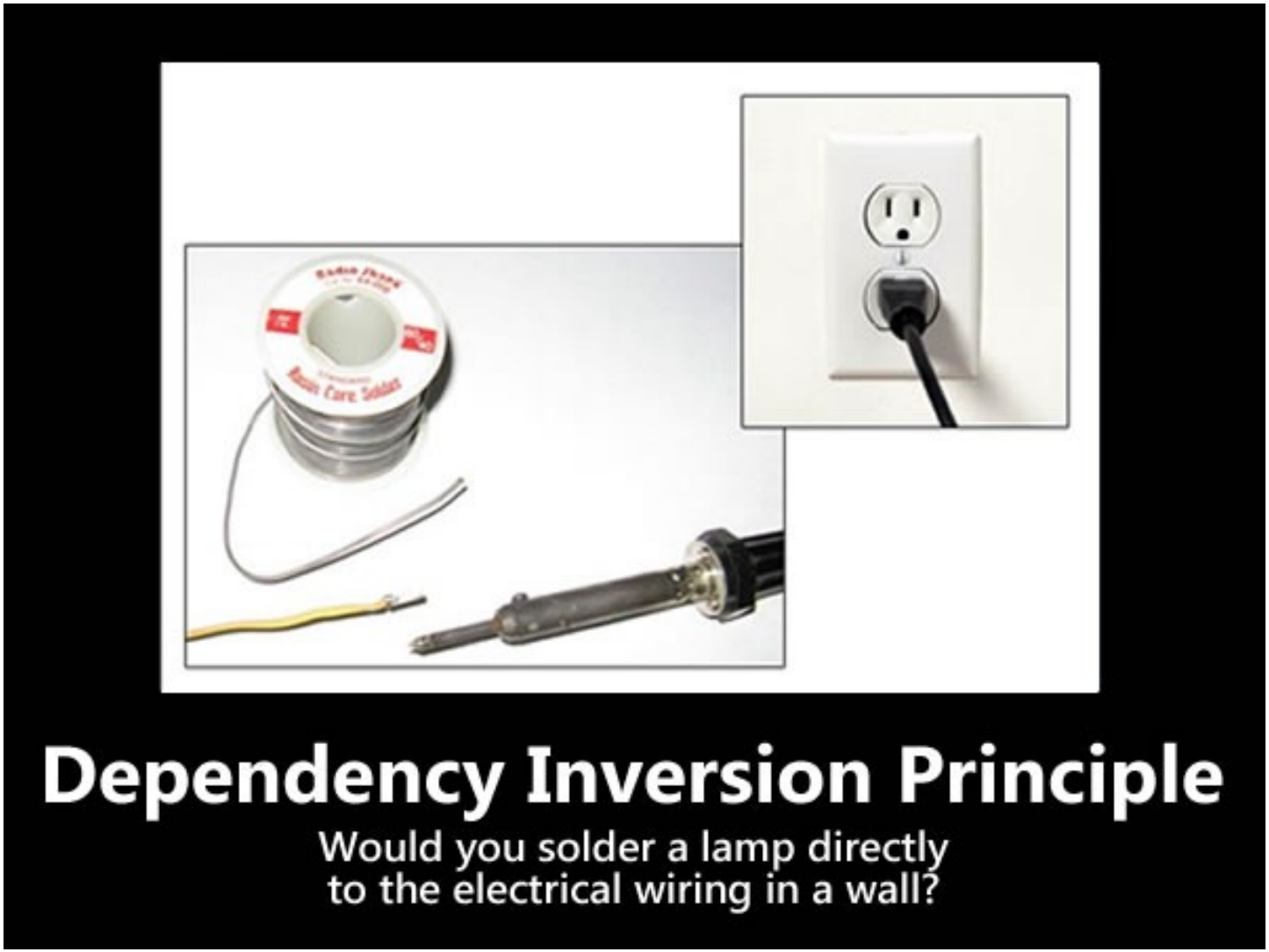
```ruby
class Quadrado
  extend Forwardable
  def_delegators :@ret, :area, :perimetro, :rotacao

  def initialize(:lado, :canto_inf_esq)
    @ret = Retangulo.new(lado, lado, canto_inf_esq)
  end

  def lado=(s)
    @ret.largura = @ret.altura = s
  end
end
```

Substituição de Liskov

Resumo

Toda a implementação da super classe deve fazer sentido para as suas subclasses

SOLID: Injeção de Dependência



Injeção de dependência

- Se duas classes dependem uma da outra, mas suas implementações podem mudar, seria bom para ambas dependerem de uma interface abstrata separada que seja "injetada" entre elas

Injeção de Dependêcia
Definição

Injeção de dependência

- Se duas classes dependem uma da outra, mas suas implementações podem mudar, seria bom para ambas dependerem de uma interface abstrata separada que seja "injetada" entre elas
- O Princípio de Injeção de Dependência (PID) é a criação de uma interface com o objetivo de tratar a manipulação de objetos de forma correta em tempo de execução

Injeção de Dependêcia
Exemplo


```
1  class EmailList
2    attr_reader :mailer
3    delegate :send_email, :to => :mailer
4    def initialize
5      @mailer = MailerMonkey.new
6    end
7  end
8  # in RottenPotatoes EmailListController:
9  def advertise_discount_for_movie
10   moviegoers = Moviegoer.interested_in params[:
       movie_id]
11   EmailList.new.send_email_to moviegoers
12 end
```

Injeção de Dependêcia
Exemplo

- Se quisermos adicionar novas maneiras para enviar mensagens, seria necessário modificar o funcionamento do *controller*, adicionando condicionais
- Nesse caso, entra em ação o Princípio de Injeção de Dependência (PID)!

Injeção de Dependêcia
Exemplo - Refatoração

Controller

```
1 def advertise_discount_for_movie
2   moviegoers = Moviegoer.interested_in(params[:
      movie_id])
3   EmailList.new(Config.emailer).send_email_to(
      moviegoers)
4   end
5 end
```

Config Class

```
1  class Config
2    def self.emailer
3      if email_disabled? then NullMailer else
4        if has_amiko? then AmikoAdapter else
       MailerMonkey end
5      end
6    end
7  end
```

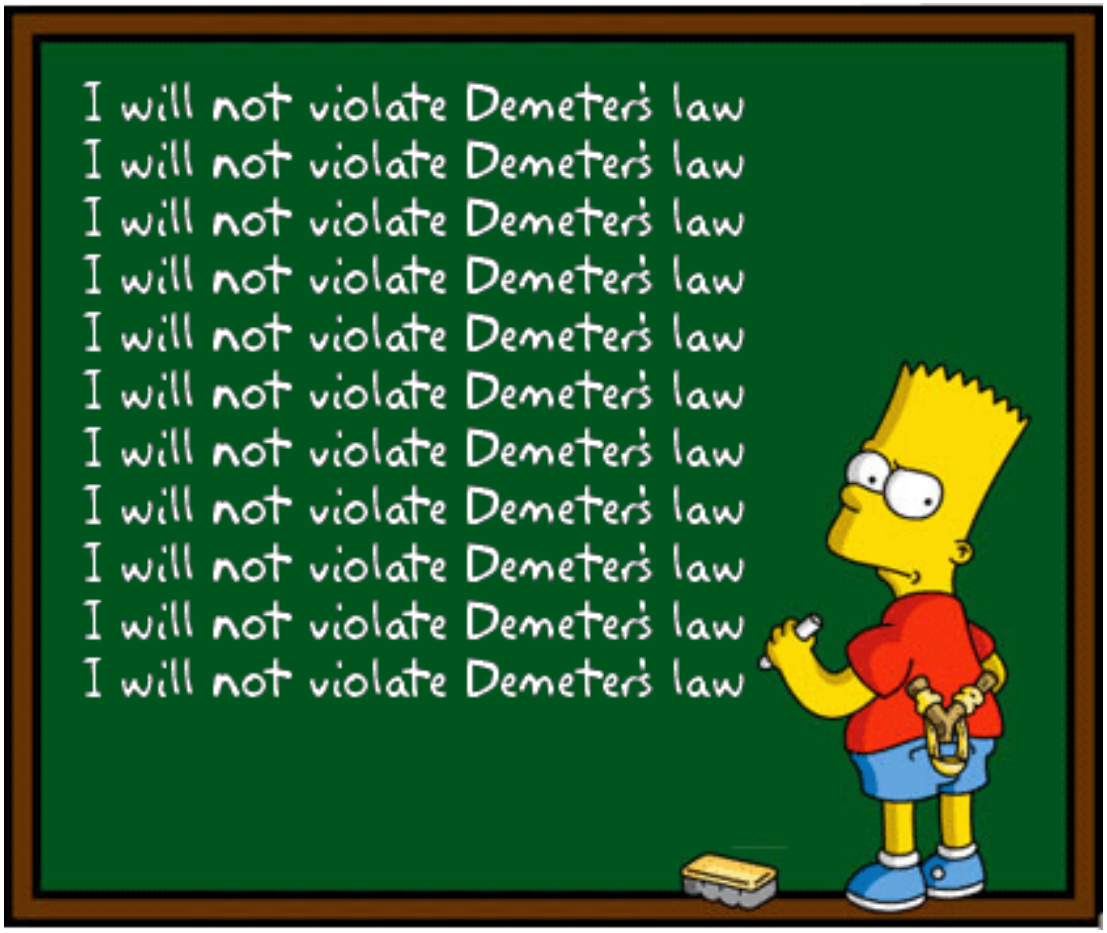
AmikoAdapter Class

```
1  class AmikoAdapter
2    def initialize ; @amiko = Amiko.new(...) ; end
3    def send_email
4      @amiko.authenticate(...)
5      @amiko.send_message(...)
6    end
7  end
```

PID pode ser uma aplicação de um padrão

- *Fachada*
- *Fábrica abstrata:* Cria diferentes tipos de objetos com o mesmo objetivo
- *Adaptador:* Interface que modifica a interação com uma classe com o objetivo de padronizar o acesso (no nosso caso)
- *Proxy:* Mesmo comportamento de outra classe, mas com tratamento para casos "estranhos"

SOLID: Lei de Demeter



Lei de Demeter
Definição

Princípio de Demeter

Um método pode chamar outros métodos em sua própria classe e métodos nas classes de suas próprias variáveis de instância; todo o resto é tabu

Lei de Demeter
Definição

Princípio de Demeter

Um método pode chamar outros métodos em sua própria classe e métodos nas classes de suas próprias variáveis de instância; todo o resto é tabu

Conselhos...

Converse com seus amigos - Não fique íntimo de estranhos

Seus parâmetros

Quando o seu método recebe parâmetros, ele pode chamar algum método fornecido por este parâmetro diretamente

Seus parâmetros

Quando o seu método recebe parâmetros, ele pode chamar algum método fornecido por este parâmetro diretamente

```
1 class JobPost
2   def post_url(id)
3     'http://www.kuniri.org/posts/#{id}'
4   end
5 end
```

Qualquer objeto criado ou instanciado

Quando o seu método cria objetos locais, ele pode chamar métodos nos objetos locais

Qualquer objeto criado ou instanciado

Quando o seu método cria objetos locais, ele pode chamar métodos nos objetos locais

```
1 class Mailer
2   def prepare_emails(list)
3     email_sanitizer = EmailSanitizer.new
4     email_sanitizer.sanitize(list)
5   end
6 end
```

Seu próprio componente de objeto

Seu método pode chamar métodos nos seus próprios campos diretamente (mas não em campos do campo)

Seu próprio componente de objeto

Seu método pode chamar métodos nos seus próprios campos diretamente (mas não em campos do campo)

```
1  class Mailer
2    def initialize(email_sanitizer, list)
3      @email_sanitizer = EmailSanitizer.new
4      @list = list
5    end
6
7    def prepare_emails(list)
8      @email_sanitizer.sanitize(list)
9    end
10 end
```

A si mesmo

Seu método pode chamar outros métodos ou atributos na sua própria classe diretamente

A si mesmo

Seu método pode chamar outros métodos ou atributos na sua própria classe diretamente

```
1 class Address
2   attr_reader :city, :state
3
4   def full_address
5     "#{city}, #{state}"
6   end
7 end
```

Lei de Demeter
Exemplo


```
1
2 def sold_brand_email(order)
3   @order = order
4   mail(
5     to: @order.line_items.last.brand.designer.email,
6     subject: 'You sold a brand!')
7 end
```

Lei de Demeter
Exemplo


```
1
2 def sold_brand_email(order)
3   @order = order
4   mail(
5     to: @order.line_items.last.brand.designer.email,
6     subject: 'You sold a brand!')
7 end
```

Lei de Demeter
Exemplo - Refatorando


```
1 class Mailer
2   def send_mailer(order)
3     mail(
4       to: order.customer.email,
5       subject: "Thanks for purchasing!")
6   end
7 end
```

Lei de Demeter

Exemplo - Aplicando a Lei de Demeter


```
1 class Mailer
2   def send_mailer(order)
3     customer_email = @customer.email
4     mail(
5       to: order.customer_email,
6       subject: "Thanks for purchasing!")
7   end
8 end
```

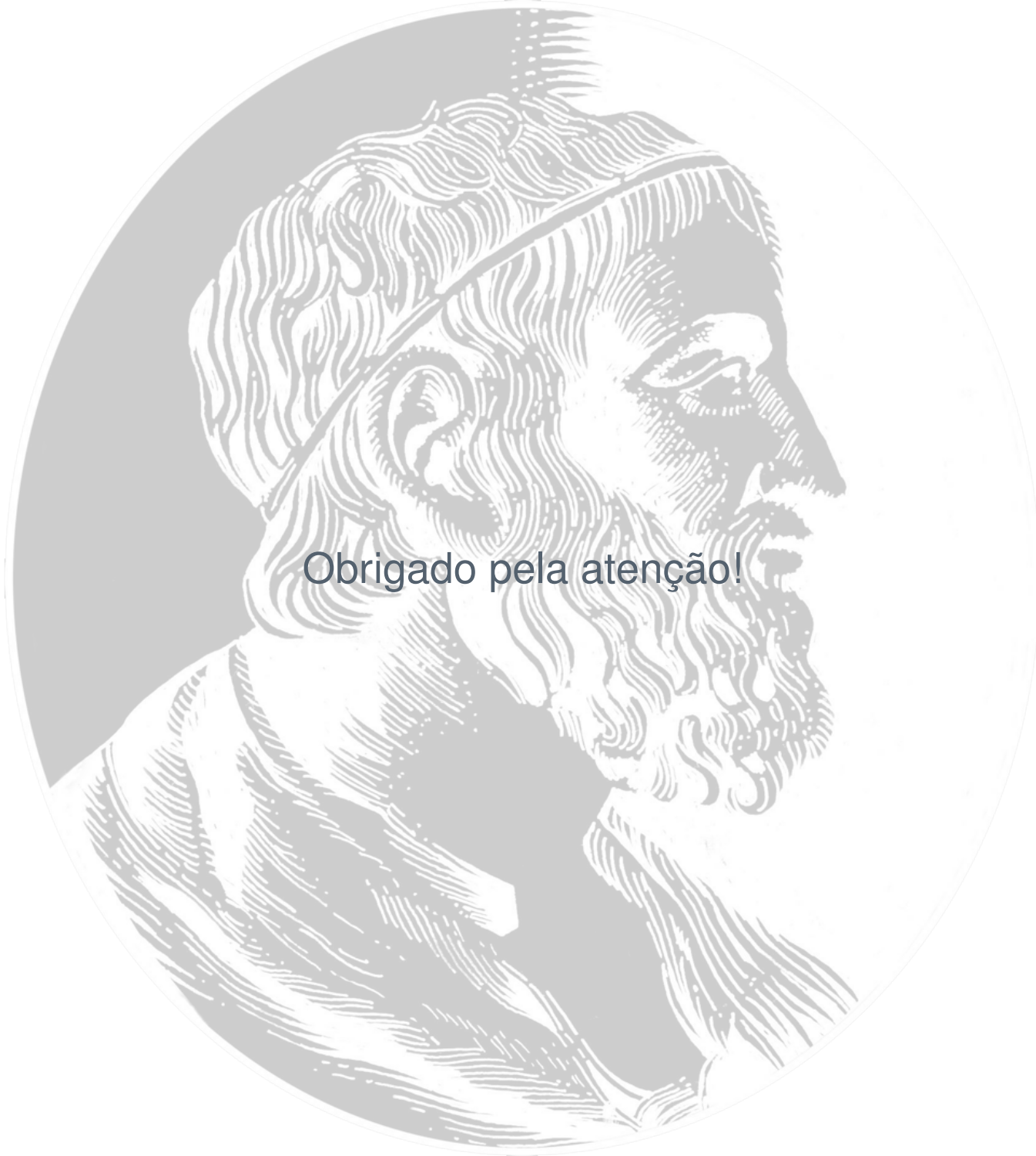
Lei de Demeter

Exemplo - Aplicando a Lei de Demeter


```
1  class Mailer
2    def send_mailer(order)
3      order.send_mailer
4    end
5  end
6
7  class Order
8    def send_mailer
9      mail(
10       to: @customer.email,
11       subject: "Thanks for purchasing!")
12   end
13 end
```

Bibliografia

- Fox, A.; Patterson, D. *Engineering Software as a Service*: An Agile Approach using Cloud Computing. Versão 1.1.1.
- http://rails-bestpractices.com/posts/2010/07/24/the-law-of-demeter/
- https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- http://gespinosa.org/2015/law-of-demeter/



Obrigado pela atenção!