

# Refatorando com padrões de projeto

Um guia em Ruby



Casa do  
Código

MARCOS BRIZENO

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil



**Casa do Código**  
Livros para o programador

**Uma editora de livros técnicos  
feita por desenvolvedores  
para desenvolvedores.**



**Inscreva-se em nossa newsletter e  
receba novidades e lançamentos**

[www.casadocodigo.com.br/newsletter](http://www.casadocodigo.com.br/newsletter)



**Curta nossa fanpage no Facebook**

[www.facebook.com/casadocodigo](http://www.facebook.com/casadocodigo)



**Caelum:  
Cursos de TI presenciais e online**

[www.caelum.com.br](http://www.caelum.com.br)



Dê seu feedback sobre o livro. Escreva para [contato@casadocodigo.com.br](mailto:contato@casadocodigo.com.br)

# Já conhece os nossos títulos?

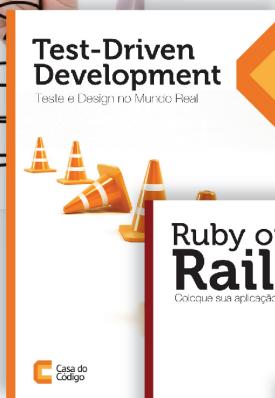


Aplicações Java para web com  
**JSF e JPA**

Casa do Código



Casa do Código



Casa do Código



Casa do Código

**Web Design  
Responsivo**

Páginas adaptáveis para todos os dispositivos



Casa do Código



Casa do Código

**iOS: Programa para  
iPhone e iPad**



Casa do Código

E muito mais em:  
[www.casadocodigo.com.br](http://www.casadocodigo.com.br)

**Casa do Código**  
Livros para o programador

RAFAEL STEIL

# Prefácio

Desenvolvimento de software é uma profissão muito jovem. Entre meados de 1940 (com o computador de Alan Turing) e a década de 2010, são meros 70 anos de história. Para a humanidade, isso não é nada. A maioria da população ainda viveu boa parte de sua vida sem a influência de um computador no seu dia a dia. Tudo é muito novo e muda muito rápido. Ainda não temos um conhecimento aprofundado e difundido do que dá certo e do que não dá.

Padrões de projeto são uma tentativa de estabelecer uma coletânea destes conhecimentos. Eles são apresentados em um formato que segue resumidamente a seguinte estrutura:

- 1) Nome;
- 2) Objetivo;
- 3) Motivação (o problema);
- 4) Contexto;
- 5) Solução;
- 6) Exemplo.

Idealmente, essas coletâneas se tornarão um recurso para programadores identificarem o problema com o qual estão lidando, e aplicarem uma solução que é conhecida e “garantida”. Nesse cenário, ainda falta uma coisa para que possamos usar esse conhecimento de forma segura: como caminhar da situação atual até a solução do padrão, sem introduzir problemas.

Este livro apresenta exemplos práticos em Ruby para seguir essa jornada de forma responsável. Marcos Brizeno apresenta claramente todos os passos para refatorar o código sem causar problemas para os testes automatizados (e, portanto, em seu programa), e chegar a uma implementação de um padrão de projeto conhecido.

Ao passar por 9 padrões documentados no primeiro livro sobre padrões de projeto (Design Patterns) em computação, Marcos não se contenta em explicar a refatoração, nem o padrão. Ele nos guia pelos passos que toma e os motivos pelos quais seria perigoso seguir outro caminho.

O livro é bem fluido, e lê-lo de ponta a ponta não é um problema. Recomendo que, ao final de cada capítulo, você pause a leitura e tente refazer o código daquele capítulo sozinho. A experiência do Marcos no assunto fica óbvia com a clareza dos exemplos e o detalhe dos passos necessários. Se conseguir atingir os mesmos resultados que ele sem quebrar os testes, estará dando um belo passo em direção à excelência técnica.

Caso a leitura de ponta a ponta não seja seu estilo, não se preocupe. Cada capítulo é coeso e independente. Sinta-se à vontade para pular de padrão a padrão, na ordem que lhe convir, desde que pare para tentar seguir os passos você mesmo.

Independentemente do quanto você já conhece de Ruby, de refatoração ou de padrões de projeto, este livro lhe guiará por uma jornada consciente no uso conjunto dessas técnicas em uma linguagem que incorpora vários aspectos de linguagens modernas, como Orientação a Objetos, funções de primeira ordem, e tipagem forte e dinâmica. Mesmo que você não use Ruby no dia a dia, o exercício de refatoração será de muito valor e poderá ser aplicado no seu cotidiano.

*Hugo Corbucci*

## Sobre o autor

Marcos Brizeno é Cientista da Computação pela Universidade Estadual do Ceará e Consultor na ThoughtWorks Brasil. Apaixonado por Engenharia de Software, em especial Metodologias Ágeis, gosta dos desafios de desenvolver software e se apaixonou à primeira vista por Ruby. Publica regularmente em <http://brizeno.wordpress.com>, e gosta de jogar videogames para passar o

tempo e ter novas ideias.

## Agradecimentos

Gostaria de agradecer à minha esposa Ingrid, pela paciência que precisou ter enquanto passava meu tempo escrevendo, e à nossa gata Maggie, que nos ajudou a manter a motivação.

Também gostaria de agradecer aos meus pais, Marcos e Lúcia, que desde cedo incentivaram meu interesse em ler e escrever junto ao meu irmão Brunno.

Consegui realizar este objetivo graças ao esforço e dedicação da minha família.



# Sumário

<b>Introdução</b>	<b>1</b>
<b>1 O que você quer aprender?</b>	<b>3</b>
1.1 O que você vai encontrar . . . . .	3
1.2 Como aproveitar bem o livro . . . . .	4
<b>2 Refatoração e padrões de projeto</b>	<b>7</b>
2.1 O que é refatoração? . . . . .	7
2.2 Técnicas de refatoração . . . . .	10
2.3 O que são padrões de projeto? . . . . .	17
2.4 Refatorando com padrões . . . . .	18
<b>3 Ruby e o paradigma orientado a objetos</b>	<b>21</b>
3.1 Pensando orientado a objetos . . . . .	22
3.2 O que torna Ruby tão especial . . . . .	23
3.3 Outras características da linguagem . . . . .	27
<b>Padrões de projeto comuns</b>	<b>31</b>
<b>4 Factory: gerenciando a criação de objetos</b>	<b>33</b>
4.1 O custo da flexibilidade . . . . .	34
4.2 Os padrões Factory . . . . .	38
<b>5 Strategy: dividir para simplificar</b>	<b>47</b>
5.1 Um login com vários provedores . . . . .	48
5.2 O padrão Strategy . . . . .	53

<b>6 Template Method: definindo algoritmos extensíveis</b>	<b>59</b>
6.1 Nem tão diferentes assim . . . . .	60
6.2 O padrão Template Method . . . . .	63
<b>7 Adapter: seja como a água</b>	<b>69</b>
7.1 Caos e ordem . . . . .	70
7.2 O padrão Adapter . . . . .	73
<b>Padrões de projeto situacionais</b>	<b>79</b>
<b>8 State: 11 estados e 1 objeto</b>	<b>81</b>
8.1 Maria e seus poderes . . . . .	81
8.2 O padrão State . . . . .	85
<b>9 Builder: construir com classe</b>	<b>93</b>
9.1 Muita informação em um só lugar . . . . .	94
9.2 O padrão Builder . . . . .	96
<b>10 Decorator: adicionando características</b>	<b>103</b>
10.1 Espada mágica flamejante da velocidade . . . . .	104
10.2 O padrão Decorator . . . . .	107
<b>11 Mediator: notificações inteligentes</b>	<b>113</b>
11.1 O espaguete de notificações . . . . .	114
11.2 O padrão Mediator . . . . .	116
<b>Conclusão</b>	<b>123</b>
<b>12 Os outros padrões</b>	<b>125</b>
12.1 Padrões pouco utilizados . . . . .	126
12.2 Padrões mal utilizados . . . . .	127
12.3 Padrões que ninguém deveria utilizar . . . . .	128

<b>13 Padrões de projeto e linguagens dinâmicas</b>	<b>131</b>
13.1 Padrões mais simples . . . . .	132
13.2 Padrões invisíveis . . . . .	134
<b>14 Conclusão</b>	<b>139</b>
14.1 Design evolucionário . . . . .	139

Versão: 19.2.7



# **Parte I**

## **Introdução**



## CAPÍTULO 1

# O que você quer aprender?

Geralmente, os autores começam explicando o que será encontrado dentro dos seus livros. Entretanto, antes disso, quero lhe sugerir uma inversão desse pensamento: o que você quer aprender com este livro? O que o levou a escolhê-lo?

O objetivo de responder essas perguntas é dar mais clareza sobre o que é mais importante para você neste momento, aproveitando melhor as partes que mais lhe interessam. Ao colocar em prática os conceitos aqui apresentados, você terá uma nova visão sobre os seus projetos e seus padrões.

### **1.1 O QUE VOCÊ VAI ENCONTRAR**

Este livro vai apresentar, de uma maneira bem prática, 9 dos 23 padrões de projeto catalogados no livro *Design Patterns: elements of reusable object-*

*oriented software*, de Erich Gamma, Ralph Johnson, Richard Helm e John Vlissides (1994).

Quadrados e retângulos para explicar herança e um carro que tem quatro rodas para explicar composição são exemplos simples e bastante usados para explicar conceitos básicos, mas que ficam bem distante da realidade do dia a dia de um desenvolvedor.

A ideia do livro não é ser uma referência para padrões de projeto, mas sim apresentar com exemplos práticos e próximos da realidade como eles podem melhorar sua vida. Vamos explorar bastante o uso dos padrões de projeto, mostrando não só o código da aplicação como também testes!

Cada capítulo começa com um problema e uma amostra de código que é uma solução “rápida”. Depois, serão propostas extensões do problema, que vão expor algumas falhas no design inicial. Isso vai nos forçar a refatorar o código, criar uma situação para aplicar um padrão de projeto e avaliar o resultado final.

Ao refatorar o código, mostraremos quais passos tomar para aplicar as mudanças e também quais modificações de testes serão necessárias. Afinal de contas, mesmo utilizando um padrão bem conhecido e usado em várias situações, é necessário ter segurança ao aplicar mudanças.

Os padrões estão divididos em dois grupos: padrões comuns e padrões situacionais. Padrões comuns são aqueles mais facilmente encontrados em projetos, para o qual talvez você ache uma aplicação assim que ler o capítulo. Já os padrões situacionais não são tão comuns aqui, mas resolvem bem situações específicas, desde que o contexto seja levado em conta.

## 1.2 COMO APROVEITAR BEM O LIVRO

O primeiro passo é participar da lista de discussão do grupo! Acesse <http://forum.casadocodigo.com.br/> e assine as discussões por e-mail.

Lá poderemos conversar e trocar ideias diretamente, além de falar também com outras pessoas que também estão lendo o livro.

Padrões de projeto são um tópico avançado de Orientação a Objetos (OO); logo, é necessário ter um bom conhecimento sobre OO. É esperado que você entenda os relacionamentos entre objetos, interfaces, contratos etc.

Se você já tem experiência com isso, então os problemas de design ficarão bem claros, o que facilita entender a motivação por trás dos padrões.

Os exemplos e as discussões serão feitos considerando sua aplicação em Ruby, por isso é necessário ter conhecimento razoável sobre a linguagem. Não é preciso conhecer todas as suas funcionalidades; os exemplos são simples e curtos. Se você já conhece bem Ruby, talvez ache pontos em que o código poderia ser simplificado ou diminuído utilizando algumas funcionalidades da linguagem, mas esse não é o foco do livro.

Além da linguagem Ruby, também vamos usar bastante *RSpec* (<http://rspec.info>) , uma ferramenta para escrever testes automatizados. Se você não tem experiência com *RSpec*, não se preocupe! Os testes serão bem simples e foram escritos pensando em você.

Um típico teste escrito em *RSpec* começa com uma chamada ao método `it`, passando uma *string* e um bloco como parâmetro. O bloco, então, será executado e, ao final, as condições internas serão validadas para saber se o teste passa ou não.

No exemplo a seguir, usamos o método `expect` para avaliar que o retorno da chamada `criterio.por_pagina` é igual a `20`.

```
it 'retorna resultado por página quando especificado' do
  parametros = {
    produto: 'produto qualquer',
    resultados_por_pagina: 20
  }
  criterio = Busca.criar_criterio(parametros)
  expect(criterio.por_pagina).to eq(20)
end
```

Devido ao *syntactic sugar* de Ruby, podemos encadear as chamadas do método `expect()`.`to` `eq()` com um espaço entre elas. Esse código é equivalente a `expect().to(eq())`, a diferença é que podemos deixar um espaço entre o método e seus parâmetros.

É comum ver testes escritos assim, pois a separação entre o que deve ser chamado e o que é esperado fica visualmente mais clara.

Outro ponto importante a se notar é que cada teste é organizado em três partes, seguindo o padrão AAA (<http://c2.com/cgi/wiki?ArrangeActAssert>) :

- 1) **Arranjo:** onde os dados do teste são preparados;
- 2) **Ação:** onde a ação alvo do teste é chamada;
- 3) **Asserção:** onde o resultado da ação sobre os dados é validada.

Se em algum momento surgirem dúvidas sobre alguma parte do código do teste, basta ir à página do projeto *RSpec* para encontrar toda a informação que precisa.

Todos os exemplos de código utilizados no livro estão disponíveis em um repositório no GitHub: <https://github.com/MarcosX/rppr>. Se você já conhece Git e GitHub, basta clonar o repositório. Se você não conhece essas ferramentas é possível fazer o download do código na página do repositório. No *branch* principal (`master`) você encontrará os códigos iniciais, antes da refatoração, e no *branch* `refatorado` estarão os exemplos após a refatoração. Assim você pode comparar os códigos sempre que precisar.

Uma vez que você fez uma cópia do repositório, para instalar as dependências basta utilizar uma outra ferramenta chamada *Bundler* (<http://bundler.io>). Com ela instalada, basta executar `bundle install` na pasta do repositório que todas as *gems* necessárias estarão disponíveis.

## CAPÍTULO 2

# Refatoração e padrões de projeto

Neste capítulo, serão brevemente apresentados os conceitos de refatoração e padrões de projeto, e como eles podem ajudar no seu dia a dia. Mesmo que você seja experiente e conheça os conceitos, recomendo a leitura deste capítulo, pois os dois já estão por aí há bastante tempo, e cada pessoa tem um entendimento próprio do que eles são ou não.

## 2.1 O QUE É REFATORAÇÃO?

Refatoração já não é mais novidade, o livro *Refactoring: improving the design of existing code*, de Martin Fowler (1999), já possui mais de 15 anos de existência. Mesmo assim, o assunto continua sendo relevante e pertinente tanto para novos profissionais como para quem nunca parou para entender o que é e o que não é refatoração.

A definição de refatoração por Martin Fowler pode ser entendida em 3 partes:

- **Melhorar o design existente:**

Melhorar o design existente é uma definição muito importante, pois é bem comum pensar que qualquer coisa que não seja adição de funcionalidade é refatoração. Correções não devem fazer parte do processo de refatoração, lembre-se de que o foco é melhorar o design! Por isso é melhor reduzir a quantidade de mudanças, o que nos leva à segunda parte da definição.

- **Aplicar mudanças em pequenos passos:**

O conceito de aplicar mudanças em pequenos passos não é muito objetivo, afinal, o que são “pequenos passos”? O objetivo principal é reduzir a quantidade de mudanças, pois menos mudanças é igual a menos problemas! Se você está alterando os parâmetros de um método e percebe que algo na lógica poderia ser melhor, tome nota e volte para fazer a mudança depois.

- **Evitar deixar o sistema quebrado:**

Essa terceira parte da definição é sobre evitar deixar o sistema quebrado, garantindo que as melhorias não acabem afetando funcionalidades existentes. Para isso, é importante ter uma boa suíte de testes, tanto com relação à cobertura quanto à facilidade e velocidade de execução. Além de executar frequentemente os testes da classe em mudança, também é importante garantir que a integração com o resto do sistema continue funcionando.

Um exemplo bem simples de refatoração seria renomear o método para que o código fique mais expressivo. Note o comentário explicando o que o método faz para facilitar o entendimento do código:

```
# metodo que calcula o preco final
def calcula(preco_base, modificador_de_porcentagem)
  if modificador_de_porcentagem.nil? ||
    modificador_de_porcentagem < 10
```

```
    preco_base
else
  preco_base * (modificador_de_porcentagem/100)
end
end
```

Um exemplo de teste seria:

```
it 'calcula o preço final com modificador de 10%', do
  expect(calcula(100,20)).to eq(120)
end
```

No entanto, se mudarmos o nome do método de uma vez, todos os testes quebrariam, pois estariam usando o método antigo. Criamos, então, um novo método que chama o anterior:

```
def preco_final(preco_base, modificador_de_porcentagem)
  calcula(preco_base, modificador_de_porcentagem)
end
```

Agora, podemos atualizar os testes chamando o novo método:

```
it 'calcula o preço final com modificador de 10%', do
  expect(preco_final(100,10)).to eq(110)
end
```

Por fim, atualizamos o nome do método e apagamos o antigo:

```
# metodo que calcula o preço final
def preco_final(preco_base, modificador_de_porcentagem)
  if modificador_de_porcentagem.nil? ||
    modificador_de_porcentagem < 10
    preco_base
  else
    preco_base * (modificador_de_porcentagem/100)
  end
end
```

Apesar dos vários passos para simplesmente renomear um método e deixar o código mais expressivo, ao longo de todas as mudanças, os testes estavam sempre passando. Uma outra mudança que poderia ser feita era extrair

a lógica do `if`, mas ela não é necessária para renomear o método, então deixamos para depois.

## 2.2 TÉCNICAS DE REFATORAÇÃO

Ao aplicar refatorações, existem várias técnicas que podem ser aplicadas para alcançar a melhoria desejada. O exemplo de refatoração descrita na seção anterior é chamado de **Renomear Método**. As técnicas descrevem passos que podem ser tomados garantindo que, a cada mudança, todos os testes continuem passando.

Martin Fowler descreve no livro *Refactoring* um conjunto de técnicas que podem ser utilizadas em várias situações. Ao longo do livro, vamos usar algumas delas para modificar o código e aplicar os padrões.

### Extrair Método

Extrair Método é uma técnica bem simples e poderosa, utilizada quando precisamos quebrar um método que possui mais de uma responsabilidade. O método a seguir, apesar de se chamar `desativar_usuarios`, faz muito mais do que isso.

Primeiro, ele busca os usuários para desativar, para depois executar a ação de desativá-los e, por fim, notificar os usuários por e-mail.

```
class DesativarUsuariosWorker
  def desativar_usuarios
    desativar_usuarios =
      Usuarios.all.select do |usuario|
        usuario.ultimo_login > 1.month.ago && usuario.ativo?
      end
    usuarios_para_desativar.each(&:deactivate)
    NotificarUsuarioViaEmail.desativados(desativar_usuarios)
  end
end
```

Vamos usar Extrair Método para retirar a responsabilidade de buscar os usuários. O primeiro passo é criar o novo método para representar a ideia

que estamos extraíndo e, em seguida, copiar o código, duplicando-o por enquanto.

```
class DesativarUsuariosWorker
  def usuarios_para_desativar
    Usuarios.all.select do |usuario|
      usuario.ultimo_login > 1.month.ago && usuario.ativo?
    end
  end
end
```

Ao extrair um método, é preciso estar atento às variáveis locais, pois elas também precisam existir dentro do novo método. Podemos apenas copiar as variáveis locais, passá-las como argumento ou extraí-las para seu próprio método.

Outro ponto para manter em mente é também mover testes que sejam específicos ao novo método. Nem sempre esse será o caso, mas, geralmente, quando um método possui mais de uma responsabilidade, ele também vai ter vários testes unitários. Ao dividi-lo, vale a pena atualizar os testes para garantir que o novo método se comporte da mesma forma que o anterior.

Uma vez que o novo método já está criado, todo o seu escopo está funcional e possui testes unitários. O próximo passo é substituir o código pelo novo método.

```
class DesativarUsuariosWorker
  def desativar_usuarios
    usuarios = usuarios_para_desativar
    usuarios.each(&:deactivate)
    NotificarUsuarioViaEmail.desativados(usuarios)
  end
end
```

## Mover Método

Mover Método pode ser usado quando temos um método que utiliza mais informações de outra classe do que da sua própria. Assim, reduzimos a complexidade do código, pois ele vai ter acesso a todas as informações locais da nova classe em vez de ficar perguntando antes de tomar ações.

Voltando para o exemplo anterior, o método `desativar_usuarios` da classe `DesativarUsuariosWorker`, apesar de ter sido quebrado em mais de um método, ainda possui muito código relacionado a usuários. Quando o nome de outro objeto aparece muitas vezes, é um sinal de que talvez a responsabilidade esteja no lugar errado.

```
class DesativarUsuariosWorker
  def desativar_usuarios
    usuarios = usuarios_para_desativar
    usuarios.each(&:deactivate)
    NotificarUsuarioViaEmail.desativados(usuarios)
  end

  def usuarios_para_desativar
    Usuarios.all.select do |usuario|
      usuario.ultimo_login > 1.month.ago && usuario.ativo?
    end
  end
end
```

Vamos usar Mover Método para que `usuarios_para_desativar` seja responsabilidade da classe `Usuarios`. O primeiro passo é duplicar o código, sem alterar nenhuma funcionalidade.

```
class Usuario
  def self.usuarios_para_desativar
    Usuarios.all.select do |usuario|
      usuario.ultimo_login > 1.month.ago && usuario.ativo?
    end
  end
end
```

Como o método vai faz parte de outra classe, também é necessário duplicar os seus testes unitários para garantir que ele continue funcionando como esperado. Após duplicar o método, basta substituir sua chamada e executar todos os testes novamente.

```
class DesativarUsuariosWorker
  def desativar_usuarios
```

```
    usuarios = Usuario.usuarios_para_desativar
    usuarios.each(&:deactivate)
    NotificarUsuarioViaEmail.desativados(usuarios)
  end
end
```

## Mover Campo

Semelhante ao **Mover Método**, Mover Campo é útil quando temos um atributo que é mais utilizado em outra classe do que em sua própria. O benefício principal é que, ao ser definido na nova classe, garantimos que ele fique protegido de modificações externas.

No exemplo a seguir, temos a classe `CalculadorDePreco`, que possui a responsabilidade de calcular o preço final da corrida, e a classe `Taxi`, que representa um táxi com suas informações. Apesar de as constantes `BANDEIRA_UM` e `BANDEIRA_DOIS` estarem definidas no `Taxi`, elas são realmente usadas no `CalculadorDePreco`.

```
class CalculadorDePreco
  VALOR_POR_KM = 0.48
  def self.calcular_corrida(km_rodados, bandeira)
    bandeira * (km_rodados * VALOR_POR_KM)
  end
end

class Taxi
  BANDEIRA_UM = 1.2
  BANDEIRA_DOIS = 1.8

  def calcular_corrida(km_rodados)
    if (dia_de_semana?)
      calcular_corrida(km_rodados, BANDEIRA_UM)
    else
      calcular_correida(km_rodados, BANDEIRA_DOIS)
    end
  end
end
```

O primeiro passo é copiar os atributos na nova classe e, em seguida,

referenciá-los na classe antiga:

```
class CalculadorDePreco
  VALOR_POR_KM = 0.48
  BANDEIRA_UM = 1.2
  BANDEIRA_DOIS = 1.8

  def self.calcular_carrada(km_rodados, bandeira)
    bandeira * (km_rodados * VALOR_POR_KM)
  end
end

class Taxi
  def calcular_carrada(km_rodados)
    if (dia_de_semana?)
      calcular_carrada(km_rodados,
                        CalculadorDePreco::BANDEIRA_UM)
    else
      calcular_carrada(km_rodados,
                        CalculadorDePreco::BANDEIRA_DOIS)
    end
  end
end
```

Uma vez que os atributos só são usados na nova classe, o próximo passo é encontrar uma maneira de removê-los da classe antiga. No nosso código, vamos tirar o parâmetro `bandeira` e passar o `dia_da_semana?`. Assim, a lógica de verificação fica completamente no método `CalculadorDePreco.calcular_carrada`.

```
class CalculadorDePreco
  VALOR_POR_KM = 0.48
  BANDEIRA_UM = 1.2
  BANDEIRA_DOIS = 1.8

  def self.calcular_carrada(km_rodados, dia_de_semana)
    if dia_de_semana
      BANDEIRA_UM * (km_rodados * VALOR_POR_KM)
    else
```

```
    BANDEIRA_DOIS * (km_rodados * VALOR POR KM)
end
end
end

class Taxi
  def calcular_corrida(km_rodados)
    calcular_corrida(km_rodados, dia_de_semana?)
  end
end
```

Agora, a lógica que antes estava divida entre as classes ficou concentrada apenas no `CalculadorDePreco`.

## Extrair Classe

Extrair Classe pode ser entendido como uma evolução de **Extrair Método**. Usamos essa técnica quando uma classe possui mais de uma responsabilidade. Para construir a nova classe, vamos utilizar as técnicas **Mover Campo** e **Mover Método**.

O código a seguir mostra o `BaixarRegistrosDeVendaFtpNoBancoWorker`, que tem como responsabilidades baixar arquivos de um servidor `ftp` e salvá-lo no banco de dados. A classe mistura tanto informações sobre o servidor de FTP, como `host`, `usuario` etc., bem como informações sobre qual tabela do banco deve ser atualizada.

```
class BaixarRegistrosDeVendaFtpNoBancoWorker
  attr_reader :host, :porta, :usuario, :senha

  def self.requisitar_ftp(arquivo)
    Net::FTP.open(@host) do |ftp|
      ftp.login(@usuario, @senha)
      salvar_arquivo_no_banco(ftp.gettextfile(arquivo))
    end
  end

  def salvar_arquivo_no_banco(arquivo)
    RegistroDeVendas.ler_de_arquivo(arquivo)
```

```
end  
end
```

Para reduzir a complexidade, vamos separar a responsabilidade de baixar arquivos de um servidor FTP da parte que os salva no banco. Para isso, o primeiro passo é criar a classe que vai conter a nova responsabilidade e copiar os métodos para ela, bem como os testes específicos desse comportamento.

```
class BaixarArquivosFtp  
  attr_reader :host, :porta, :usuario, :senha  
  
  def self.requisitar_ftp(arquivo)  
    Net::FTP.open(@host) do |ftp|  
      ftp.login(@usuario, @senha)  
      ftp.gettextfile(arquivo)  
    end  
  end  
end
```

Em seguida, vamos substituir o código antigo pela nova classe. No nosso exemplo, vamos apenas chamar `BaixarArquivosFtp.requisitar_ftp`, e passar o arquivo recebido para `salvar_arquivo_no_banco`.

```
class BaixarRegistrosDeVendaFtpNoBancoWorker  
  def self.requisitar_ftp(arquivo)  
    arquivo_texto = BaixarArquivosFtp.requisitar_ftp(arquivo)  
    salvar_arquivo_no_banco(arquivo_texto)  
  end  
  
  def salvar_arquivo_no_banco(arquivo)  
    RegistroDeVendas.ler_de_arquivo(arquivo)  
  end  
end
```

Agora, com os métodos simplificados e as responsabilidades separadas, fica até mais fácil encontrar nomes mais sugestivos para as classes, deixando claro o que cada uma faz. A classe `BaixarArquivosFtp` é genérica o suficiente para ser usada por qualquer outro *worker*. Já a classe

BaixarRegistrosDeVendaFtpNoBancoWorker pode ser nomeada para apenas SalvarRegistroDeVendasWorker.

```
class SalvarRegistroDeVendasWorker
  def salvar_arquivo_no_banco()
    arquivos = BaixarArquivosFtp.requisitar_ftp(arquivo)
    RegistroDeVendas.ler_de_arquivo(arquivo)
  end
end

class BaixarArquivosFtp
  attr_reader :host, :porta, :usuario, :senha

  def self.requisitar_ftp(arquivo)
    Net::FTP.open(@host) do |ftp|
      ftp.login(user: @usuario, passwd: @senha)
      ftp.gettextfile(arquivo)
    end
  end
end
```

## 2.3 O QUE SÃO PADRÕES DE PROJETO?

Assim como boa parte da área de Computação, padrões de projeto também vieram da Engenharia, neste caso mais especificamente da Arquitetura. Portas geralmente são colocadas nos cantos da parede para que elas não ocupem muito espaço ao abrir, certo? Isso é um exemplo de padrão arquitetural.

Uma definição de padrão de projeto que gosto bastante também é dividida em 3 partes: **Uma solução comum para um problema em um determinado contexto.**

Voltando ao problema da porta, posicioná-la no canto é uma solução bem comum. Mas e o contexto?

Imagine que agora a porta é na entrada de uma agência bancária. Ela seria giratória e geralmente é posicionada no centro da parede. Aplicar o padrão de posicionamento da porta no canto não faz muito sentido aqui, embora ainda possa ser utilizado. É bastante comum ver aplicações de padrões de projeto que não fazem muito sentido porque o contexto foi totalmente ignorado.

Dizer que um padrão é uma solução comum implica que nenhum padrão é “criado”, mas sim documentado. Um bom exemplo é o livro da Gangue dos Quatro (*Gang of Four*), *Design Patterns: elements of reusable object-oriented software* – comentado no capítulo anterior –, no qual os 23 padrões documentados não são criações dos autores; eles vieram de observações e generalizações de situações comuns.

Para que exista uma solução, é preciso que primeiro se tenha o problema. Todos os padrões tentam resolver algum tipo específico de problema, e é assim que eles são classificados.

Voltando ao livro da Gangue dos Quatro, os padrões são categorizados como de:

- **Criação** – problemas que envolvem criar objetos;
- **Estruturais** – problemas com a arquitetura da aplicação;
- **Comportamentais** – problemas com o estado interno e o comportamento de objetos.

Por fim, mas não menos importante, o contexto do problema é o fator principal ao decidir aplicar ou não um padrão! Ao longo do livro, vamos explorar exemplos e discutir bastante o contexto da situação.

Aplicar um padrão simplesmente por aplicar pode resultar em um design muito mais complexo que o necessário e montes de código que ninguém consegue entender sem ter de pular entre várias classes.

## 2.4 REFATORANDO COM PADRÕES

Como uma solução comum para um problema existente, dentro de um contexto, pode ajudar a melhorar o design em pequenos passos sem deixar o sistema quebrado?

Uma vantagem bem óbvia de utilizar padrões de projeto é que eles facilitam a comunicação. Se você e uma colega entendem o conceito de tipos e classes, por exemplo, não é necessário ficar explicando-os, pois isto faz parte do vocabulário de vocês. Além disso, provavelmente vai ser mais fácil entender um código ao perceber que ele usa um padrão conhecido.

Por serem soluções que já foram utilizadas várias e várias vezes, os padrões de projeto tendem a ser genéricos o suficiente para facilitar a acomodação de mudanças. Além disso, já foram validados por várias pessoas em vários projetos.

Ao longo da discussão sobre os padrões, será fácil ver que as soluções seguem os princípios de design orientado a objetos, como uma responsabilidade por classe e facilitar a extensão do comportamento.

Quando estou fazendo uma refatoração, geralmente sigo estes passos:

- 1) **Identificar uma oportunidades de melhoria:** antipadrões, ou “maus cheiros” (do inglês *code smells*), como são mais conhecidos, são sintomas de que o seu código não está tão bom quanto poderia. Um clássico exemplo são as linhas de comentários explicando o que o código está fazendo. O já mencionado livro do Martin Fowler (1999), *Refactoring: improving the design of existing code*, apresenta um bom catálogo de antipadrões.

Além dos antipadrões, também é possível melhorar o código pensando em facilitar futuras alterações. Ao desenvolver um sistema com vários tipos de usuários (usuários comuns, administradores, suporte, gerentes etc.), é mais fácil modelar uma classe `Usuario` considerando todos esses possíveis subtipos desde o começo.

- 2) **Entender o contexto do código:** como já foi falado antes, é muito importante levar em consideração o contexto ao decidir aplicar um padrão de projeto. Apesar de ser uma solução amplamente usada e testada, adicionar a estrutura para suportar um padrão também tem seus custos. Geralmente, são criadas abstrações que deixam o código mais espalhado entre classes.

Ao imaginar o design ideal, procure sempre soluções mais simples antes de aplicar um padrão. O termo *YAGNI*, que em inglês quer dizer “você não vai precisar disso” (*You ain't gonna need it*), explica exatamente a situação onde o código é desnecessariamente genérico.

- 3) **Aplicar pequenas mudanças nos testes e códigos:** um ponto interessante no processo de refatoração é seguir a mentalidade do TDD (*Test Driven*

*Development)* e fazer as alterações primeiro no teste. Apesar de não introduzir uma nova funcionalidade, refatorar o código pode causar mudanças nas interfaces dos componentes; portanto, modifique o teste para ver como o componente será usado antes de tomar a decisão final de aplicar ou não o padrão.

Após a refatoração, o sistema deve estar funcional, com todos os testes passando. Aplicar pequenas mudanças ajuda a reduzir o tempo em que o sistema fica quebrado, aumentando a confiança. Se o tempo entre testes de sucesso for muito grande, talvez seja melhor refatorar partes menores. Mas lembre de manter a visão do design ideal, para que não acabe com uma refatoração pela metade.

## CAPÍTULO 3

# Ruby e o paradigma orientado a objetos

Nas próximas seções do livro, vamos aplicar alguns dos padrões descritos pela Gangue dos Quatros. Porém, antes de mergulhar neles, vamos avaliar algumas características da linguagem Ruby que impactam diretamente na utilização dos padrões.

A linguagem Ruby, criada por Yukihiro “Matz” Matsumoto, permite misturar vários paradigmas. É possível escrever o código utilizando uma abordagem orientada a objetos, criando objetos que trocam mensagens, mas também podemos passar funções para métodos, atribuí-las a variáveis, e por aí vai. Esse conjunto bem variado de características impacta diretamente na utilização dos padrões em Ruby.

## 3.1 PENSANDO ORIENTADO A OBJETOS

### Mensagens e estado

Alan Kay, criador da programação orientada a objetos, descreve a linguagem usando a metáfora de células biológicas: objetos iriam se comunicar utilizando mensagens, visando proteger e esconder seu estado interno ([http://userpage.fu-berlin.de/\char126ram/pub/pub\\_jf47ht8iHt/doc\\_kay\\_oop\\_en](http://userpage.fu-berlin.de/\char126ram/pub/pub_jf47ht8iHt/doc_kay_oop_en)).

Uma célula não altera a estrutura interna de outra, elas apenas trocam mensagens por meio de estímulos. Uma vez recebido o estímulo externo, o estado interno da célula sofrerá modificações. A definição de Orientação a Objetos é fortemente baseada nesses dois conceitos, **mensagens e estado**.

O estado de um objeto é basicamente composto por seus atributos, mas eles representam mais do que apenas os dados que um objeto possui. Como um sistema é orquestrado ao redor de objetos, os dados representam uma parte do mundo virtual que foi modelado.

Se em algum ponto os dados estiverem em um estado inconsistente (por não existir ou possuir um valor inesperado), o sistema todo pode estar em risco. Esse é o principal motivo para que o estado interno de um objeto seja escondido, e um dos grandes motivadores da abstração de dados em objetos.

Ter o seu estado interno escondido e protegido não quer dizer que mudanças não ocorram, apenas que elas devem ser centralizadas dentro dos objetos. Para acionar essas mudanças, dois objetos trocam mensagens por meio de uma interface. Essa interface é formada pelo conjunto de métodos do objeto, e representa o comportamento dos objetos.

A linguagem Smalltalk, cocriada por Alan Kay, representa a aplicação dos conceitos fundamentais de OO, e foi uma forte influência para a linguagem Ruby. Assim, os conceitos de mensagens e estado também estão fortemente presentes na base de programas Ruby.

### Padrões orientado a objetos

Os padrões apresentados no livro *Design Patterns: elements of reusable object-oriented software* são implementados em duas linguagens: C++ e Smalltalk, ambas orientadas a objetos. No entanto, estas possuem características bem diferentes, uma que merece atenção especial é a tipagem.

C++ é estaticamente tipado e requer declaração de tipo. Ao criar uma variável, é necessário dizer explicitamente qual o seu tipo, e este não pode ser alterado durante a execução do programa. Para atribuir um valor de tipo diferente a uma variável, é preciso convertê-lo primeiro, mesmo que ocorra perda de informação.

```
float d = 1.97f;  
int i = (int) d; //converte para 1
```

Em linguagens com tipagem estática, um mecanismo de herança e polimorfismo é necessário para facilitar a troca dos tipos e evitar perda de informação. O principal impacto da tipagem estática no uso de padrões é a necessidade de criar mais classes para que a linguagem permita a troca dos tipos.

No exemplo a seguir, `LivroPromocional` precisa herdar da classe `Livro`, e apenas os métodos definidos em `Livro` estarão disponíveis, mesmo que `livro` aponte para um `LivroPromocional`. Então, o método `calcularPrecoFinal` está definido em `Livro`, mas pode ser sobreescrito em `LivroPromocional`.

```
Livro* livro = new LivroPromocional();  
p->calcularPrecoFinal();
```

Por outro lado, Smalltalk, que possui características bem semelhantes a Ruby, é dinamicamente tipado e não requer declaração de tipo, ou seja, não é preciso dizer qual o tipo do objeto que a variável vai referenciar. Qualquer tipo passado será atribuído sem problemas e, ao executar um método, o único requisito é que ele esteja declarado no tipo referenciado pela variável.

As diferenças na tipagem tornam as duas linguagens bem singulares, mas ainda assim ambas são orientadas a objetos. Então, ambas podem se beneficiar da aplicação de padrões, apenas a implementação será um pouco diferente.

## 3.2 O QUE TORNA RUBY TÃO ESPECIAL

Ruby possui um conjunto de características que impactam diretamente a implementação de quaisquer algoritmos. Os conceitos apresentados nesta seção

serão fortemente usados ao longo do livro.

## Tudo é um objeto

A primeira grande característica de Ruby é que tudo é visto como um objeto, mesmo inteiros ou caracteres (que são primitivos em algumas linguagens). Até mesmo classes são objetos do tipo `Class`, então regras que se aplicam a objetos são aplicáveis em todo o código Ruby.

```
livro = Livro.new
classe_livro = livro.class # retorna Livro
classe_livro.to_s # retorna "Livro"
classe_livro.class # retorna Class
```

Assim, o tipo de um objeto nada mais é do que uma referência para um objeto `Class`. Os métodos definidos na classe estão disponíveis para todas as instâncias, mas também é possível definir métodos diretamente em um objeto.

```
s = 'string qualquer'

# define método apenas em s
def s.bla; puts 'bla bla bla'; end

# retorna array vazio: []
String.instance_methods.grep /bla/

# retorna array com nome do método: [:bla]
s.methods.grep /bla/
```

Ao executar um método em um objeto, o interpretador Ruby vai buscar por uma implementação no próprio objeto e, caso não encontre, procura na sua classe. Essa é uma explicação simples – mas boa o suficiente – sobre como o modelo de objetos Ruby funciona e como tudo é um objeto.

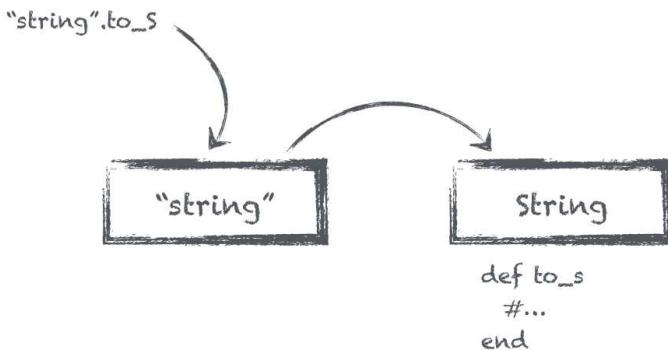


Fig. 3.1: Modelo de objetos Ruby simplificado

## Ducktyping

Como já falado, Ruby é bem semelhante a SmallTalk, principalmente na tipagem dinâmica ou *duck typing*. A ideia básica é que não importa qual tipo do objeto passado, desde que ele responda as mensagens necessárias.

Para exemplificar, vamos construir uma função que, dado um objeto, converta para string utilizando `to_s`, e mostre-a em caracteres minúsculos:

```

def mostrar_minusculo(objeto)
  puts objeto.to_s.downcase
end
  
```

Qualquer objeto que implemente `to_s` e `downcase` funciona sem problemas. Vejamos a seguir o poder do *duck typing* em prática:

```

mostrar_minusculo("UmA StrinG quAlquEr")
mostrar_minusculo(12) # mostra "12"
mostrar_minusculo(String) # mostra "string"
  
```

## Flexibilidade de métodos

Além de definir método apenas em um objeto, também podemos adicionar e reescrever métodos já definidos diretamente nas classes. Um bom exemplo é definir um método em `Numeric` (classe que engloba números) que calcula a raiz quadrada do número.

Ruby possui o módulo `Math`, que implementa diversas funções matemáticas – inclusive `sqrt`, que calcula a raiz quadrada do número passado como parâmetro.

```
Math.sqrt(25) # retorna 5.0
```

No entanto, seria muito mais legal ter esse comportamento diretamente em um número. Assim, em vez de passar o número como parâmetro para `Math.sqrt`, apenas executaríamos a chamada no próprio número. Para isso, basta reabrir a classe `Numeric` e definir o método:

```
class Numeric
  def sqrt
    Math.sqrt(self)
  end
end
```

```
25.sqrt # retorna 5.0
```

Além disso, também é possível reescrever um método de uma classe existente, o que altera o comportamento para todas as instâncias. Para redefinir os métodos de uma classe, é preciso apenas implementá-la novamente:

```
s = 'uma string qualquer'
s.to_s # retorna 'uma string qualquer'
class String
  def to_s
    'vish... quebrei...'
  end
end
s.to_s # retorna 'vish... quebrei...'
```

Ao reabrir uma classe, não estamos trocando todo o seu código interno, mas apenas adicionando ou modificando. Métodos já existentes continuam a

funcionar normalmente. Então, até mesmo o funcionamento das bibliotecas padrões em Ruby pode ser modificado, oferecendo uma grande flexibilidade para a criatividade do desenvolvedor, e uma igualmente grande responsabilidade ao escrever o código.

### 3.3 OUTRAS CARACTERÍSTICAS DA LINGUAGEM

Além das características citadas na seção anterior, Ruby possui várias outras funcionalidades, mas elas não serão abordadas diretamente ao longo do livro.

#### Mixins

Apesar de ser dinamicamente tipado, Ruby também possui mecanismos de herança e polimorfismo. O modelo de objetos de Ruby usa o conceito de antepassados para representar as classes da hierarquia de um objeto. Assim, ao chamar um método, o interpretador vai procurar por sua definição no objeto, na classe e em todos os seus antepassados. A figura a seguir mostra o modelo de objetos com antepassados, mas ainda é incompleto.

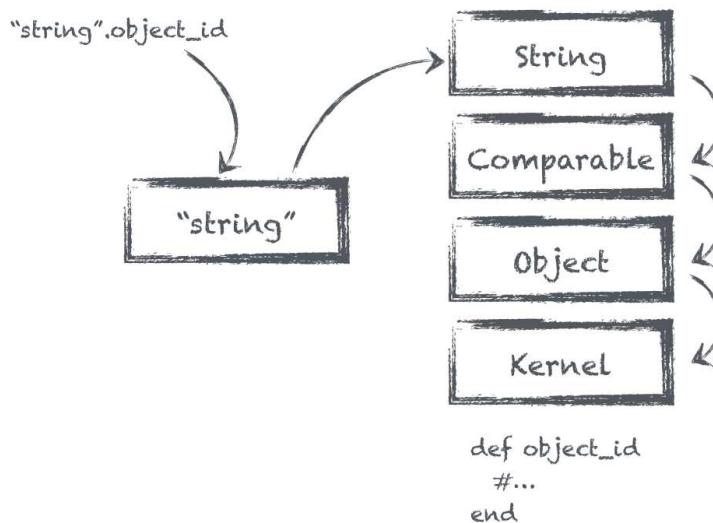


Fig. 3.2: Modelo de objetos Ruby com antepassados

Para alterar os antepassados de um objeto, é possível uma classe herdar de outra, mas apenas de uma classe. O método `ancestors` pode ser usado para verificar a lista de classes que são antepassados de uma classe:

```

class LivroPromocional < Livro
  # ...
end
LivroPromocional.ancestors
# => [LivroPromocional, Livro, Object, Kernel, BasicObject]
    
```

Nesse exemplo, é possível ver que classes já possuem um conjunto de antepassados em comum: `Object`, `Kernel` e `BasicObject`. Mas, além da herança única, a linguagem permite utilizar módulos que podem ser incluídos usando o conceito de *mixin*.

Ao incluir um módulo, todos os seus métodos ficam disponíveis na classe, e esta será adicionada à lista de antepassados.

```

class Livro
    
```

```
include Promocional
# ...
end
Livro.ancestors
# => [Livro, Promocional, Object, Kernel, BasicObject]
```

## Closures

Em Ruby, funções ou métodos são cidadãos de primeira classe, ou seja, eles são tratados da mesma maneira que qualquer outro objeto. Então, da mesma forma que guardamos valores em variáveis e passamos de um lado para outro como parâmetro, podemos fazer o mesmo com métodos.

O exemplo a seguir mostra como podemos capturar um método em uma variável e, depois, executá-lo:

```
def chama_metodo_com_log(metodo)
  puts "chamando #{metodo.name} em #{metodo.receiver}"
  metodo.call
end
metodo = "string".method(:upcase)
metodo.class
# => Method
chama_metodo_com_log(metodo)
# chamando upcase em string
# => "STRING"
```

Além de capturar métodos, também é possível criar blocos de código ao usarmos objetos `Proc`. Blocos são criados como parâmetros em métodos, utilizando lambda ou simplesmente instanciando um objeto `Proc`.

Todo método em Ruby aceita um bloco como parâmetro, mesmo que ele não seja explicitamente declarado como um. Acionar esse bloco requer o uso da palavra-chave `yield`, que vai transferir a chamada para o bloco passado.

```
def chama_bloco_com_log
  puts "executando bloco"
  yield
  puts "bloco executado"
end
```

```
chama_bloco_com_log do
  puts "codigo do bloco"
end
# executando bloco
# codigo do bloco
# bloco executado
```

O bloco passado como parâmetro para `chama_bloco_com_log` é uma instância de `Proc`. A principal diferença entre o objeto `Method` e o `Proc` é que o método fica atrelado a um objeto receptor (`metodo.receiver`).

Além de passar implicitamente como parâmetro, também podemos criar blocos usando a palavra-chave `lambda` (ou o operador `->`), ou instanciando `Proc`.

Para executar um bloco de código, basta chamar o método `call`.

```
def chama_bloco_com_log(bloco)
  puts "executando bloco"
  bloco.call
  puts "bloco executado"
end

bloco = lambda do # ou Proc.new do
  puts "codigo do bloco"
end

chama_bloco_com_log(bloco)
# executando bloco
# codigo do bloco
# bloco executado
```

## **Parte II**

### **Padrões de projeto comuns**



## CAPÍTULO 4

# Factory: gerenciando a criação de objetos

Rafael é um desenvolvedor com bastante experiência, não apenas com Ruby. No entanto, ele sabe ser pragmático e resolver problemas em vez de procurar a solução perfeita para cada situação. Além de ser um bom desenvolvedor, ele também tem o espírito empreendedor e está sempre em busca de uma ideia para seguir!

Em meio a tantos sites de compras com modelos iguais, Rafael e alguns amigos estão criando uma nova aplicação que promete ser diferente de todos os outros e revolucionar a maneira como compras são feitas! As expectativas estão altas, e a equipe quer buscar investidores para ajudar a fazer a ideia ficar popular.

Após uma pesquisa de mercado, Rafael nota que as características mais

importantes para um site de compras é uma ferramenta de busca. Usuários querem encontrar os produtos o mais rápido possível. Esse é exatamente um dos pontos fortes da aplicação de Rafael que ele planeja demonstrar em uma reunião com potenciais investidores.

Durante a reunião, tudo vai muito bem e o grupo de investidores fica maravilhado com a nova ferramenta, decidindo investir nela! Com toda a animação após a grande conquista, Rafael se junta à sua equipe, e decide que é hora de fazer algumas melhorias no produto para compensar as decisões tomadas na correria dos últimos dias. O primeiro alvo é a parte que prepara as buscas.

## 4.1 O CUSTO DA FLEXIBILIDADE

Como uma das principais funcionalidades são as várias buscas, o código que lida com elas precisa ser bem flexível para acomodar as necessidades de cada um dos usuários. Para essa primeira demonstração com os potenciais investidores, Rafael preparou três diferentes tipos de buscas: normal, por categoria e promocional.

É possível especificar em uma busca:

- a) A quantidade de produtos exibidos por página, sendo 15 por padrão;
- b) A categoria de produtos;
- c) A ordem de exibição dos produtos.

Na busca normal, apenas o nome do produto precisa ser especificado. Caso nenhuma opção de ordem de exibição seja usada, o resultado será ordenado por “relevância”.

Já na busca por categoria, é necessário especificar qual categoria se está buscando; caso contrário, ela volta a ser uma busca normal. A ordem de exibição dos resultados deve ser por “mais recente”, a menos que um valor seja especificado.

Por fim, na busca promocional, a categoria será sempre “em promoção” e o resultado sempre será ordenado por “mais recente”, não importando os outros valores.

Rafael ficou responsável por fazer a parte de busca da aplicação. O requisito básico é receber um *hash* com informações sobre a busca a ser realizada e, então, executar a busca.

O *hash* deve ser parecido com o modelo a seguir:

- `tipo_de_busca` – indica qual tipo de busca foi selecionado pelo usuário;
- `resultados_por_pagina` – indica quantos produtos por páginas devem ser exibidos;
- `categoria` – define que tipo de produto deve ser pesquisado;
- `ordenar_por` – diz qual o critério de ordenação escolhido pelo usuário;

```
#exemplo de hash
{
  tipo_de_busca: :promocional,
  resultados_por_pagina: 10,
  categoria: :eletronicos,
  ordenar_por: :relevancia
}
```

Rafael decidiu por criar uma classe `Busca` como ponto de entrada, recebendo o *hash* com os parâmetros da busca e, a partir dele, criar um objeto `CriterioDeBusca`, que vai basicamente conter as informações sobre a busca, como: ordem, quantidade de produtos por página etc. Com o critério de busca definido, a classe `ServicoDeBusca` será utilizada para fazer a busca de fato, retornando os `ids` dos itens correspondentes. Assim ficou o código:

```
#lib/factory/busca.rb
class Busca
  def self.por(params)
    criterio = criar_criterio(params)
    ids = ServicoDeBusca.realizar_busca_com(criterio)
    encontrar_produtos_por_ids(ids)
  end
end
```

Ao olhar para o código novamente, Rafael nota que esse método de Busca ficou bem simples, curto e com nomes legíveis. No entanto, o real problema está no método que cria o `CriterioDeBusca` a partir dos parâmetros. Devido à pressa para a grande reunião, seu código ficou bem “feio”, com vários `ifs` e bem difícil de manter. Rafael olha para o código e imagina o quanto difícil seria mudar um tipo de busca já existente, ou até mesmo incluir um novo.

```
#lib/factory/busca.rb
class Busca
  def self.criar_criterio(params)
    criterio = CriterioDeBusca.new
    criterio.por_pagina =
      params[:resultados_por_pagina] || 15
    criterio.categoria = params[:categoria]

    if params[:tipo_de_busca] == :promocional
      criterio.categoria = :em_promocao
      criterio.ordenar_por = :mais_recente
    elsif params[:tipo_de_busca] == :por_categoria
      if params[:categoria]
        criterio.ordenar_por =
          params[:ordenar_por] || :mais_recente
      else # volta para busca normal
        criterio.ordenar_por =
          params[:ordenar_por] || :relevancia
      end
    else # busca normal
      criterio.ordenar_por =
        params[:ordenar_por] || :relevancia
    end

    criterio
  end
end
```

Apesar de perceber que o código ficou feio, Rafael lembra de que também desenvolveu testes para garantir que tudo está funcionando como esperado, e fica mais aliviado. Um exemplo de teste é criar um conjunto específico

de parâmetros e validar o critério que foi construído a partir deles. Quando buscamos apenas pelo nome do produto, os outros atributos devem assumir valores padrão:

```
#spec/factory/busca_spec.rb
context 'para busca normal' do
  it 'retorna valores padrão' do
    params = {produto: 'produto qualquer'}
    criterio = Busca.criar_criterio(params)
    expect(criterio.por_pagina).to eq(15)
    expect(criterio.ordenar_por).to eq(:relevancia)
    expect(criterio.categoria).to eq(:tudo)
  end
end
```

Para cada um dos casos de busca descritos anteriormente, foram criados testes apropriados, garantindo que todos eles estejam cobertos e funcionando. O teste a seguir garante que, ao especificarmos uma quantidade de resultados por página, o critério é construído apropriadamente:

```
#spec/factory/busca_spec.rb
it 'retorna resultado por página quando especificado' do
  params = {
    produto: 'produto qualquer',
    resultados_por_pagina: 20
  }
  criterio = Busca.criar_criterio(params)
  expect(criterio.por_pagina).to eq(20)
end
```

Ao pensar em refatorar o código, Rafael tem a ideia de extrair a lógica de criação de buscas para outra classe, deixando para `Busca` apenas a responsabilidade de chamar o serviço de busca para mapear os `ids` dos produtos. Além de criar outra classe, ele também gostaria que cada tipo de busca pudesse ser criado em seu método próprio, separando melhor os vários `ifs` do código.

Extrair a lógica de criação de objetos para classes e métodos específicos é o objetivo dos padrões *Factory*: Simple Factory, Factory Method e Abstract Factory.

## 4.2 OS PADRÕES FACTORY

O problema que esses padrões resolvem é criar instâncias de objetos separadas do resto da lógica; por isso são classificados como padrões de criação. Entretanto, Simple Factory, Factory Method e Abstract Factory diferem bastante na solução para o problema, o que torna a escolha de qual utilizar muito dependente do contexto em que serão usados.

Uma nomenclatura comum ao falar sobre padrões *Factory* é que os objetos criados são chamados de **produtos**, e as classes que os criam são as **fábricas**, utilizando a metáfora de que fábricas fazem produtos.

Vamos começar com o Simple Factory, que é o mais simples e parece ser exatamente a refatoração que Rafael precisa!

### Simple Factory

O contexto de utilização do Simple Factory é quando você possui apenas um tipo produto para ser criado e existe apenas uma maneira de fazê-lo, ou seja, apenas uma fábrica. A solução é criar uma classe para extrair o comportamento de criação de objetos.

Para o problema de Rafael, no qual ele quer separar a lógica que cria os objetos de busca da que os utiliza, esse é exatamente o contexto. O único objeto que precisa ser criado é o `CriterioDeBusca` e, apesar das diferentes configurações, todos são criados da mesma maneira. A solução parece que ficará bem simples e ideal.

Será criada, então, a classe `FabricaDeCriterio`, na qual serão instanciados objetos `CriterioDeBusca` com os valores apropriados. Para cada um dos tipos de busca, será criado um método que vai conter as regras específicas deles. Essa refatoração é conhecida por **Extrair Classe**, que tem como principal objetivo separar responsabilidades em duas classes diferentes.

No nosso caso, a classe `Busca` configura um critério e realiza a busca, assim, vamos separar a responsabilidade de criar/configurar um critério em `FabricaDeCriterio`. Como um exemplo, veja como ficaria a criação de buscas promocionais:

```
#lib/factory/fabrica_de_criterio.rb
class FabricaDeCriterio
```

```
def self.criar_promocional(params)
  criterio = CriterioDeBusca.new
  criterio.por_pagina = params[:resultados_por_pagina] || 15
  criterio.categoria = :em_promocao
  criterio.ordenar_por = :mais_recente
  criterio
end
end
```

Agora, na classe Busca, basta acionar o método apropriado da FabricaDeCriterio para criar o CriterioDeBusca.

```
#lib/factory/busca.rb
class Busca
  def self.criar_criterio(params)
    if params[:tipo_de_busca] == :promocional
      FabricaDeCriterio.criar_promocional(params)
    elsif params[:tipo_de_busca] == :por_categoria
      FabricaDeCriterio.criar_por_categoria(params)
    else
      FabricaDeCriterio.criar_busca_normal(params)
    end
  end
end
```

A próxima refatoração é mover o método `criar_criterio` para a `FabricaDeCriterio`, utilizando **Mover Método**, para separar completamente as responsabilidades.

```
#lib/factory/fabrica_de_criterio.rb
class FabricaDeCriterio
  def self.criar_criterio(params)
    if params[:tipo_de_busca] == :promocional
      criar_promocional(params)
    elsif params[:tipo_de_busca] == :por_categoria
      criar_por_categoria(params)
    else
      criar_busca_normal(params)
    end
  end
```

```
end
end
```

Agora que a lógica para criar critérios está definida na classe `FabricaDeCriterio`, podemos apenas mover os testes da classe `Busca` e usar a nova estrutura.

```
#spec/factory/fabrica_de_criterio_spec.rb
context 'para busca normal' do
  it 'retorna valores padrão' do
    params = {produto: 'produto qualquer'}
    criterio = FabricaDeCriterio.criar_criterio(params)
    expect(criterio.por_pagina).to eq(15)
    expect(criterio.ordenar_por).to eq(:relevancia)
    expect(criterio.categoria).to eq(:tudo)
  end
end
```

Agora, nos testes da classe `Busca`, vamos apenas garantir que a classe `FabricaDeCriterio` está sendo chamada. Para isso, esperaremos que o método `criar_criterio` seja chamado na classe `FabricaDeCriterio`, com o conjunto especificado de parâmetros, ao chamar `Busca.por`.

```
describe Busca do
  it 'utiliza FabricaDeCriterio para criar criterio de busca' do
    params = {tipo_de_busca: :por_categoria}
    expect(FabricaDeCriterio).to
      receive(:criar_criterio).with(params)
    Busca.por(params)
  end
end
```

Por fim, utilizáramos a classe `FabricaDeCriterio` diretamente no método `Busca.por`:

```
#lib/factory/busca.rb
class Busca
  def self.por(params)
    criterio = FabricaDeCriterio.criar_criterio(params)
```

```
ids_de_resultado =  
  ServicoDeBusca.realizar_busca_com(criterio)  
encontrar_produtos_por_ids(ids_de_resultado)  
end  
end
```

No final, a classe `Busca` ficou bem mais simples e com menos responsabilidades. O Simple Factory não é tão complicado, é apenas uma nova classe com alguns métodos especializados em criar outros objetos.

Na verdade, o Simple Factory nem é considerado, por alguns autores, um padrão, mas sim um idioma de linguagem. Veja o quão simples é o diagrama de uso:

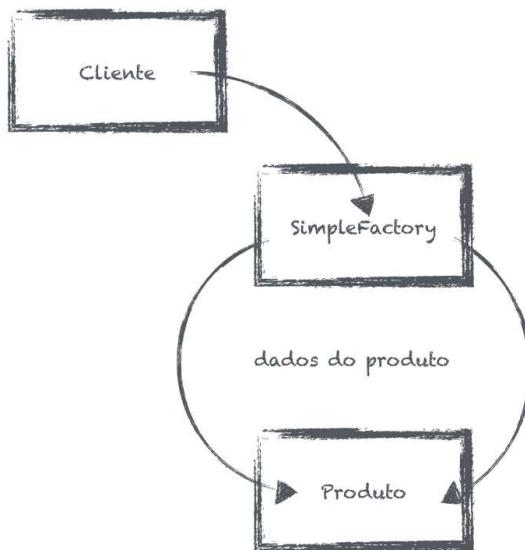


Fig. 4.1: Funcionamento do Simple Factory

Apesar de simples, ele resolve o problema sem criar uma estrutura de classes muito grande, o que agrada muito um desenvolvedor pragmático como Rafael. Porém, caso a quantidade de métodos implementados na classe fábrica comece a aumentar demais, talvez seja um indicador de que é melhor

criar novas fábricas, ou até mesmo avaliar a utilização de outro padrão.

## Factory Method

O contexto do Factory Method é criar um mesmo tipo de produto, mas com diversas fábricas, cada uma funcionando de uma maneira própria. Assim, além das configurações passadas, cada fábrica possui uma lógica específica que afeta o produto final.

No exemplo de Rafael, não é necessário se preocupar em maneiras de criar os objetos, pois todos serão criados do mesmo jeito. Portanto, usar o Factory Method nesse caso seria mais complicado que o Simple Factory, e não traria nenhum outro benefício.

No entanto, imagine que, para oferecer mais tipos diferentes de busca, ele deseja utilizar múltiplas *engines*, como Elasticsearch, Solr ou uma busca comum no banco de dados. Além das configurações do objeto de busca `CriterioDeBusca`, cada uma das ferramentas de busca precisaria de configurações próprias, como o nome do `host`, a consulta formatada, configurações de limite de tempo etc.

O código simplificado a seguir mostra como seria a implementação para a situação anterior. O `hash` `params` contém também qual ferramenta de busca deve ser usada para a pesquisa e, a partir dela, decidimos qual *engine* utilizar:

```
if params[:ferramenta_de_busca] == :elasticsearch
  if params[:tipo_de_busca] == :promocional
    # código para tratar busca promocional
  elsif params[:tipo_de_busca] == :por_categoria
    # código para tratar busca por categoria
  else
    # código para tratar busca normal
  end
  # configuração elasticsearch
elsif params[:ferramenta_de_busca] == :solr
  # mesmos ifs anteriores
  # configuração Solr
elsif params[:ferramenta_de_busca] == :banco_de_dados
  # mesmos ifs anteriores
  # configuração banco de dados
```

```
end
```

Aplicar o Factory Method ajudaria da seguinte maneira: para cada *engine*, seria criada uma classe fábrica que saberia apenas sobre suas próprias configurações, e todas as classes fábricas criariam os mesmos objetos de busca. Assim, o mesmo código do cliente que usa a busca no banco de dados pode também utilizar a via ElasticSearch.

A classe `FabricaElasticsearch` terá apenas a configuração relacionada a *engine* `ElasticSearch` e como os objetos `CriterioDeBusca` devem ser criados para que a busca seja feita corretamente.

Ao aplicar o Factory Method, cada um dos `ifs` que verifica a ferramenta de busca seria extraído para uma classe fábrica, e cada `if` dentro dele seria extraído para um método. Por exemplo, para criar buscas com `ElasticSearch`, essa fábrica seria utilizada:

```
class FabricaElasticsearch
  @@configuracao = {
    host: 'elasticsearch.myhost.com',
    porta: '4004',
    limite_de_tempo: 10
  }

  def self.promocional; end
  def self.por_categoria; end
  def self.normal; end
end
```

O diagrama de utilização do Factory Method é simples, a única diferença é que vamos ver mais de uma classe fábrica criando os produtos:

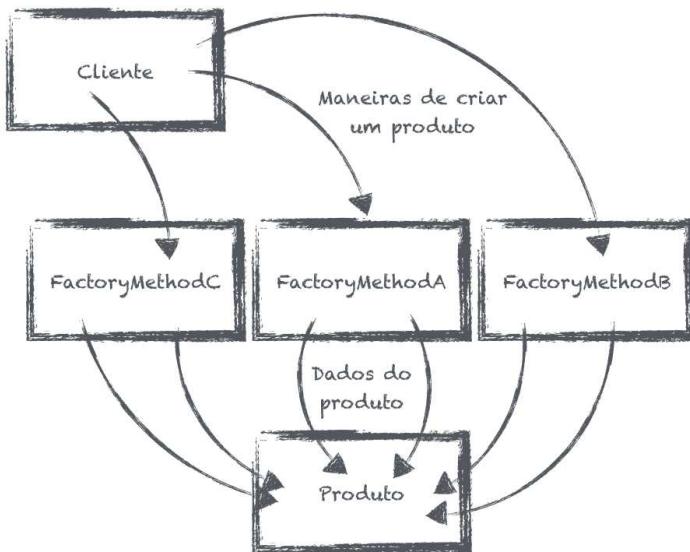


Fig. 4.2: Funcionamento do Factory Method

A classe que continha a lógica maluca para criar buscas agora vai usar os seguintes passos:

- 1) Instanciar uma nova fábrica, ou recebê-la como parâmetro;
- 2) Chamar o método específico para criar o produto;
- 3) A fábrica vai executar sua lógica e retornar o produto.

## INVERSÃO DE DEPENDÊNCIAS

Ao utilizar o Factory Method, podemos receber a fábrica diretamente em vez de decidir, a partir de um parâmetro, qual deve ser instanciada. Dessa forma, vamos apenas usar a fábrica que foi passada e chamar o método diretamente:

```
def self.criar_criterio(fabrica, params)
  if params[:tipo_de_busca] == :promocional
    fabrica.criar_promocional
  # ...
end
```

Essa inversão de pensamento é conhecida como o **Princípio da Inversão de Dependência**, que sugere depender de classes abstratas, e não de classes concretas. Assim, o mesmo pedaço de código funciona em várias situações diferentes.

Em Ruby, devido ao *duck typing*, não precisamos realmente criar uma classe “abstrata”, basta que o mesmo método seja implementado.

## Abstract Factory

Para o Abstract Factory, o contexto de utilização é mais complexo que o Factory Method. Seguindo a analogia das fábricas novamente, existirão múltiplas fábricas que criam múltiplos tipos de produtos.

Como um exemplo mais concreto para usar o Abstract Factory, imagine que você está trabalhando em um framework no qual escrevemos um aplicativo em Ruby, e que ele vai converter o código para múltiplas plataformas, como Android, iOS e HTML.

Ao desenvolver uma aplicação, não precisamos especificar qual plataforma será utilizada, então, podemos usar uma fábrica para isso. Precisamos que as fábricas criem vários elementos de interface, como botões, *inputs*, *dropdowns* etc.

Logo, uma `FabricaHTML` definiria a lógica específica para criar um botão em HTML, enquanto a `FabricaAndroid` vai definir a lógica para criar

o botão em Java etc. O cliente agora pode criar uma família de produtos (por exemplo, botões e *dropdowns*), e cada um destes pode ser criado por uma fábrica diferente (por exemplo, Android e HTML).

O diagrama de uso do Abstract Factory é bem mais complexo do que os anteriores:

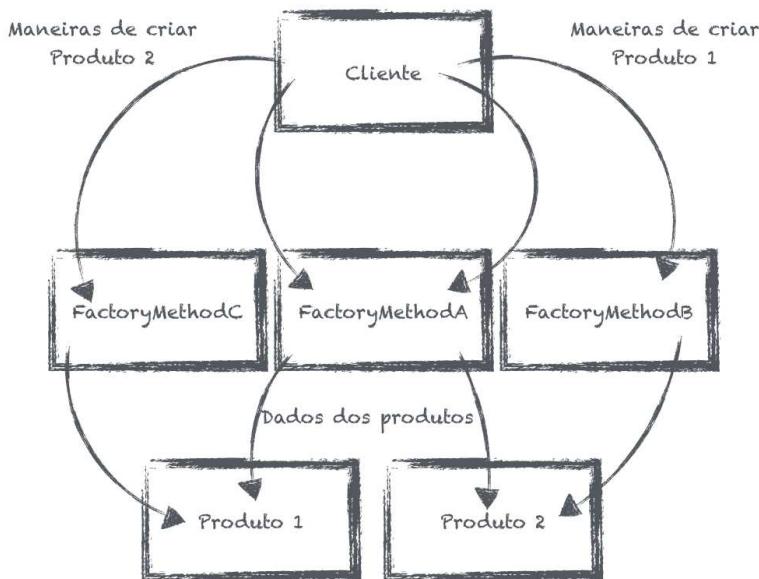


Fig. 4.3: Funcionamento do Simple Factory

A não ser que seu projeto tenha múltiplas dependências, como no exemplo anterior, utilizar o Simple Factory vai resolver o problema, sem a necessidade de criar várias camadas diferentes.

## CAPÍTULO 5

# Strategy: dividir para simplificar

Paula sempre foi fã de redes sociais e possui uma conta em praticamente todas que você possa imaginar. Então, ao desenvolver o **EuS2Livros**, ela juntou suas grandes paixões: livros, redes sociais e programação!

O EuS2Livros é uma nova rede social para juntar autores e pessoas apaixonadas por livros, aproximando mais os autores de seu público. O projeto parece bem promissor, e ela e seu grupo estão animados com as possibilidades de crescimento do negócio. Como última etapa do desenvolvimento, eles querem facilitar a entrada de novos usuários, permitindo-os utilizarem suas contas já existentes em outras redes sociais.

A arquitetura e o design do aplicativo foram muito bem desenvolvidos, e o módulo de login pode ser facilmente acoplado ao resto da aplicação. Tudo o que eles precisam fazer é validar o usuário, usando a rede social de sua escolha, e devolver o status (autenticado ou não) e uma mensagem que será exibida em caso de erros.

## 5.1 UM LOGIN COM VÁRIOS PROVEDORES

Suponha que você tem uma conta na rede social FaceNote, com ela você pode fazer o login no EuS2Livros sem precisar criar um usuário e senha. A aplicação pede permissão para o FaceNote enviando seus dados de usuário, assim, o FaceNote valida as suas informações e diz se você permitiu ou não o acesso ao EuS2Livros.

Paula ficou responsável por essa parte do sistema e decidiu criar classes de serviço para cada rede social, enviando os dados do usuário por meio de sua API específica. Os serviços vão receber um *hash* com as informações da requisição, incluindo as informações sobre o usuário. Um exemplo é o serviço que faz login via FaceNote, que foi implementado assim:

```
#lib/strategy/servico_facenote_login.rb
class ServicoFaceNoteLogin
  def self.autenticar(dados)
    begin
      resposta = atenticacao_via_post(dados[:usuario])
      return resposta.status
    rescue TimeoutException => e
      log.error('Timeout ao fazer login via FaceNote')
    end
  end
end
```

Inicialmente, Paula criou o `ServicoFaceNoteLogin` para fazer login via FaceNote, e `ServicoZuiterLogin` para fazer login via Zuiter, outra famosa rede social. Sua ideia ao criar as classes de serviço é garantir que eles possam evoluir separadamente, conforme necessário, já que cada um dos serviços possui suas particularidades:

- **API do FaceNote**
  - a) Login com sucesso 200
  - b) Aplicação com acesso revocado 403
  - c) Aplicação bloqueada 408

- API do Zuiter

- Login com sucesso 202
- Autorização pendente 400

Estes códigos de resposta serão utilizados pela classe `Login` para identificar o status da autorização. Paula adicionou várias constantes dentro da classe para evitar os números mágicos no meio do código, e deixá-lo mais legível.

```
#lib/strategy/login.rb
class Login
  FACE_NOTE_SUCESSO = 200
  FACE_NOTE_REVOCADO = 403
  FACE_NOTE_BLOQUEUADO = 408

  ZUITER_SUCESSO = 202
  ZUITER_PENDENTE = 400
end
```

O código da classe `Login` recebe um *hash* contendo o método de login que deve ser utilizado e os dados do usuário. Depois, chama os serviços passando os dados, e confere o código de retorno para saber se o usuário conseguiu ou não fazer o login.

```
#lib/strategy/login.rb
class Login
  def self.com(parametros)
    resposta = if parametros[:metodo] == :facenote
      ServicoFaceNoteLogin.autenticar(parametros[:dados])
    elsif parametros[:metodo] == :zuiter
      parametros[:dados][:usuario].downcase!
      ServicoZuiterLogin.autenticar(parametros[:dados])
    end

    status, mensagem = false, 'não foi possível autenticar'

    if resposta == FACE_NOTE_SUCESSO ||

```

```

        resposta == ZUITER_SUCESSO
        status, mensagem = true, 'login com sucesso'
    elsif resposta == FACE_NOTE_REVOCADO
        status, mensagem = false, 'acesso revocado'
    elsif resposta == FACE_NOTE_BLOQUEADO
        status, mensagem = false, 'aplicação bloqueada'
    elsif resposta == ZUITER_PENDENTE
        status, mensagem = false, 'autorização pendente'
    end

    {status: status, mensagem: mensagem}
end
end

```

No final, essa é a maneira como a aplicação vai se comunicar com a API do FaceNote para permitir ou não o acesso de usuários:

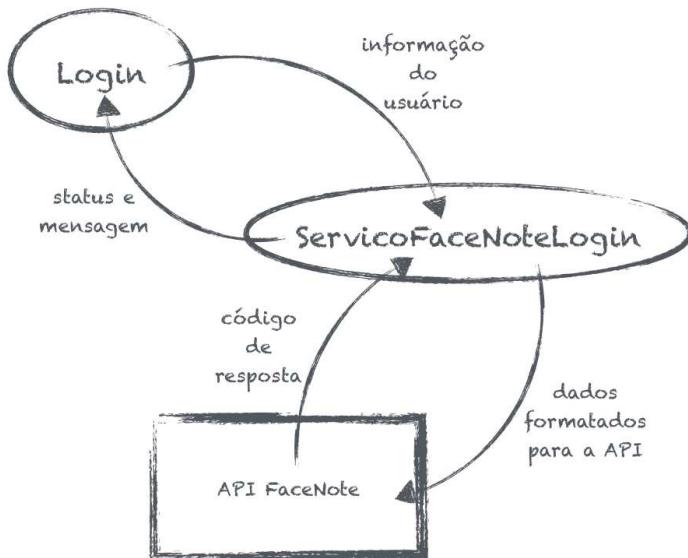


Fig. 5.1: Fluxo de chamada para API do FaceNote

Um exemplo de teste que Paula escreveu para essa classe é: criar o *hash*

parametros com dados falsos, chamar `Login.com(parametros)`, e verificar o status e a mensagem que são retornadas.

```
#spec/strategy/login_spec.rb
context 'fazendo login via FaceNote' do
  it 'retorna sucesso para o usuário Gil' do
    parametros = {
      metodo: :facenote,
      dados: {
        usuario: 'Gil',
      }
    }
    resposta = Login.com(parametros)
    expect(resposta[:status]).to be true
    expect(resposta[:mensagem]).to eq('login com sucesso')
  end
end
```

Como os testes precisam ser executados frequentemente, Paula também implementou dublês de teste do tipo *fake*. Dublês de teste ajudam a reduzir o custo dos testes, ignorando comportamentos desnecessários. Um *fake*, por exemplo, sobrecarrega o comportamento original da classe, e possibilita controlar diretamente o retorno dos métodos. Assim, Paula consegue evitar a chamada ao serviço externo durante os testes.

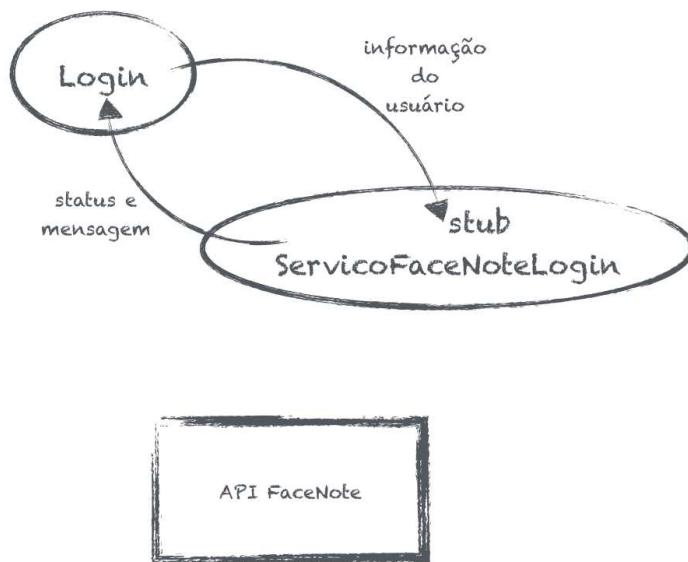


Fig. 5.2: Fluxo de chamada para API do FaceNote utilizando fake

Por exemplo, será retornado 202 sempre que o nome do usuário passado como parâmetro for 'Gil', e 404 para qualquer outro usuário.

```
#spec/strategy/servico_facenote_login_fake.rb
class ServicoFaceNoteLogin
  def self.autenticar(dados)
    return 200 if dados[:usuario] == 'Gil'
    404
  end
end
```

Assim, no teste, podemos usar o usuário 'Gil' para os casos de login com sucesso. Para cobrir os outros cenários, basta adicionar novos usuário no *fake*. Por exemplo, Paula utiliza um usuário 'Ana' para testar o caso da aplicação com acesso revocado; o teste seria bem parecido com o anterior:

```
#spec/strategy/login_spec.rb
context 'fazendo login via FaceNote' do
```

```
it 'retorna acesso revocado para usuário Ana' do
  parametros = {
    metodo: :facenote,
    dados: {
      usuario: 'Ana',
    }
  }
  resposta = Login.com(parametros)
  expect(resposta[:status]).to be false
  expect(resposta[:mensagem]).to eq('acesso revocado')
end
end
```

Para o teste passar, basta alterar o duplê da classe `ServicoFaceNoteLogin` para lidar com o novo usuário:

```
#spec/strategy/servico_facenote_login_fake.rb
class ServicoFaceNoteLogin
  def self.autenticar(dados)
    return 200 if dados[:usuario] == 'Gil'
    return 403 if dados[:usuario] == 'Ana'
    404
  end
end
```

Com essa primeira versão terminada e bem testada, o time agora vai procurar por oportunidades de refatoração. O primeiro mau cheiro levantado é que o método `Login.com` já está bem grande e vai crescer mais ainda, conforme novas redes sociais forem adicionadas.

Paula olha o código mais um pouco e sugere separar as lógicas de login específicas de cada provedor, extraindo-as para classes próprias e criando diferentes estratégias de login. Para isso, ela sugere utilizar o padrão Strategy!

## 5.2 O PADRÃO STRATEGY

Por ser classificado como um padrão de comportamento, o padrão Strategy resolve problemas de distribuição de responsabilidades. Para sua aplicação, é necessário que exista uma maneira clara de separar essas responsabilidades

em algoritmos próprios. A solução proposta é encapsular esses algoritmos nas estratégias, de maneira que trocá-las seja bem fácil.

Cada uma das maneiras de fazer login pode ser considerada uma estratégia e, em vez de a classe `Login` conter todas as lógicas, ela vai apenas delegar a chamada de acordo com os parâmetros recebidos. Dessa forma, a lógica específica de uma API fica autocontida na sua classe, centralizando mudanças.

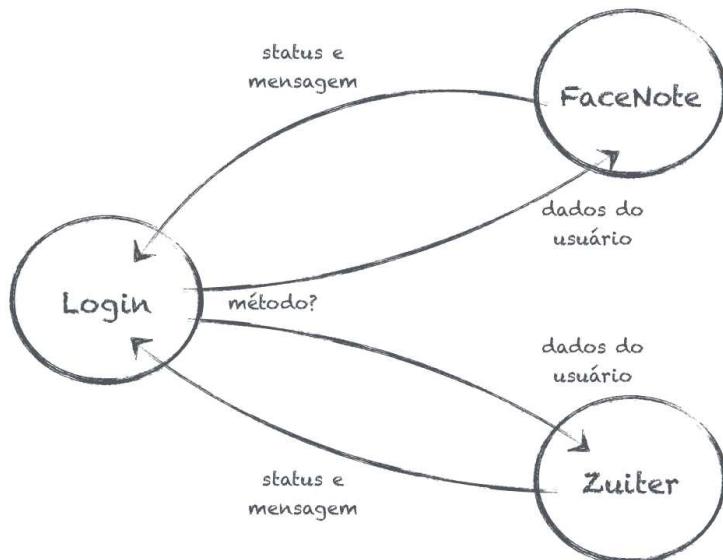


Fig. 5.3: Comunicação com diferentes estratégias

Caso a API do Zuiter sofra uma alteração, em vez de alterar o método gigante da classe `Login`, basta fazer a mudança na classe de estratégia específica do `Zuiter`. O resto da lógica de validação continua funcionando da mesma maneira.

O primeiro passo para aplicar o `Strategy` é separar as lógicas ainda dentro da classe `Login`. Em vez de o método `Login.com` possuir lógica para todos os serviços, ele vai apenas delegar a chamada para outros métodos. Como exemplo, vamos usar **Extrair Método** para separar a lógica de autenticação via `FaceNote`, de forma que o método já retorne o `hash` com `status`

e mensagem:

```
#lib/strategy/login.rb
class Login
  def self.login_via_face_note(parametros)
    resposta = ServicoFaceNoteLogin.autenticar(parametros)
    if resposta == FACE_NOTE_SUCESSO
      status, mensagem = true, 'login com sucesso'
    elsif resposta == FACE_NOTE_REVOCADO
      status, mensagem = false, 'acesso revocado'
    elsif resposta == FACE_NOTE_BLOQUEADO
      status, mensagem = false, 'aplicação bloqueada'
    else
      status, mensagem = false, 'não foi possível autenticar'
    end
  end
end
```

Após extrair todas as formas de login para seus próprios métodos, Paula nota que o método principal fica responsável apenas por decidir qual forma de autenticação usar. Isso deixa o código bem menor e mais simples de se entender.

```
#lib/strategy/login.rb
class Login
  def self.com(parametros)
    resposta = if parametros[:metodo] == :facenote
      login_via_face_note(parametros)
    elsif parametros[:metodo] == :zuiter
      parametros[:dados][:usuario].downcase!
      login_via_zuiter(parametros[:dados])
    end
  end
end
```

O próximo passo é extrair as estratégias. Paula cria a classe `LoginViaFaceNote`, onde ficará contida a lógica para chamar o `ServicoFaceNoteLogin` e tratar o código de resposta da API. Com os métodos já separados na classe `Login`, basta utilizar **Extrair Classe** e

mover os métodos e constantes relacionadas à autenticação via FaceNote para a classe específica.

```
#lib/strategy/login_via_facenote.rb
class LoginViaFaceNote
  FACE_NOTE_SUCESSO = 200
  FACE_NOTE_REVOCADO = 403
  FACE_NOTE_BLOQUEADO = 408

  def self.autenticar(parametros)
    resposta = ServicoFaceNoteLogin.autenticar(parametros)
    if resposta == FACE_NOTE_SUCESSO
      status, mensagem = true, 'login com sucesso'
    elsif resposta == FACE_NOTE_REVOCADO
      status, mensagem = false, 'acesso revocado'
    elsif resposta == FACE_NOTE_BLOQUEADO
      status, mensagem = false, 'aplicação bloqueada'
    else
      status, mensagem = false, 'não foi possível autenticar'
    end
  end
end
```

Um bom exemplo da transparência do padrão Strategy é que, ao extrair estratégias, os testes da classe `Login` não precisam ser alterados. O contrato, receber `hash` com parâmetros e devolver um `hash` com status e mensagem, continua o mesmo.

Mover os testes da classe `Login` para as classes de estratégia é uma boa ideia, pois deixa claro e bem documentado qual o comportamento esperado das estratégias, além de ser uma mudança bem simples.

Veja como ficariam os testes para `LoginViaFaceNote`:

```
#spec/strategy/login_via_facenote_spec.rb
describe LoginViaFaceNote do
  it 'retorna sucesso para o usuário Gil' do
    parametros = {
      usuario: 'Gil'
    }
```

```
resposta = LoginViaFaceNote.autenticar(parametros)
expect(resposta[:status]).to be true
expect(resposta[:mensagem]).to eq('login com sucesso')
end
end
```

Para os testes da classe `Login`, Paula precisa apenas garantir que, ao tentar uma autenticação, a estratégia correta será aplicada. Testar, por exemplo, que `LoginViaFaceNote` é usado ao tentar fazer login via FaceNote requer apenas verificar que o método `LoginViaFaceNote.autenticar` será chamado ao executar `Login.com`:

```
#spec/strategy/login_spec.rb
describe Login do
  context 'fazendo login via FaceNote' do
    it 'utiliza LoginViaFaceNote para autenticar' do
      parametros =
      {
        metodo: :facenote,
        dados: {
          usuario: 'Gil'
        }
      }
      expect(LoginViaFaceNote).to receive(:autenticar)
        .with(parametros[:dados])
      expect(LoginViaZuiter).to receive(:autenticar).never
      Login.com(parametros)
    end
  end
end
```

Por fim, tudo o que a classe `Login` vai precisar fazer é chamar a estratégia `LoginViaFaceNote.autenticar`, ficando bem mais simples:

```
#lib/strategy/login.rb
class Login
  def self.com(parametros)
    if parametros[:metodo] == :facenote
      LoginViaFaceNote.autenticar(parametros[:dados])
```

```
elsif parametros[:metodo] == :zuiter
    LoginViaZuiter.autenticar(parametros[:dados])
end
end
end
```

Responsabilidades foram distribuídas e as lógicas não estão mais misturadas em um só lugar. Além disso, adicionar novas maneiras de login requer poucas mudanças nas classes já existentes, basta a classe `Login` chamar para a nova estratégia, que precisa apenas obedecer ao contrato já definido.

Paula olha de novo para o código e gosta do que vê. As lógicas das várias redes sociais estão bem separadas nas estratégias e podem crescer sem nenhum impacto maior na aplicação. Na verdade, ela percebe que a sua ideia de criar diferentes classes de serviço nem é mais necessária, dado que o código dos serviços `ServicoFaceNoteLogin` e `ServicoZuiterLogin` é praticamente o mesmo. Nesse momento, ela já começa a pensar na sua próxima refatoração.

## CAPÍTULO 6

# Template Method: definindo algoritmos extensíveis

Em sistemas web, é muito comum existirem tarefas que precisam ser realizadas de maneira assíncrona, para que o servidor não fique travado apenas esperando e consumindo recursos desnecessários. Os processos que executam essas tarefas paralelas ao sistema principal são conhecidos como *workers*.

No seu projeto atual, Débora está trabalhando em algoritmos para alguns *workers* de um sistema. Ela já desenvolveu funcionalidades parecidas antes para outros, mas esse em especial requer um trabalho a mais. Esse projeto precisará de vários tipos de *workers*, por exemplo, enviar e-mails, importar informações a partir de um arquivo, ou até mesmo realizar buscas periódicas no banco de dados.

## 6.1 NEM TÃO DIFERENTES ASSIM

Inicialmente, Débora implementou o *worker* que envia e-mails. O teste a seguir mostra o seu funcionamento básico: dado um usuário, uma lista com destinatários e qual tipo de conteúdo da mensagem (convite, promocional etc.). Verificamos que foi enviado um e-mail para cada um dos destinatários com os dados corretos:

```
#spec/template/email_worker_spec.rb
it 'deve mandar um email para cada destinatario' do
  id_usuario = 1
  lista_de_destinatarios = ['email@email.com',
                            'outro_email@email.com',
                            'email_qualquer@email.com']
  assunto = 'convite enviado por Usuario 1'
  email = EmailWorker.new.enviar(
    usuario: id_usuario,
    destinatarios: lista_de_destinatarios,
    mensagem: :convite)
  expect(email[:para]).to eq(lista_de_destinatarios)
  expect(email[:assunto]).to eq(assunto)
  expect(email[:emails_enviados]).to eq(3)
end
```

A implementação também é bem simples: primeiro, buscamos as informações do usuário, para então gerar o conteúdo do e-mail e o assunto. Em seguida, chamamos o método `enviar_email` passando as informações para envio:

```
#lib/template/email_worker.rb
class EmailWorker
  def enviar parametros
    usuario = buscar_usuario(parametros[:usuario])
    corpo = gerar_mensagem(parametros[:mensagem],
                           usuario)
    assunto = gerar_assunto(parametros[:mensagem],
                           usuario)
    enviar_email(de: usuario,
                 para: parametros[:destinatarios],
```

```
    assunto: assunto,
    corpo: corpo)
end
end
```

Os outros métodos não precisam ser detalhados aqui, pois não influenciam no entendimento do problema.

Ao planejar a implementação dos próximos *workers*, Débora percebe que será necessário adicionar comportamentos comuns, como *logs*, gerenciamento de exceções, estabelecer um tempo limite para que o processo não fique executando indefinidamente, entre outros.

Levando esses novos requisitos em consideração, a implementação final do método `enviar` ficou assim:

```
#lib/template/email_worker.rb
class EmailWorker
  def enviar parametros
    @limite_timeout = 10
    @limite_tentativas = 5
    @logger = Logger

    usuario = buscar_usuario(parametros[:usuario])
    corpo = gerar_mensagem(parametros[:mensagem],
                           usuario)
    assunto = gerar_assunto(parametros[:mensagem],
                           usuario)
    contador_tentativas = 0
    begin
      timeout(@limite_timeout) do
        enviar_email(de: usuario,
                     para: parametros[:destinatarios],
                     assunto: assunto,
                     corpo: corpo)
      end
    rescue Timeout::Error
      contador_tentativas += 1
      @logger.error("Timeout::Error EmailWorker.enviar_email")
      retry if contador_tentativas < @limite_tentativas
    end
  end
end
```

```

end
end
end

```

Após avaliar o problema com uma visão mais geral, Débora percebe que, apesar de as tarefas serem diferentes, o fluxo de execução dos *workers* pode seguir um mesmo padrão. Ela acaba rascunhando algo assim:



Fig. 6.1: Fluxo de execução dos workers

Pensando em uma maneira de fazer com que esse algoritmo fique simples de ser reutilizado, ela avalia criar apenas um *worker*, e definir o que ele executa usando o padrão Strategy para trocar as implementações. O problema é que cada *worker* precisa fazer mais do que apenas o método de execução, o que pode tornar as estratégias mais complicadas.

O que ela precisa é definir um modelo para todos os *workers* seguirem e, assim, reduzir as duplicações entre eles. E é então que ela descobre o Template Method!

## 6.2 O PADRÃO TEMPLATE METHOD

Assim como no Strategy, o Template Method também é um padrão comportamental que busca simplificar as responsabilidades dos objetos. O problema a ser resolvido é que temos um conjunto de algoritmos que, apesar de seguirem um mesmo fluxo, precisam de alguma flexibilidade.

O contexto para utilização do padrão é quando podemos separar o comportamento base, comum a todos os algoritmos, criando um *template* com pontos de extensão.

A estrutura base define os métodos que são chamados, a ordem de execução e o que é retornado por eles, em uma única classe que será utilizada pelo cliente para executar os algoritmos. Cada algoritmo específico herda dessa classe base, e sobrescreve os métodos para implementar sua própria lógica.

Os métodos que são sobreescritos em cada algoritmo são chamados de **métodos gancho**, e fornecem um alto nível de flexibilidade para a solução, em que parte fica compartilhada e parte é específica. Além disso, por existir apenas uma única classe para chamar os algoritmos, o código de utilização fica o mesmo para qualquer que seja a implementação.

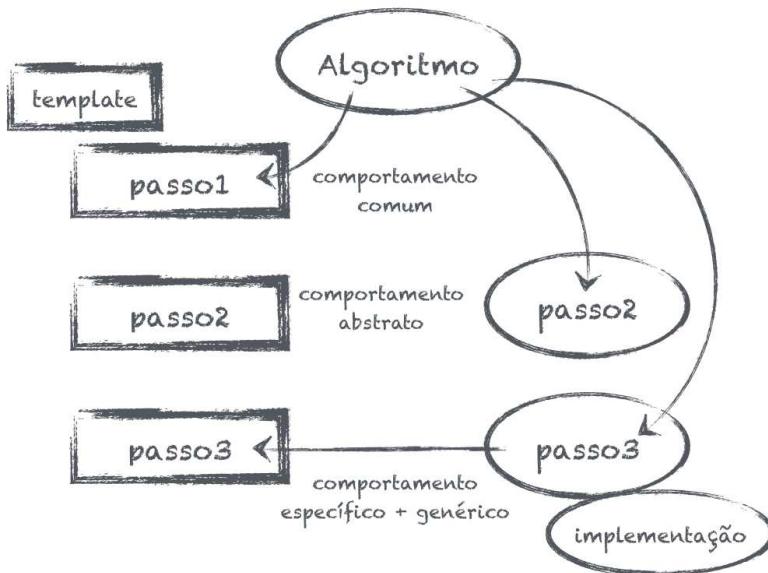


Fig. 6.2: Template e sua implementação

Pela figura, podemos perceber a flexibilidade da utilização do padrão:

- O `passo1` mostra um método comum a todas as implementações, portanto, é definido apenas no lado do `template`.
- Já o `passo2` é específico de cada algoritmo, sendo definido apenas na implementação.
- E o `passo3` mostra que é possível um comportamento misto, com uma parte comum a todos (definida no `template` e acessada via `super`), e outra específica de cada algoritmo (definida na implementação).

Para resolver o problema da Débora, ela cria a classe base `TemplateWorker` definindo os passos do algoritmo. No método `executar`, vamos chamar os ganchos que serão definidos pelas classes filhas. O teste desse método é simplesmente garantir que todos os ganchos sejam chamados:

```
#spec/template/template_worker_spec.rb
it 'deve executar os métodos especificados e logar erros' do
  e = StandardError.new
  mensagem_de_erro = "StandardError ao executar TemplateWorker"
  worker = TemplateWorker.new

  expect(Logger).to receive(:error).with(mensagem_de_erro)
  expect(worker).to receive(:trabalhar).with({}).and_raise(e)
  expect(worker).to receive(:deve_tentar_novamente).with(e, {})

  worker.executar({})
end
```

Como cada *worker* precisa de dados diferentes para funcionar, a implementação do método `executar` recebe um *hash* genérico com informações para as classes filhas e os métodos gancho. Além disso, é preciso coletar o resultado do método `trabalhar` para que ele seja retornado no final da execução do *worker*.

```
#lib/template/template_worker.rb
class TemplateWorker
  attr_reader :logger

  def executar(parametros)
    antes_execuacao(parametros)
    begin
      resultado = trabalhar(parametros)
    rescue Exception => e
      @logger.error("#{e.class} ao executar #{self.class}")
      retry if deve_tentar_novamente(e, parametros)
    end
    resultado
  end

  def antes_execuacao(parametros)
    @logger = Logger
  end
end
```

A classe `TemplateWorker`, além de definir o algoritmo, pode também

conter a lógica que será comum a todos os *workers*, como usar o *logger* para coletar as informações sobre falhas, ou a lógica para tentar novamente a execução em caso de falha.

Os testes da classe `EmailWorker` podem continuar os mesmos de antes, a única diferença é que agora eles devem chamar o método `executar`:

```
it 'deve mandar um email para cada destinatario' do
  id_usuario = 1
  lista_de_destinatarios = ['email@email.com',
                            'outro_email@email.com',
                            'email_qualquer@email.com']
  assunto = 'convite enviado por Usuario 1'
  email = EmailWorker.new.executar(
    usuario: id_usuario,
    destinatarios: lista_de_destinatarios,
    mensagem: :convite)
  expect(email[:para]).to eq(lista_de_destinatarios)
  expect(email[:assunto]).to eq(assunto)
  expect(email[:emails_enviados]).to eq(3)
end
```

Com a utilização dos métodos ganchos, a lógica do antigo método `enviar` será dividida, utilizando vários **Extrair Método**. O método `trabalhar` vai conter a execução principal do *worker*, agrupando informações e chamando o método `enviar_email`:

```
#lib/template/email_worker.rb
class EmailWorker < TemplateWorker
  def trabalhar(parametros)
    usuario = buscar_usuario(parametros[:usuario])
    corpo = gerar_mensagem(parametros[:mensagem],
                           usuario)
    assunto = gerar_assunto(parametros[:mensagem],
                           usuario)

    timeout(@limite_timeout) do
      enviar_email(de: usuario,
                  para: parametros[:destinatarios],
                  assunto: assunto,
                  ...)
```

```
        corpo: corpo)
    end
end
end
```

No método gancho `antes_execuacao`, será feita a configuração específica do *worker* como limites de tempo de espera, tentativas etc.

```
#lib/template/email_worker.rb
class EmailWorker < TemplateWorker
  def antes_execuacao(parametros)
    super(parametros)
    @limite_timeout = 10
    @limite_tentativas = 5
    @contador_tentativas = 0
  end
end
```

Já no método `deve_tentar_novamente`, vamos adicionar a lógica para executar o método novamente, caso o `@contador_tentativas` ainda não tenha ultrapassado o `@limite_tentativas`.

```
#lib/template/email_worker.rb
class EmailWorker < TemplateWorker
  def deve_tentar_novamente(e, parametros)
    @contador_tentativas += 1
    @contador_tentativas < @limite_tentativas
  end
end
```

Com o algoritmo bem definido, adicionar novos *workers* será apenas uma questão de encaixar sua lógica nos ganchos, pois a execução principal se mantém a mesma. Caso o novo *worker* precise de um novo gancho, basta definir um método vazio no `TemplateWorker` para que os *workers* já existentes não precisem ser modificados.

Débora precisa que o *worker* para baixar imagens seja executado novamente sempre que uma exceção do tipo `Timeout::Error` for lançada, independente do número de tentativas já realizadas. Apesar de a implemen-

tação inicial do `TemplateWorker` não possuir um tratamento especial de exceções, ela pode resolver o problema facilmente.

O primeiro passo é adicionar o novo método gancho `tratar_excecao` no método `TemplateWorker.executar`, logo no início do bloco `rescue`. Esse gancho receberá o objeto da exceção que foi lançada para executar sua lógica.

```
#lib/template/template_worker.rb
class TemplateWorker
  def executar(parametros)
    antes_execuacao(parametros)
    begin
      resultado = trabalhar(parametros)
    rescue Exception => e
      tratar_excecao(e, parametros)
      @logger.error("#{e.class} ao executar #{self.class}")
      retry if deve_tentar_novamente(e, parametros)
    end
    resultado
  end
end
```

No entanto, Débora percebe que simplesmente adicionar um novo método gancho vai causar problemas em *workers* já existentes, nos quais o método não foi definido. O problema não é apenas definir o método em classes já existentes – nem todos os *workers* precisam tratar exceções –, então, é necessário fazer com que esse método seja opcional.

Para resolver o problema, Débora define o método `tratar_excecao` no `TemplateWorker` com uma implementação vazia:

```
#lib/template/template_worker.rb
class TemplateWorker
  def tratar_excecao(e, parametros); end
end
```

Assim, além de ser fácil adicionar novos algoritmos para *workers*, também é fácil realizar mudanças no algoritmo base. Incorporar novos requisitos à solução fica bem simples.

## CAPÍTULO 7

# Adapter: seja como a água

Quando tomamos a decisão de aposentar um sistema, ou um conjunto de sistemas, os dados antigos são uma grande preocupação. Achar uma maneira de utilizá-los é muitas vezes essencial, mas é preciso evitar que a nova aplicação fique presa ao design da anterior.

Essa é a situação na qual Celso se encontra, ele está à frente de uma equipe criando uma aplicação que vai mapear informações sobre **fornecedores**, **estoque** e **clientes** de uma grande loja online. A ideia principal é substituir um conjunto de aplicações por uma única, centralizando as informações e facilitando a vida dos atendentes que precisam abrir múltiplas janelas para buscar partes da informação.

## 7.1 CAOS E ORDEM

Celso olha mais uma vez para a documentação dos sistemas antigos pensando em como orquestrar todas essas informações. A aplicação que controla o **estoque** guarda todas as informações em uma base de dados não relacional. Já a que contém informações sobre os **fornecedores** utiliza um serviço de mensagens em fila. Por fim, as informações sobre **clientes** são exposta em uma API *soap* trocando arquivos XML (<http://pt.wikipedia.org/wiki/SOAP>) .

Para a primeira grande entrega do novo sistema, será permitido acessar e editar as informações dos clientes que estão acessíveis utilizando a API *soap* do sistema legado. A classe `Cliente` foi criada para obter essas informações, e possui um atributo chamado `id_universal`, que é um código usado para identificar clientes.

```
#lib/adapter/cliente.rb
Class Cliente
    attr_reader :id_universal

    def initialize(id_universal)
        @id_universal = id_universal
    end
end
```

No sistema antigo, as preferências eram divididas de acordo com seu propósito, por exemplo, preferências de notificações por e-mail ou quais os endereços para entregas. Porém, nessa nova versão, todas elas serão exibidas em uma mesma tela, logo, Celso decide obter todas as preferências e guardar no mesmo objeto.

O método que obtém as preferências de um cliente vai executar uma série de chamadas ao serviço externo, coletar cada pedaço da informação e depois agrupá-las em um único *hash* (no método `mapear_para_hash`). Por fim, o método converte o *hash* para `JSON`, pois o novo sistema oferecerá uma API para acessar os dados.

```
#lib/adapter/cliente.rb
class Cliente
    def preferencias
```

```
email_xml = obter_preferencias_de_email || ''
email = Validar.email(email_xml).split(', ')

endereco_xml = obter_preferencias_de_endereco || ''
endereco = Validar.endereco(endereco_xml)

pagamento_xml = obter_preferencias_de_pagamento || ''
pagamento = Validar.pagamento(pagamento_xml)

telefone_xml = obter_preferencias_de_telefone || ''
telefone = Validar.telefone(telefone_xml).split(', ')

preferencias =
  mapear_para_hash(email, endereco, pagamento, telefone)
  preferencias.to_json
end
end
```

Construir o *hash* é relativamente fácil, basta adicionar os dados nas chaves apropriadas:

```
#lib/adapter/cliente.rb
class Cliente
  def mapear_para_hash(email, endereco, pagamento, telefone)
    {
      email: email,
      endereco: endereco,
      pagamento: pagamento,
      telefone: telefone
    }
  end
end
```

Os métodos `obter_preferencias_*` não apenas executar a chamada ao serviço *soap* externo, e não precisam ser detalhados aqui. As partes internas dos métodos da classe `Validar` também são irrelevantes para o exemplo. A questão maior do problema é a definição do contrato entre o sistema legado e o novo.

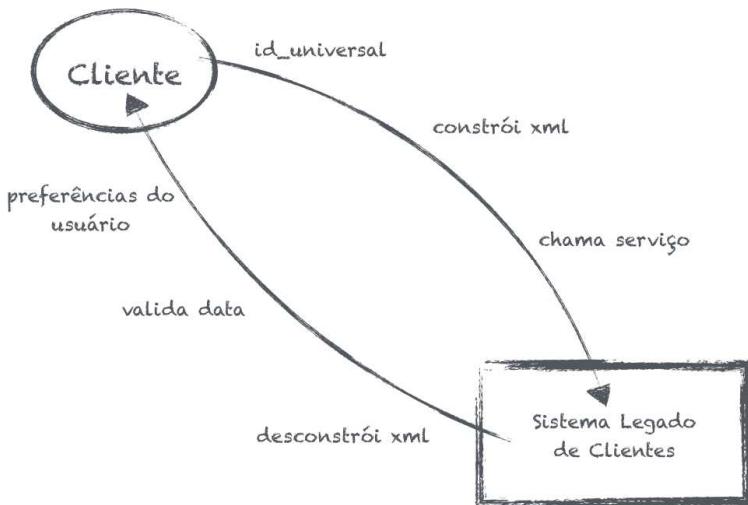


Fig. 7.1: Contrato entre Cliente e o serviço legado

Nos testes para o método `preferencias`, Celso utilizou *stubs* para que as chamadas externas não influenciem na execução.

```
#spec/adapter/cliente_spec.rb
it 'retorna a lista de emails do cliente' do
  id_universal = 'FG1234'
  cliente = Cliente.new(id_universal)
  emails = 'cliente@cliente.com, cliente@email.com'
  allow(cliente).to
    receive(:obter_preferencias_de_email) { emails }
  preferencias = JSON.parse(cliente.preferencias)
  emails = preferencias["email"]
  expect(emails.size).to eq(2)
  expect(emails[0]).to eq('cliente@cliente.com')
  expect(emails[1]).to eq('cliente@email.com')
end
```

Apesar de o método `preferencias` delegar ações para vários métodos, ele ainda concentra muita lógica de maneira bem útil. Note as pequenas

verificações de valor nulo, como `obter_preferencias_de_pagamento || ''`, e as chamadas de `split(',', ' )` para separar *string* em um *array*. Mesmo que simples, ainda são lógicas para transformar os dados recebidos do sistema legado.

Celso não consegue parar de pensar em quão doloroso será quando o sistema antigo for aposentado e todo o código já feito tiver de ser alterado. Ele sabe que o código vai ser alterado, mas não como e nem quando! Pode ser que, no futuro, os dados de preferências sejam lidos em um banco de dados local, ou talvez seja necessário utilizar tanto o serviço externo quanto o banco local ao mesmo tempo!

A ideia que Celso vai tentar aplicar é definir uma interface intermediária para obtenção de dados. Dessa forma, obtê-los será feito independentemente de como serão utilizados, e eles estarão em classes separadas e compartilhadas. Essa camada de flexibilidade intermediária pode ser implementada seguindo o padrão Adapter!

## 7.2 O PADRÃO ADAPTER

O problema resolvido pelo Adapter é quando temos duas classes com interfaces diferentes, mas que precisam trabalhar em conjunto. A solução é isolar as conversões de uma interface para a outra, do resto da lógica de negócios. Para aplicar bem a solução, é preciso que o contexto de uso do padrão permita separar bem a transformação dos dados do resto da lógica.

Utilizar o Adapter vai definir um contrato com o que é esperado pela classe cliente e com o que o adaptador precisa definir. Criar novos adaptadores não requer integração com o cliente, basta que o dado seja retornado na maneira esperada.

A figura a seguir mostra como o Adapter introduz uma camada intermediária para deixar a classe `Cliente` mais leve e com menos responsabilidades:

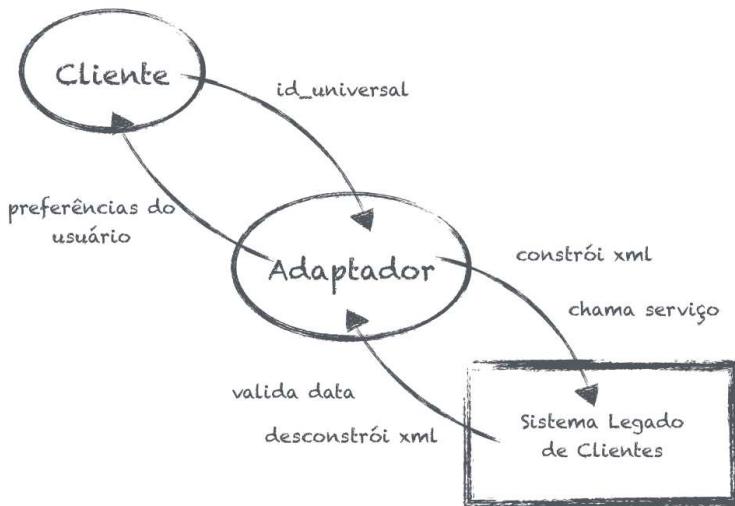


Fig. 7.2: O Adaptador é quem define o contrato com Cliente

O contrato é definido no método `preferencia`, que utiliza o `id_universal` e retorna as informações de e-mail, endereço, pagamento e telefone em um único `hash`. O primeiro passo de Celso é usar **Extrair Método** para separar a responsabilidade de fazer a requisição *soap*.

```
#lib/adapter/cliente.rb
class Cliente
  def preferencias
    preferencias = requisitar_preferencias
    preferencias.to_json
  end

  def requisitar_preferencias
    email_xml = obter_preferencias_de_email || ''
    email = Validar.email(email_xml).split(', ')
    endereco_xml = obter_preferencias_de_endereco || ''
    endereco = Validar.endereco(endereco_xml)
  end
end
```

```
pagamento_xml = obter_preferencias_de_pagamento || ''
pagamento = Validar.pagamento(pagamento_xml)

telefone_xml = obter_preferencias_de_telefone || ''
telefone = Validar.telefone(telefone_xml).split(', ')

mapear_para_hash(email, endereco, pagamento, telefone)
end
end
```

Com a separação feita, fica fácil saber o que deve ser movido para o adaptador *soap*. Utilizamos **Extrair Classe** em `Cliente`, e criamos a classe `AdaptadorSoap`, que precisa apenas do `id_universal` de um `Usuario` para realizar as requisições.

```
#lib/adapter/adaptador_soap.rb
class AdaptadorSoap
  attr_accessor :id_universal

  def initialize(id_universal)
    @id_universal = id_universal
  end

  def preferencias
    email_xml = obter_preferencias_de_email || ''
    email = Validar.email(email_xml.split(', '))

    endereco_xml = obter_preferencias_de_endereco || ''
    endereco = Validar.endereco(endereco_xml)

    pagamento_xml = obter_preferencias_de_pagamento || ''
    pagamento = Validar.pagamento(pagamento_xml)

    telefone_xml = obter_preferencias_de_telefone || ''
    telefone = Validar.telefone(telefone_xml.split(', '))

    mapear_para_hash(email, endereco, pagamento, telefone)
  end
```

```
end
```

Uma boa ideia seria copiar os testes da classe Cliente para AdaptadorSoap, já que agora a lógica se encontra na classe adaptador:

```
#spec/adapter/adaptador_soap_spec.rb
it 'retorna a lista de emails do cliente' do
  id_universal = 'FG1234'
  adaptador = AdaptadorSoap.new(id_universal)
  adaptador.id_universal = id_universal
  emails = 'cliente@cliente.com, cliente@email.com'
  allow(adaptador).to
    receive(:obter_preferencias_de_email) { emails }
  emails = adaptador.preferencias["email"]
  expect(emails.size).to eq(2)
  expect(emails[0]).to eq('cliente@cliente.com')
  expect(emails[1]).to eq('cliente@email.com')
end
```

## RUBY E CLASSES ABSTRATAS

Ruby não possui o conceito de classes abstratas com o qual definimos métodos e “obrigamos” a sua implementação pelas classes herdeiras. Algumas implementações tentam forçar essa obrigação utilizando métodos que apenas lançam exceções:

```
class ClasseAbstrata
  def metodo_abstrato
    raise "Metodo deve ser implementado em classes filhas"
  end
end
```

A vantagem dessa implementação é que fica bem documentado no código qual é o contrato. O problema é que este é redundante, pois, ao tentar executar um método que não existe, o Ruby já lança uma exceção:

```
objeto.metodo_inexistente # => Lança NoMethodError
```

O código final da classe `Cliente` fica bem mais simples agora que a única preocupação é delegar a obtenção dos dados para o adaptador, e depois transformar esse *hash* em um objeto `JSON`:

```
#lib/adapter/cliente.rb
class Cliente
  attr_reader :id_universal, :adaptador

  def initialize(id_universal, adaptador)
    @id_universal = id_universal
    @adaptador = adaptador
  end

  def preferencias
    preferencias = @adaptador.preferencias
    preferencias.to_json
  end
end
```

Feito isso, Celso pode remover o código da classe cliente bem como seus testes, pois o foco da classe `Cliente` é apenas na transformação para `JSON`. Continuamos utilizando *stubs* nos testes para retornar um *hash* específico, mas aplicados ao adaptador.

```
#spec/adapter/cliente_spec.rb
it 'retorna a lista de emails do cliente' do
  id_universal = 'FG1234'
  adaptador = AdaptadorSoap.new(id_universal)
  cliente = Cliente.new(id_universal, adaptador)
  preferencias = {
    email: ['cliente@cliente.com'],
    endereco: 'rua 0 casa 1',
    pagamento: '1234 5678 8765 4321',
    telefone: ['+0123456789']
  }
  allow(adaptador).to receive(:preferencias) { preferencias }
  preferencias = JSON.parse(cliente.preferencias)
  expect(preferencias["email"][0]).to eq('cliente@cliente.com')
  expect(preferencias["endereco"]).to eq('rua 0 casa 1')
```

```
expect(preferencias["pagamento"]).to eq('1234 5678 8765 4321')
expect(preferencias["telefone"])[0]).to eq('+0123456789')
end
```

Agora, caso seja necessário ler os dados de qualquer outra forma, o contrato entre a classe `Cliente` e os adaptadores está bem claro: basta que o adaptador receba o `id_universal` e retorne as preferências de e-mail, endereço, pagamento e telefone em um *hash*, e tudo funcionará sem problemas.

Celso fica feliz com o resultado final da refatoração e mostra para o resto do time o quanto fácil será manter o código da aplicação, já que a comunicação com o serviço externo está isolada. Caso essa comunicação precise mudar, também será algo fácil de fazer, basta seguir o contrato.

## **Parte III**

### **Padrões de projeto situacionais**



## CAPÍTULO 8

# State: 11 estados e 1 objeto

Guilherme é um desenvolvedor que conseguiu o emprego dos seus sonhos: desenvolver jogos! Seu primeiro projeto é um jogo de plataforma no qual Maria, a personagem principal, é uma encanadora que foi parar em um mundo surreal para salvar o príncipe do reino.

Ao longo de sua jornada, Maria vai utilizar vários itens que lhe darão novos poderes para enfrentar os desafios. No entanto, o desafio real será Guilherme conseguir modelar o código de uma maneira que facilite as várias mudanças no comportamento da heroína.

### **8.1 MARIA E SEUS PODERES**

Segundo as especificações do jogo, Maria começa-o em um formato pequeno, sem poderes especiais, e pode ir pegando itens que lhe darão novos poderes ao longo dele. Ao colidir com um inimigo, ela leva dano e perde seus poderes;

caso não tenha nenhum, morre e perde o jogo. A figura a seguir mostra as transições da personagem de acordo com os acontecimentos do jogo:

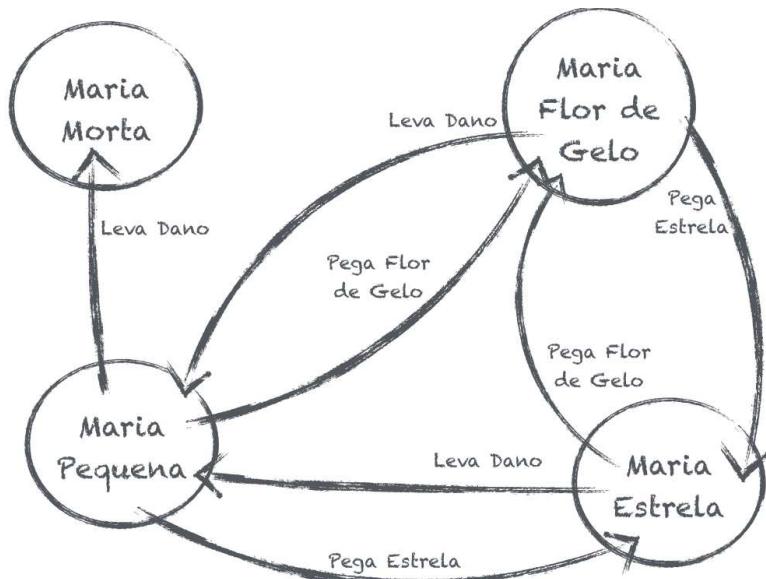


Fig. 8.1: Ações e itens de Maria

Com base nesses requisitos iniciais, Guilherme começou uma implementação básica do comportamento da personagem na classe `Maria`, criando constantes dentro da classe para identificar qual seu estado atual.

```

#lib/state/maria.rb
class Maria
  attr_reader :estado_atual

  # estados
  PEQUENA = :pequena
  FLOR_DE_GELO = :flor
  ESTRELA = :estrela
  MORTA = :morta

  def initialize
  
```

```
  @estado_atual = Maria::PEQUENA
end
end
```

Ainda na classe `Maria`, Guilherme definiu as ações da personagem: pegar a flor de gelo, levar dano de um inimigo e pegar estrela.

```
#lib/state/maria.rb
Class Maria
  def pegar_flor_de_gelo
    return if @estado_atual == Maria::ESTRELA
    @estado_atual = Maria::FLOR_DE_GELO
  end

  def pegar_estrela
    @estado_atual = Maria::ESTRELA
  end

  def levar_dano
    return if @estado_atual == Maria::ESTRELA
    if @estado_atual == Maria::PEQUENA
      @estado_atual = Maria::MORTA
    else
      @estado_atual = Maria::PEQUENA
    end
  end
end
```

Um exemplo dos testes que Guilherme escreveu é criar um novo objeto `Maria`, chamar o método `flor_de_gelo` e garantir que o estado atual seja `Maria::FLOR_DE_GELO`. Veja o código:

```
#spec/state/maria_spec.rb
context 'quando o estado atual é pequena' do
  it 'pega flor de gelo e muda o estado para flor de gelo' do
    maria = Maria.new
    maria.pegar_flor_de_gelo
    expect(maria.estado_atual).to eq(Maria::FLOR_DE_GELO)
  end
end
```

Com os requisitos atuais, a classe `Maria` está relativamente simples. Tudo bem que existem vários `ifs`, principalmente no método `levar_dano`, mas nada que doa muito.

Após uma demonstração para um pequeno grupo de usuários, a equipe do jogo viu que os itens da personagem Maria tornaram o jogo bem mais divertido. Assim, eles decidem adicionar novos itens que Maria pode utilizar ao longo de sua jornada. Ao ouvir isso, Guilherme já começa a pensar na classe `Maria`, e nas novas constantes e `ifs` que precisam mudar.

Depois de alguma discussão, o time decidiu incrementar o jogo da seguinte forma: Maria agora pode pegar o item flor de fogo, que tem comportamento bem semelhante ao item flor de gelo. A figura mostra como seriam as transições de estados considerando apenas o novo item:

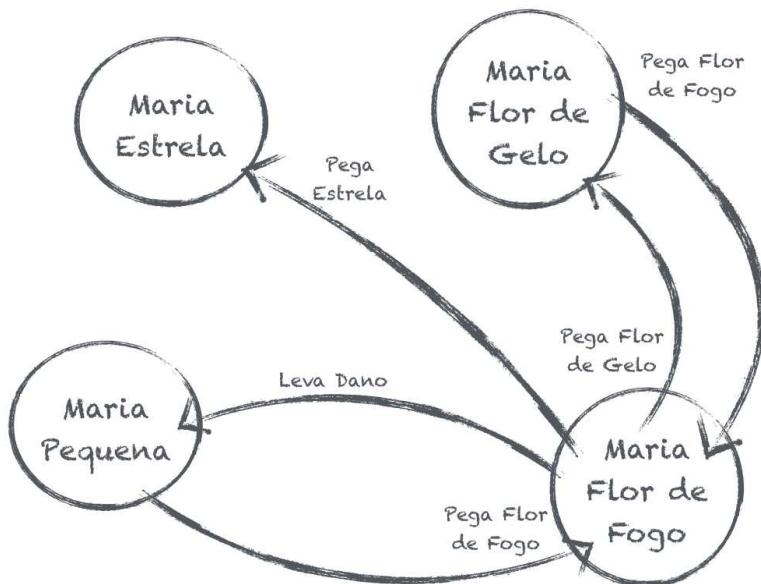


Fig. 8.2: Adicionando novo item e ações para pegar flor de fogo

Codificar mais essas ações faria com que aquele grupo pequeno de `ifs` se tornasse um inferno de manutenção. Guilherme sugeriu para a equipe que, em vez de apenas uma classe fazer tudo, tentassem dividir o código criando

uma classe para lidar com cada estado que Maria pode assumir.

Outra vantagem que Guilherme apresenta é que as regras de transição ficam definidas nos próprios estados, assim, ao executar uma ação, cada estado sabe qual deve ser o próximo. Isso evitaria que a lógica ficasse concentrada apenas em um lugar, acumulando vários `ifs`. A solução proposta é o padrão State!

## 8.2 O PADRÃO STATE

Pense no problema como sendo uma máquina de estados, em que cada habilidade (estrela, flor de gelo etc.) é um estado, e cada ação (pegar um flor de gelo, levar dano etc.) é uma transição entre estados. Ao utilizar este padrão, buscamos uma maneira de compartilhar a responsabilidade de trocar estados, tratando cada um como uma classe e cada transição como um método.

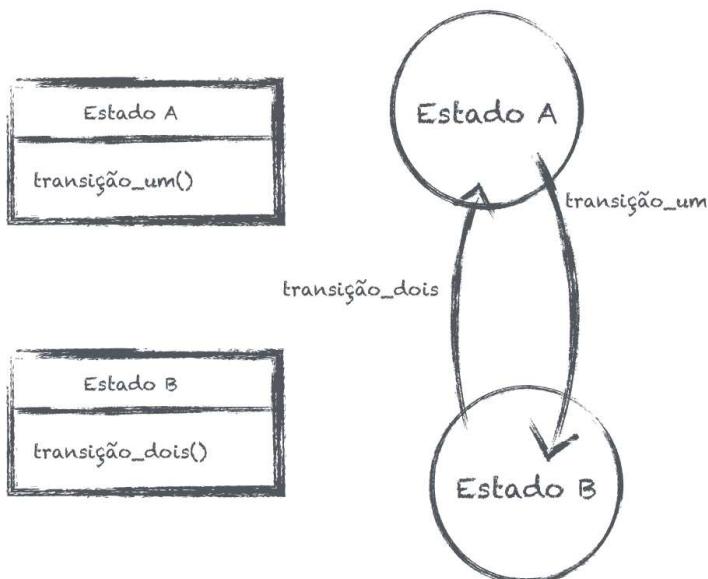


Fig. 8.3: Como estados são modelados em classes

Como os outros padrões de comportamento, o State também busca sim-

plificar o relacionamento entre classes. O contexto de utilização é quando existe um conjunto de comportamentos que precisa mudar constantemente, não sendo necessárias ações externas. A solução é separar as funcionalidades em classes diferentes e deixar que, em vez de o cliente fazer mudanças, cada estado saiba para qual estado deve mudar.

Cada estado precisa implementar os mesmo métodos de `Maria`, mas apenas com sua lógica específica. Por exemplo, para o código do estado Pequena, Guilherme usa **Extrair Classe** para copiar toda a lógica pertencente ao estado do código da classe `Maria`:

```
#lib/state/pequena.rb
class Pequena
  def pegar_flor_de_gelo
    FlorDeGelo.new
  end

  def pegar_estrela
    Estrela.new
  end

  def levar_dano
    Morta.new
  end
end
```

Para inserir o novo estado, em vez de utilizar a constante `Maria::PEQUENA`, a classe `Maria` vai ter uma instância da classe Pequena recém-criada:

```
#lib/state/maria.rb
class Maria
  attr_reader :estado_atual
  def initialize
    @estado_atual = Pequena.new
  end
end
```

Agora, Guilherme precisa usar o estado `Pequena` no resto do código

da classe `Maria`. Vamos seguir, então, as seguintes regras para substituir o símbolo `Maria::PEQUENA` por objetos da classe `Pequena`:

- Em vez de comparar com a constante `Maria::PEQUENA`, precisamos verificar se `@estado_atual` é um objeto do tipo `Pequena`.
- Se o `@estado_atual` for `Pequena`, vamos apenas delegar a ação para o próprio `@estado_atual`.
- Onde antes iríamos atribuir a constante `Maria::PEQUENA`, vamos utilizar uma instância da classe `Pequena`.

Veja como ficaria o método `levar_dano` misturando as constantes existentes e o novo estado `Pequena`:

```
#lib/state/maria.rb
Class Maria
  def levar_dano
    return if @estado_atual == Maria::ESTRELA
    if @estado_atual.is_a?(Pequena)
      @estado_atual = @estado_atual.levar_dano
    else
      @estado_atual = Pequena.new
    end
  end
end
```

Note que, como temos apenas um estado implementado, as constantes precisam continuar lá para que a classe continue funcionando durante nossa refatoração. Ruby não se importa muito com o tipo de uma variável, então, podemos atribuir símbolos ou estados à variável `@estado_atual` que o código funciona sem problemas.

Como cada método retorna o próximo estado, após Guilherme implementar todos os estados, a classe `Maria` não precisa mais fazer verificações como no código anterior `@estado_atual.is_a?(Pequena)`. Ela vai apenas delegar para o seu `@estado_atual` as chamadas das ações, e atualizá-lo com o resultado da ação.

```
#lib/state/maria.rb
class Maria
  attr_reader :estado_atual

  def initialize
    @estado_atual = Pequena.new
  end

  def pegar_flor_de_gelo
    @estado_atual = @estado_atual.pegar_flor_de_gelo
  end

  def pegar_estrela
    @estado_atual = @estado_atual.pegar_estrela
  end

  def levar_dano
    @estado_atual = @estado_atual.levar_dano
  end
end
```

A ideia é que, a cada transição, Maria atualize seu estado atual. O novo estado, por sua vez, saberá o que fazer quando outra transição ocorrer – bem semelhante a uma máquina de estados.

Como a classe cliente `Maria` continua seguindo a mesma interface, os testes continuam instanciando os mesmos objetos e chamando os mesmos métodos. A única diferença que Guilherme precisa fazer é, como `estado_atual` é uma objeto que representa o estado, vamos comparar a classe dele em vez de comparar símbolos:

```
#spec/state/maria_spec.rb
it 'pega flor de gelo e muda o estado para flor de gelo' do
  maria = Maria.new
  maria.pegar_flor_de_gelo
  expect(maria.estado_atual).to be_kind_of(FlorDeGelo)
end
```

Para implementar o novo item flor de fogo, é preciso criar a ação `pegar_flor_de_fogo` em todos os estados. A implementação segue o

mesmo raciocínio das já existentes: muda-se o `@estado_atual` para o novo estado `FlorDeFogo` e ele lidará com novas mudanças. Veja como ficaria o teste:

```
#spec/state/maria_spec.rb
context 'quando o estado atual é pequena' do
  it 'pega flor de fogo e muda para flor de fogo' do
    maria = Maria.new
    maria.pegar_flor_de_fogo
    expect(maria.estado_atual).to be_kind_of(FlorDeFogo)
  end
end
```

A implementação do novo estado `FlorDeFogo` será tão simples quanto as anteriores. Na ação `pegar_flor_de_fogo`, como já estamos no estado `FlorDeFogo`, nada acontece e não precisamos instanciar nenhum novo objeto.

```
#lib/state/flor_de_fogo.rb
class FlorDeFogo
  def pegar_flor_de_gelo
    FlorDeGelo.new
  end

  def pegar_estrela
    Estrela.new
  end

  def levar_dano
    Pequena.new
  end

  def pegar_flor_de_fogo
    self
  end
end
```

Por fim, como Guilherme precisou adicionar uma nova transição (`pegar a flor de fogo`), precisamos defini-la nos estados já existentes de acordo com as novas regras:

```
#lib/state/pequena.rb
class Pequena
  def pegar_flor_de_fogo
    FlorDeFogo.new
  end
end
```

## STRATEGY E STATE

Se você reparar bem, o padrão Strategy é bem parecido com o State: ambos vão dividir uma lógica complexa dentro de objetos próprios. Ao avaliar o contexto para decidir qual padrão pode ajudá-lo melhor, note as seguintes características de cada um deles:

- No **Strategy**, uma vez definida a estratégia, ela não vai mudar naquela execução.
- No **State**, cada estado sabe suas transições, então, o cliente não garante o estado atual.

Para avaliar a utilização dos padrões em um problema, basta se perguntar o seguinte:

- **Strategy**: a estratégia precisa mudar uma vez que ela foi definida? Se sim, os vários `ifs` provavelmente vão voltar a se espalhar pelo código para avaliar um resultado e definir uma nova estratégia.
- **State**: o cliente precisa ficar verificando qual o estado atual o tempo todo? Se sim, a vantagem de não ter os `ifs` espalhados pelo código desaparece completamente.

Após a sessão de refatoração, Guilherme olha para o código com bastante orgulho por sua sugestão ter ajudado a descomplicar o problema. No final das contas, o time ficou com bem mais classes do que na visão inicial da solução,

mas cada uma ficou extremamente simples, com métodos de apenas uma linha e sem nenhum `if`! Até mesmo aquela lista de constantes no começo da classe `Maria` desapareceu.



## CAPÍTULO 9

# Builder: construir com classe

Ana é uma experiente desenvolvedora Ruby que já trabalhou em vários projetos, principalmente com *startups*. As aplicações, na maior parte, eram pequenas e simples, mas mesmo assim seguiam boas práticas, como testes automatizados e integração contínua. No entanto, ao se mudar para uma nova cidade, ela começou a trabalhar em uma aplicação de uma grande empresa de venda de carros, um cenário bem diferente para ela.

Falando em aplicações corporativas, classes grandes são um problema muito difícil de ser atacado. Geralmente, elas têm muitos atributos e possuem muita lógica associada. Contudo, em alguns casos, é realmente necessário que a classe possua uma quantidade razoável de atributos. Ana acabou de entrar em uma equipe que está construindo uma aplicação para negociação de carros que sofre desse problema.

Existe uma grande quantidade de informação associada aos carros: ano de fabricação, cor, modelo, fabricante etc. Como era de se esperar, existe uma

classe `Carro` para representar essas informações, que contém também toda a lógica relacionada a elas.

## 9.1 MUITA INFORMAÇÃO EM UM SÓ LUGAR

Por ter entrado há pouco tempo, Ana consegue analisar o código sem os “costumes” dos outros integrantes da equipe. Ao olhar para a famosa classe `Carro`, ela não gosta muito da quantidade de atributos ali acumulados. Criar um objeto `Carro` é bem trabalhoso e demanda muitos parâmetros.

```
#lib/builder/carro.rb
class Carro
    attr_reader :modelo, :fabricante, :ano_fabricacao, :placa,
                :cor, :km_rodados, :ano_modelo, :preco_minimo,
                :preco_anunciado

    def initialize(modelo, fabricante, ano_fabricacao, placa,
                  cor, km_rodados, ano_modelo, preco_minimo,
                  preco_anunciado)
        @modelo = modelo
        @fabricante = fabricante
        @ano_fabricacao = ano_fabricacao
        @ano_modelo = ano_modelo
        @placa = placa
        @cor = cor || ''
        @km_rodados = km_rodados || 0
        @preco_minimo = preco_minimo || 0
        @preco_anunciado = preco_anunciado || 0
    end
end
```

Olhando apenas para os nomes dos atributos, Ana começa a pensar em possibilidades de extraí-los em classes próprias; por exemplo, `ano_fabricacao` e `ano_modelo` poderiam ter sua própria classe. Apesar de isso simplificar a criação, diminuindo a quantidade de parâmetros no construtor de `Carro`, ainda serão necessários os mesmos dados para criar instâncias.

Outro grande sintoma desse problema no código existente são os testes para essa classe. Qualquer teste que precise utilizar objetos do tipo `Carro` vai precisar passar muita informação, ou passar vários dados “lixo” (`nil`, ‘` etc.). Veja, por exemplo, esse teste que verifica a validação de que o ano do modelo não pode ser inferior ao de fabricação:

```
#spec/builder/carro_spec.rb
it 'nao pode ter ano de modelo menor que ano de fabricacao' do
  ano_fabricacao = 2000
  ano_modelo = 1999
  carro_invalido = Carro.new('modelo A', 'fabricante A',
                               ano_fabricacao, 'ABC1234', nil, nil,
                               ano_modelo, nil, nil)
  erro =
    'ano do modelo nao pode ser anterior ao ano de fabricacao'
  expect(carro_invalido.validar!).to eq(false)
  expect(carro_invalido.erros.size).to eq(1)
  expect(carro_invalido.erros.first).to eq(erro)
end
```

O problema é que alguns daqueles atributos são de fato necessários para criar um novo objeto, e até existem validações para eles! O efeito colateral aqui é que vários testes precisam passar informações que não têm a menor utilidade para o caso de teste (como ‘ABC1234’), dificultando entender o seu real objetivo.

Observe a quantidade de informação presente no teste a seguir, que valida se a placa está presente, e tente descobrir o que é passado como placa no construtor do `Carro`:

```
#spec/builder/carro_spec.rb
it 'deve ter uma placa especificada' do
  carro_invalido = Carro.new('modelo A', 'fabricante A', nil,
                             nil, nil, nil, nil, nil, nil)
  erro = 'placa nao pode ser nulo'
  expect(carro_invalido.validar!).to eq(false)
  expect(carro_invalido.erros.size).to eq(1)
  expect(carro_invalido.erros.first).to eq(erro)
end
```

Ana pensa um pouco e avalia a utilização do padrão Simple Factory para separar a responsabilidade de criação, mas logo muda de ideia, pois as possibilidades de combinações de dados são muito grandes. A classe fábrica precisaria definir muitos métodos ou receber muitos parâmetros, o que seria uma lógica semelhante à do construtor atual, apenas escondida em um método.

O que ela realmente gostaria de fazer é definir um conjunto de valores padrão para os campos obrigatórios, mas ainda assim permitir que eles fossem sobreescritos. Assim, seria simples adicionar apenas os dados necessários para a situação em mãos, como as informações de ano no teste anterior. É, então, que ela se depara com o padrão *Builder*, que pode ser exatamente o que ela procura!

## 9.2 O PADRÃO BUILDER

Por ser classificado como um padrão de criação, o *Builder* resolve o mesmo problema dos padrões *Factory*: separar a criação de objetos da lógica de negócio. No entanto, o contexto de sua utilização é quando o processo de criação precisa de muitos atributos diferentes, ou precisa ser mais flexível do que apenas chamar um método.

A solução é definir uma classe para criar o objeto por partes, assumindo um valor padrão para atributos não definidos pelo cliente. Dessa forma, garantimos que, ao final do processo, criamos um objeto válido e com as informações necessárias.

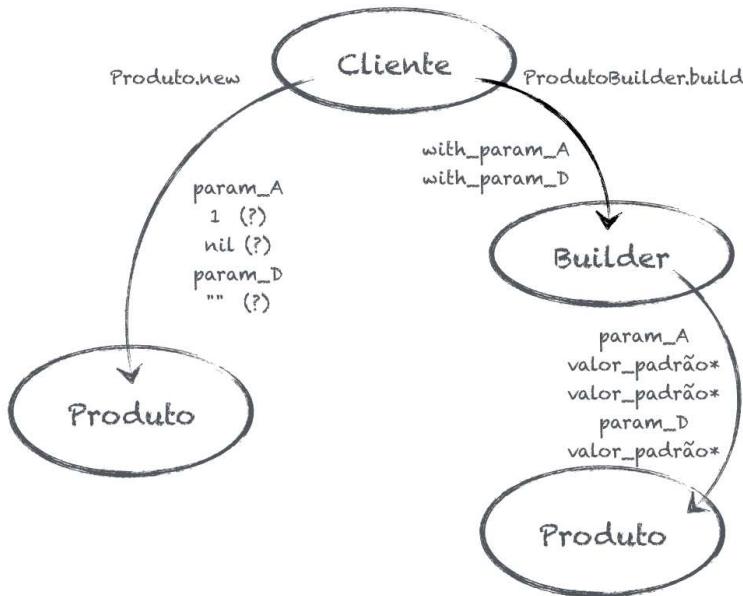


Fig. 9.1: Chamada do construtor vs. Builder

Nessa figura, chamar o construtor diretamente exige que passemos parâmetros que não são importantes para o cliente. Em alguns casos, será necessário até mesmo criar objetos apenas para que o método a ser chamado funcione sem nenhuma exceção ser lançada. Usando o padrão, toda essa preocupação fica por conta do Builder, que apenas retorna o objeto com os valores de interesse do cliente.

## BUILDER E FACTORY

Os padrões *Builder* e *Factory* são bem parecidos. Para decidir quando utilizar um ou o outro, pense apenas no processo de criação:

- Se o objeto a ser criado tem poucos atributos ou não precisa de muitas opções, utilize o Factory Method ou mesmo Simple Factory;
- Se existem muitos atributos a serem utilizados, ou é necessário dar muitas opções para criar o objeto, utilize o Builder.

Os padrões Simple Factory e Factory Method são os básicos para criação de objetos, pois apenas definem um método para instanciar objetos. Geralmente, eles serão a primeira opção, pois resolvem o problema de maneira bem simples. É conforme o projeto evolui que podemos ver a necessidade de deixar o processo de criação mais flexível; logo, o Builder pode ser uma boa opção.

Ana cria a classe `CarroValidoBuilder`, e define nela um conjunto padrão de valores que um carro deve ter para ser válido. O teste para especificar esse comportamento seria instanciar o Builder – sem definir nenhum valor para os atributos do carro –, criar a instância de `Carro` e verificar se ela é válida:

```
#spec/builder/carro_valido_builder_spec.rb
context 'com valores padrão' do
  it 'deve criar um carro válido' do
    builder = CarroValidoBuilder.new
    carro = builder.construir_carro
    expect(carro.validar!).to eq(true)
  end
end
```

Os atributos padrão do carro serão definidos no construtor do `CarroValidoBuilder`, para que eles sejam atribuídos a uma nova instância de `Carro` no método `construir_carro`.

```
#lib/builder/carro_valido_builder.rb
class CarroValidoBuilder
    attr_reader :modelo, :fabricante, :ano_fabricacao, :placa,
                :cor, :km_rodados, :ano_modelo, :preco_minimo,
                :preco_anunciado

    def initialize
        @modelo = 'Modelo A123'
        @fabricante = 'Fabricante ABCD'
        @placa = 'ABC1234'
        @ano_fabricacao = 2000
        @ano_modelo = 2001
    end

    def construir_carro
        Carro.new(@modelo, @fabricante, @ano_fabricacao, @placa,
                  @cor, @km_rodados, @ano_modelo, @preco_minimo,
                  @preco_anunciado)
    end
end
```

Para deixar o `CarroValidoBuilder` mais flexível, serão adicionados métodos para configurar valores para os outros atributos. Um exemplo de teste seria criar um carro com o `CarroValidoBuilder`, modificar sua cor e quilometragem, e verificar os valores:

```
#spec/builder/carro_valido_builder_spec.rb
it 'deve configurar um carro com uma cor e quilometragem' do
    carro = CarroValidoBuilder.new
        .com_cor('azul')
        .com_quilometragem(100)
        .construir_carro
    expect(carro.cor).to eq('azul')
    expect(carro.km_rodados).to eq(100)
end
```

Encadear as chamadas dos métodos do Builder requer retornar o próprio Builder ao final de cada método de configuração. Ou seja, os métodos `com_*`

vão atribuir o valor para o atributo e retornar `self` para que novos métodos possam ser encadeados.

```
#lib/builder/carro_valido_builder.rb
class CarroValidoBuilder
  def com_cor(cor)
    @cor = cor
    self
  end

  def com_quilometragem(km_rodados)
    @km_rodados = km_rodados
    self
  end
end
```

Essa técnica de encadear chamadas de métodos de um mesmo objeto se chama Interface Fluente, ou *Fluent Interface*, em inglês (descrição por Martin Fowler em <http://martinfowler.com/bliki/FluentInterface.html>). Com ela, favorecemos a legibilidade do código, deixando as chamadas mais expressivas.

Veja como seria o código que configura um `Carro` sem a interface fluente, e compare com o código do teste visto anteriormente:

```
#sem interface fluente
builder = CarroValidoBuilder.new
builder.com_cor('azul')
builder.com_quilometragem(100)

#com interface fluente
builder = CarroValidoBuilder.new
  .com_cor('azul')
  .com_quilometragem(100)
```

Com a classe `CarroValidoBuilder` definida, podemos partir para a refatoração dos códigos onde carros são instanciados. O exemplo de teste da classe `Carro` ficaria assim:

```
#spec/builder/carro_valido_builder_spec.rb
it 'deve ter uma placa especificada' do
```

```
carro_invalido = CarroValidoBuilder.new
    .com_placa(nil)
    .construir_carro
erro = 'placa não pode ser nulo'
expect(carro_invalido.validar!).to eq(false)
expect(carro_invalido.erros.size).to eq(1)
expect(carro_invalido.erros.first).to eq(erro)
end
```

Facilitar a instanciação de objetos da classe `Carro` com o padrão Builder também permite que outros tipos de Builder sejam criados e usados, basta manter a mesma interface. Também é possível utilizá-lo no código de produção, aproveitando ainda mais da semântica das interfaces fluentes para deixar o código de criação de carros mais legível.

Ruby possui uma boa funcionalidade conhecida como *Keyword Arguments*, que pode ajudar a melhorar a legibilidade quando muitos parâmetros precisam ser passados a um método. Com eles, é possível nomear os parâmetros e também definir valores padrão.

Veja o exemplo a seguir, mas lembre-se de que essa funcionalidade só está disponível a partir da versão 2.1 da linguagem.

```
def calcular_total(valor:, bonus:, modificador: 1)
  (valor + bonus) * modificador
end

# modificador assume o valor padrão 1
calcular_total(valor: 10, bonus: 10)
```

O padrão Builder permite mais do que apenas melhorar a legibilidade na construção de objetos. Os métodos de configuração dão uma maior flexibilidade, permitindo adicionar lógica específica em algum ponto. Assim, mesmo que a chamada do método fique legível utilizando *Keyword Arguments*, usar um Builder divide as responsabilidades entre as classes.

Ao completar a refatoração, Ana mostra para seus colegas de equipe o quanto mais legível ficou o código com a utilização do padrão Builder. Além disso, nenhum código de produção foi alterado por essa refatoração e os testes ficaram bem mais legíveis, o que facilitou a aceitação pelo resto da equipe.



## CAPÍTULO 10

# Decorator: adicionando características

Módulos são uma excelente maneira de classes compartilharem código em Ruby, basta incluir o módulo e já é possível utilizar seus métodos. Entretanto, nem sempre é a melhor solução, principalmente quando é preciso ter uma grande interação entre o módulo e a classe que o inclui.

Em um dos seus projetos de final de semana, Tarso está desenvolvendo um jogo no qual os personagens terão acesso a um grande número de armas. Além disso, cada arma poderá ser incrementada ao longo do jogo, melhorando as habilidades do personagem que a utiliza. Esses incrementos também poderão se acumular, permitindo que cada jogador customize ao máximo seu personagem e suas habilidades.

## 10.1 ESPADA MÁGICA FLAMEJANTE DA VELOCIDADE

Todo jogador começa com uma arma básica, cada uma com características diferentes. A partir daí, o jogador pode usar encantamentos na sua arma de acordo com seu estilo de jogo. Esse é o fator que torna o jogo atraente: um bom nível de customização para os jogadores permite que eles tenham muitos caminhos diferentes para seguir.

Para exemplificar, o seguinte caso de teste mostra como uma adaga simples modificaria os atributos de um personagem. O primeiro passo é criar um `Personagem` e uma `Adaga`, depois a adaga será equipada ao personagem e o teste verifica que a sua `forca_de_ataque` recebeu o bônus da adaga.

```
#spec/decorator/personagem_spec.rb
context 'utilizando adaga' do
  it 'tem +10 de força de ataque' do
    personagem = Personagem.new(dano_base: 5)
    adaga = Adaga.new
    personagem.equipar_arma(adaga)
    forca_de_ataque = personagem.forca_de_ataque
    expect(forca_de_ataque).to eq(15)
  end
end
```

A implementação do `Personagem` é bem simples, ele possui seus atributos básicos e uma arma. No método `forca_de_ataque`, o bônus da arma que ele está usando é adicionado ao dano básico do personagem. O mesmo é válido para o método `velocidade_de_ataque`:

```
#lib/decorator/personagem.rb
class Personagem
  attr_reader :dano_base, :arma_atual, :velocidade

  def initialize(dano_base: dano_base, velocidade: velocidade)
    @dano_base = dano_base
    @velocidade = velocidade
  end

  def equipar_arma(arma)
```

```
    @arma_atual = arma
  end

  def forca_de_ataque
    @dano_base + @arma_atual.bonus_dano
  end

  def velocidade_de_ataque
    @velocidade + @arma_atual.bonus_velocidade
  end
end
```

A Adaga também possui uma implementação bem simples: ela apenas define os métodos e retorna os números dos modificadores da adaga:

```
#lib/decorator/adaga.rb
class Adaga
  def bonus_dano
    10
  end

  def bonus_velocidade
    3
  end
end
```

A implementação para outros tipos de armas seria bem parecida: basta seguir a mesma interface da classe Adaga, que os objetos Personagem poderão utilizá-las. Tarso se inspirou em outros jogos nos quais as armas dos personagens evoluem junto com eles, e quer muito adicionar várias possibilidades de melhorias nas armas. A Adaga, por exemplo, poderia ser encantada, virando uma AdagaMagica e oferecendo mais bônus de velocidade e dado:

```
#lib/decorator/adaga_magica.rb
class AdagaMagica
  def bonus_dano
    15
  end
```

```
def bonus_velocidade
  7
end
end
```

O real problema vem quando as melhorias nas armas entram em cena. Conforme novas melhorias para elas vão aparecendo, fica impraticável criar novas classes para cada uma das possibilidades. Se quisermos adicionar a propriedade “flamejante” para as armas, precisaríamos de uma `AdagaFlamejante` e de uma `AdagaMagicaFlamejante`.

O mesmo problema seria válido para novas armas. Uma Espada, por exemplo, também precisaria ser `EspadaFlamejante`, `EspadaMagicaFlamejante` ou `EspadaMagica`.

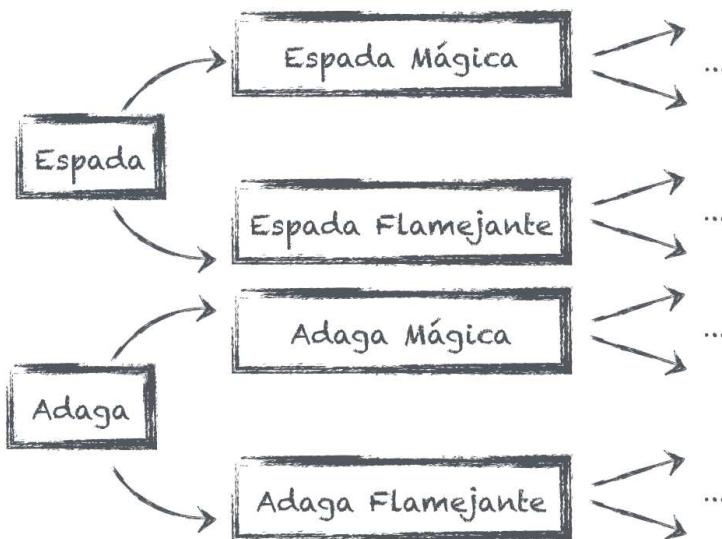


Fig. 10.1: Possibilidades de evolução das armas

Para adicionar uma melhoria que aumenta a velocidade de ataque, seriam criadas mais 4 novas classes: `AdagaDaVelocidade`, `AdagaMagicaDaVelocidade`, `AdagaFlamejanteDaVelocidade` e

AdagaMagicaFlamejanteDaVelocidade.

Criar um grande número de classes não é o único problema, mantê-las também vai ser bem complicado. Imagine que o bônus de dano para armas mágicas precise ser reajustado, todas as classes que implementam armas mágicas precisarão ser alteradas. Mesmo para um projeto de fim de semana, essa não é a melhor maneira de evoluir o código.

Tarso pensa em utilizar módulos para resolver o problema, definindo uma arma mágica como um módulo e apenas incluindo-o nas classes de armas básicas. Mas, devido à natureza do problema, é preciso um nível de interação muito grande entre as armas básicas e os encantos. É preciso que o módulo saiba sobre informações da arma básica e de outros módulos que possam estar incluídos também.

O ideal aqui seria alguma maneira de separar as armas básicas das suas melhorias, para que as bonificações sejam calculadas e aplicadas no valor base das armas. Depois de algumas conversas com seus colegas, Tarso estuda um pouco sobre o padrão *Decorator* e descobre que é exatamente a solução que ele procura.

## 10.2 O PADRÃO DECORATOR

O problema que o padrão Decorator resolve é estruturar o incremento de funcionalidades de um objeto dinamicamente, por isso é classificado como um padrão estrutural. O contexto é bem importante, pois é preciso um conjunto bem definido de objetos componentes, que possuem o comportamento básico, e objetos decoradores, que recebem um componente e incrementam seu comportamento.

Para resolver o problema, o padrão sugere que decoradores e componentes usem uma mesma interface, assim o cliente pode utilizar qualquer um. Além disso, os decoradores recebem um componente básico para que, ao executar um método, sejam chamados tanto o método do componente básico quanto a modificação do decorador.

A flexibilidade do uso de decoradores é maior ainda, pois, como eles possuem a mesma interface, é possível empilhar decoradores em um único componente. Dessa forma, o decorador mais externo vai repassar a chamada para

seu decorador interno até finalmente chegar ao componente.

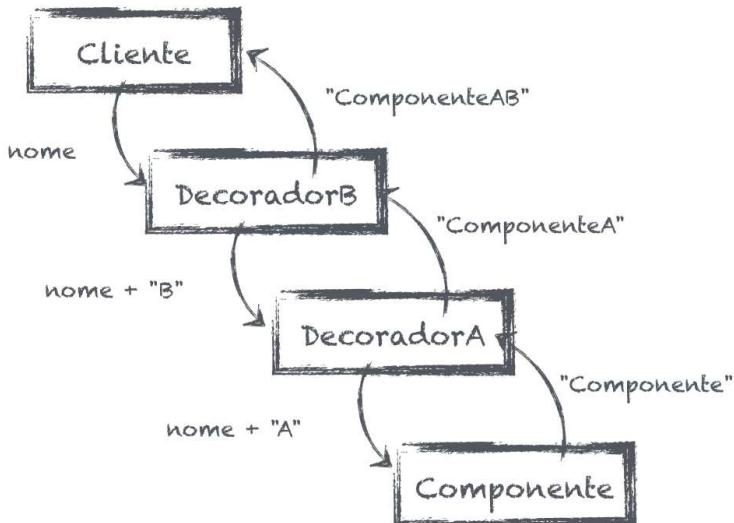


Fig. 10.2: Encadeamento de chamadas

Semelhante a uma chamada recursiva, o componente retorna a chamada para um decorador, que retorna para o próximo decorador da pilha, até atingir o decorador final, onde o cliente obterá o resultado agregado de todas as chamadas.

No problema de Tarso, nós temos as armas como componentes básicos (adagas, espadas etc.) e os encantamentos como decoradores (armas mágicas, flamejantes etc.). A nova implementação do decorador `ArmaMagica` é tão simples quanto o código já existente. A única adição é o componente básico `arma`, que será utilizado para calcular os valores:

```
#lib/decorator/arma_magica.rb
class ArmaMagica

  attr_reader :arma
```

```
def initialize(arma)
    @arma = arma
end

def bonus_dano
    @arma.bonus_dano + 5
end

def bonus_velocidade
    @arma.bonus_velocidade + 4
end

end
```

Será necessário modificar um pouco os testes da classe `personagem` para criar uma adaga mágica. Ao instanciar uma `ArmaMagica`, passamos uma `Adaga`, criando a estrutura de decorador e componente básico. Agora, ao calcular a força de ataque do personagem, a `ArmaMagica` vai repassar a chamada à `Adaga`, que retorna o valor base do bônus de dano (+10). Em seguida, é aplicado o bônus da `ArmaMagica` (+5), que será somado ao dano base do personagem (5):

```
#lib/decorator/personagem.rb
context 'com encatamento mágico' do
  it 'tem +15 de força de ataque' do
    personagem = Personagem.new(dano_base: 5)
    adaga_magica = ArmaMagica.new(Adaga.new)
    personagem.equipar_arma(adaga_magica)
    forca_de_ataque = personagem.forca_de_ataque
    expect(forca_de_ataque).to eq(20)
  end
end
```

Para definir um novo decorador – por exemplo, `Flamejante` –, basta seguir a mesma interface da `ArmaMagica` e definir seus próprios valores de bonificação. Instanciar uma adaga mágica flamejante da velocidade requer apenas englobar uma `Adaga` nos respectivos decoradores:

```
# Adaga Mágica Flamejante
adaga = Flamejante.new(ArmaMagica.new(Adaga.new))
```

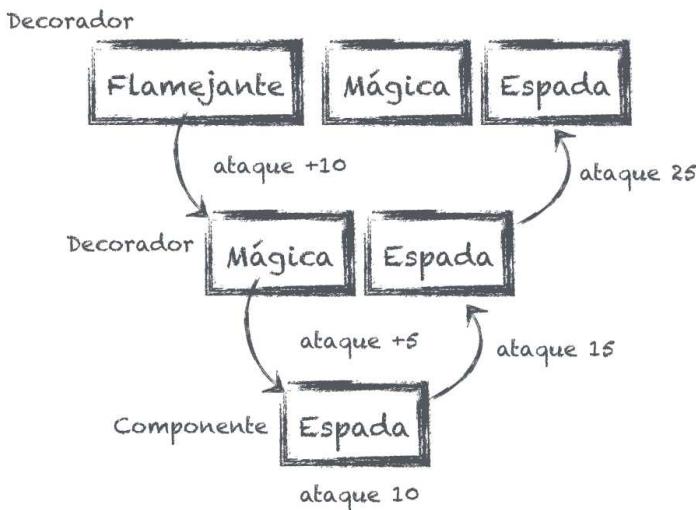


Fig. 10.3: Encadeamento de chamadas

No final, Tarso precisa de apenas 3 classes para criar todos os diferentes tipos de adagas. Além disso, se ele quiser modificar as regras de como os encantos funcionam, a mudança ocorre apenas no respectivo decorador.

Adicionar novas armas é igualmente simples, basta seguir a mesma interface e será possível agregar todos os decoradores já existentes. Criar uma Espada seria apenas definir os seus valores:

```
#lib/decorator/adaga.rb
class Espada
  def bonus_dano
    15
  end

  def bonus_velocidade
    1
  end
end
```

Agora podemos passar um objeto `Espada` da mesma forma que um objeto `Adaga`, reutilizando os mesmos decoradores.

```
# Espada Mágica Flamejante
espada = Flamejante.new(ArmaMagica.new(Espada.new))
```

## RUBY E HERANÇA

As classes `Adaga` e `Espada` servem ao mesmo propósito e possuem a mesma interface. Devido ao *duck typing*, não somos forçados a criar uma estrutura de herança, para que a linguagem entenda que queremos utilizar objetos dessas duas classes, em qualquer situação. O mesmo também é válido para os decoradores `ArmaMagica` e `Flamejante`.

Apesar de não sermos obrigados a criar uma classe mãe chamada, por exemplo, de `Arma`, pode ser uma boa ideia para melhorar a legibilidade do código. Ao olhar para as definições das classes, sabemos que, por herdarem da mesma classe, `Adaga` e `Espada` possuem uma interface em comum.

A herança faz sentido aqui pois a classe `Adaga` realmente é uma especialização de `Arma` e utilizaria todos os seus recursos, além da melhora na legibilidade. O uso de herança com o objetivo de apenas reutilizar código deve ser evitado, pois aumenta o acoplamento entre as classes, dificultando a manutenção do código.

No final da refatoração, Tarso fica contente com o ganho na flexibilidade, mesmo sendo apenas seu projeto de final de semana. Além de ter colocado o padrão em prática, ele consegue evoluir bem melhor os sistemas de arma do jogo. Agora, resta fazer mais “pesquisas” para conseguir novas ideias para o seu jogo.



## CAPÍTULO 11

# Mediator: notificações inteligentes

Gil é a atual líder técnica de uma equipe construindo uma aplicação que monitora produtos em lojas virtuais. Uma das grandes funcionalidades do sistema é emitir alertas quando determinados eventos acontecem.

Por exemplo, uma apaixonada por jogos possui uma lista de produtos que gostaria de comprar, então, sempre que for detectada uma mudança no preço desses produtos, um e-mail será enviado para esse usuário. Ou se houver uma grande procura por determinado produto, um alerta será emitido para outro sistema, caso os estoques estejam baixos.

Enquanto pareava com um dos seus colegas de equipe, Gil notou um problema que começava a aumentar. A grande variedade de fontes de alertas e os possíveis destinos que eles teriam tornariam o código de notificações cada

vez mais complexo, além da mistura com regras de negócio importantes.

## 11.1 O ESPAGUETE DE NOTIFICAÇÕES

O problema que Gil notou pode ser visto mais claramente no código do `BuscaPromocaoWorker`, que busca no banco de dados por uma lista de produtos e verifica se houve alguma mudança nos seus preços. Caso alguma mudança seja detectada, serão enviadas notificações para os usuários que tem esse produto na sua lista de favoritos (por meio de e-mail) e também para o fornecedor do produto (uma chamada REST), avisando sobre uma possível alta na demanda pelo produto.

A implementação do `worker` busca no banco de dados a lista de produtos com preços baixos (`produtos_promocionais`), e faz uma interseção com a lista de produtos de interesse de um determinado usuário. Assim, ele obtém a lista de produtos para emitir alertas, utilizando o `NotificadorCliente` e o `NotificadorFornecedor`.

```
class BuscaPromocaoWorker
  def executar(usuario)
    produtos =
      produtos_promocionais & usuario.produtos_de_interesse
    NotificadorCliente.produtos_em_promocao(produtos)
    NotificadorFornecedor.produtos_em_promocao(produtos)
    atualizar_notificacao_do_usuario
  end
end
```

Para separar as responsabilidades, foram criadas classes bem específicas de notificação. Dessa forma, os testes unitários não precisam se preocupar com o que cada notificador vai fazer.

```
it 'deve notificar sobre produto em promocao' do
  produtos = ['Super Maria Brothers', 'USB Controller']
  usuario = Usuario.new(email: 'usuario@email.com',
                        produtos_de_interesse: produtos)

  worker = BuscaPromocaoWorker.new
```

```
allow(worker).to receive(:produtos_promocionais)
    .and_return(produtos)

expect(NotificadorCliente).to
  receive(:produtos_em_promocao).with(produtos)
expect(NotificadorFornecedor).to
  receive(:produtos_em_promocao).with(produtos)

worker.executar(usuario)
end
```

Esse *worker* é apenas um dos locais onde os notificadores são usados, as fontes de onde as notificações podem ser transmitidas são bem variadas. Sem falar que as notificações possuem vários fins, desde e-mail para usuários até chamadas REST para outras aplicações.

Contudo, o problema que Gil vê com esse código é que a lógica de negócio precisa saber quais notificadores acionar. Com a evolução dessa aplicação, os objetos se relacionarão de uma maneira “muitos-para-muitos”, com várias fontes de notificação acionando vários notificadores.

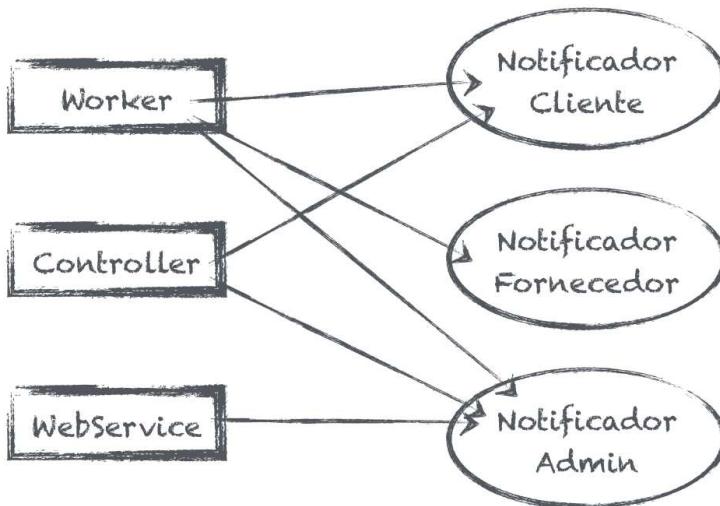


Fig. 11.1: Fontes de notificação e notificadores

Ao modelar bancos de dados relacionais, quando existe uma relação “muitos-para-muitos” entre duas tabelas, um padrão que Gil conhece é criar uma tabela de associação. Essa tabela mantém referências para as outras duas e, cada uma referencia apenas a tabela de associação em vez de todas as entradas da outra.

Para modelagem orientada a objetos, podemos utilizar o mesmo padrão e criar uma classe para intermediar um relacionamento complicado. Esse padrão é conhecido como Mediator.

## 11.2 O PADRÃO MEDIATOR

O problema que o padrão busca resolver é simplificar a forma como um conjunto de objetos interage, por isso é classificado como um padrão de comportamento. Assim, evitamos um acoplamento forte com objetos referenciados diretamente uns aos outros.

Para utilizar bem o padrão, é necessário que o relacionamento entre os

grupos de objetos seja bem claro e possa ser extraído. A solução proposta é criar um novo objeto específico para intermediar a comunicação, o mediador. Assim, em vez de referenciar vários objetos ao mesmo tempo, basta usar o mediador.

Uma metáfora para explicar bem o Mediator é pensar em uma central de atendimento telefônico. Ao ligar para uma empresa, você não precisa saber exatamente qual o telefone para o setor de vendas ou atendimento ao consumidor. Você só precisa de um único número e, ao ligar para ele, será redirecionado para o setor de seu interesse.

Para resolver o problema de Gil, precisamos encapsular a maneira como as notificações são enviadas. Assim, as fontes das notificações não precisam se preocupar com quais notificadores chamar, apenas que uma notificação precisa ser enviada.

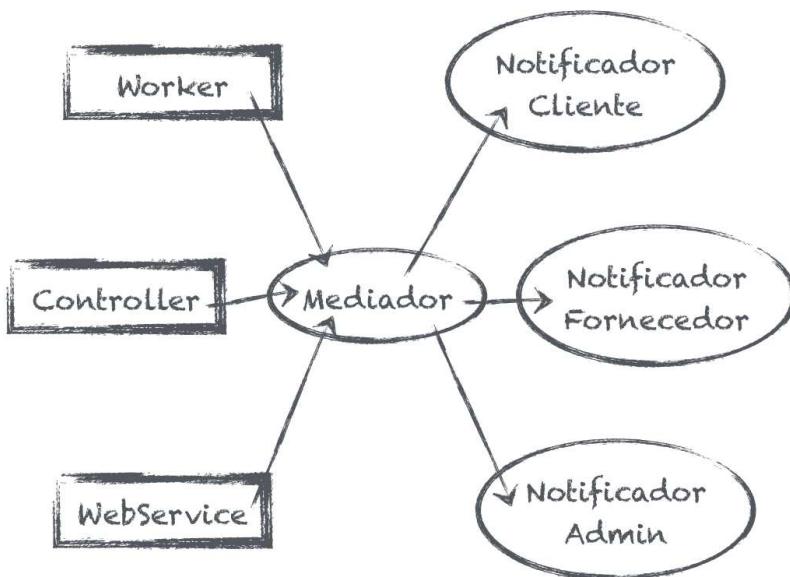


Fig. 11.2: Mediador simplificando o relacionamento

O primeiro passo é aplicar **Extrair Método** na classe BuscaPromocaoWorker para separar a lógica de notifica-

ção da lógica de negócio. Para isso, Gil cria o novo método `notificar_produtos_em_promocao`, e move a chamada do `NotificadorCliente` e `NotificadorFornecedor` para lá.

```
class BuscaPromocaoWorker
  def executar(usuario)
    produtos =
      produtos_promocionais & usuario.produtos_de_interesse
    notificar_produtos_em_promocao(produtos)
    atualizar_notificacao_do_usuario
  end

  def notificar_produtos_em_promocao(produtos)
    NotificadorCliente.produtos_em_promocao(produtos)
    NotificadorFornecedor.produtos_em_promocao(produtos)
  end
end
```

Após essa refatoração, fica bem mais claro o que é necessário na classe `NotificadorMediator`. Todos os eventos de notificação devem estar implementados na classe mediadora, e dentro desses métodos é que vamos chamar os notificadores específicos.

Então, com o comportamento esperado da classe mediadora, podemos escrever testes verificando que o `NotificadorCliente` e o `NotificadorFornecedor` sejam chamados ao chamar `NotificadorMediator.produtos_em_promocao`:

```
it 'deve notificar Cliente e Fornecedor sobre promoções' do
  produtos = ['Super Maria Brothers', 'USB Controller']
  expect(NotificadorCliente).to
    receive(:produtos_em_promocao).with(produtos)
  expect(NotificadorFornecedor).to
    receive(:produtos_em_promocao).with(produtos)
  NotificadorMediator.produtos_em_promocao(produtos)
end
```

A implementação do mediador é bem simples, basta apenas delegar a chamada dos métodos aos notificadores a quem ela interessar. Para isso, vamos

utilizar **Extrair Classe** em `BuscaPromocaoWorker` e mover a responsabilidade de notificação para o novo `NotificadorMediator`:

```
class NotificadorMediator
  def self.produtos_em_promocao(produtos)
    NotificadorCliente.produtos_em_promocao(produtos)
    NotificadorFornecedor.produtos_em_promocao(produtos)
  end
end
```

Agora, na classe `BuscaPromocaoWorker`, Gil precisa apenas usar o mediador. A melhora na simplicidade do método já fica visível nos testes da classe, que agora só precisam verificar que o método `produtos_em_promocao` no `NotificadorMediator` foi chamado, em vez de verificar cada um dos possíveis notificadores.

```
it 'deve notificar sobre produto em promocao' do
  produtos = ['Super Maria Brothers', 'USB Controller']
  usuario = Usuario.new(email: 'usuario@email.com',
                        produtos_de_interesse: produtos)

  worker = BuscaPromocaoWorker.new
  allow(worker).to receive(:produtos_promocionais)
    .and_return(produtos)

  expect(NotificadorMediator).to receive(:produtos_em_promocao)
    .with(produtos)

  worker.executar(usuario)
end
```

A implementação final do `BuscaPromocaoWorker` fica simplificada e apenas delega o envio de notificações para `NotificadorMediator`.

```
class BuscaPromocaoWorker
  def executar(usuario)
    produtos_de_interesse =
      produtos_promocionais & usuario.produtos_de_interesse
    NotificadorMediator.
```

```
    produtos_em_promocao(produtos_de_interesse)
    atualizar_notificacao_do_usuario
end
end
```

Gil olha para o código e também percebe o quanto fácil será adicionar novos notificadores a essa estrutura. Caso a notificação já exista, basta adicionar uma nova classe que implemente o mesmo método, e adicioná-la à lista de classes do mediador. Por exemplo, para alertar o time de uma possível alta no tráfego do site quando um produto entra em promoção:

```
class NotificadorSite
  def self.produtos_em_promocao(produtos)
  end
end

class NotificadorMediator
  def self.produtos_em_promocao(produtos)
    NotificadorSite.produtos_em_promocao(produtos)
    NotificadorCliente.produtos_em_promocao(produtos)
    NotificadorFornecedor.produtos_em_promocao(produtos)
  end
end
```

Se for preciso adicionar uma notificação completamente nova, basta criar o novo método no mediador e usá-lo nas classes que devem disparar a notificação. Um exemplo seria adicionar um alerta para fornecedores sobre produtos com pouco estoque. O primeiro passo é definir o novo alerta `produtos_em_baixa` em `NotificadorFornecedor`.

```
class NotificadorFornecedor
  def self.produtos_em_baixa(produtos)
  end
end
```

Em seguida, basta criar o novo método `produtos_em_baixa` em `NotificadorMediator`, que vai apenas delegar a chamada para os notificadores.

```
class NotificadorMediator
  def self.produtos_em_baixa(produtos)
    NotificadorFornecedor.produtos_em_baixa(produtos)
  end
end
```

Por fim, na fonte de alertas, basta chamar o `NotificadorMediator`, que ele cuida do resto.

```
class EstoqueBaixoWorker
  def executar(usuario)
    produtos = produtos_com_baixo_estoque()
    NotificadorMediator.produtos_em_baixa(produtos)
  end
end
```

Gil apresenta o código para a sua equipe, e todos gostam da facilidade de extensão e de como as responsabilidades ficaram bem separadas. Agora, com essa dívida técnica a menos, a equipe consegue implementar notificações de maneira bem mais rápida.



## **Parte IV**

### **Conclusão**



## CAPÍTULO 12

# Os outros padrões

Ao longo do livro, exploramos apenas 9 padrões da lista de 23 catalogados pela Gangue dos Quatro (Erich Gamma, Ralph Johnson, Richard Helm e John Vlissides). Isso não quer dizer que os outros devam ser ignorados, apenas que, como o foco deste livro é a utilização de padrões no processo de refatoração, decidi reduzir o conjunto de padrões a serem explorados e focar na sua aplicação.

Alguns dos padrões não entraram no livro, pois sua utilização é muito específica, como o *Chain of Responsibility*, e até mesmo pensar em um exemplo real não é tão simples. Outros padrões, como o *Facade*, não são tão comuns, mas são geralmente mal interpretados e utilizados. E, finalmente, padrões como *Singleton* são usualmente contestados devido aos problemas que eles resolvem.

## 12.1 PADRÕES POUCO UTILIZADOS

Vimos que alguns padrões podem ser simplificados, mas existe um outro conjunto de padrões que foram deixados de lado e não foram apresentados no livro. Isso não quer dizer que sejam inúteis, apenas que sua aplicação é muito rara.

Um bom exemplo é o padrão *Chain of Responsibility*, que propõe desacoplar a chamada do objeto do código do cliente para que mais de um objeto consiga tratar a solicitação. A solução proposta é que os objetos receptores sejam encadeados para que as chamadas sejam passadas ao longo da cadeia.

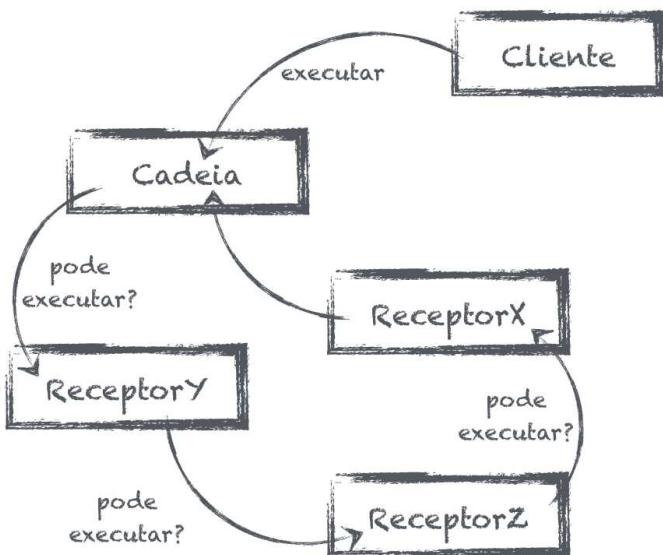


Fig. 12.1: Utilização do padrão Chain of Responsibility

Ter múltiplos objetos que devem responder a uma mesma solicitação talvez seja um problema real que você tem, mas as chances são pequenas. Mesmo que esse seja o caso, será que o problema não pode ser resolvido de uma maneira mais simples? Lembre-se de que, no começo do livro, conversamos sobre procurar soluções mais simples antes de aplicar qualquer padrão, devido à complexidade e os níveis de abstrações que os padrões trazem.

Até mesmo os padrões detalhados no livro devem ser cuidadosamente aplicados. Uma simplificação no design é sempre melhor do que qualquer aplicação de padrão de projeto.

## 12.2 PADRÕES MAL UTILIZADOS

Existem alguns padrões que são considerados por muitos como antipadrões, ou seja, soluções que foram usadas e que não melhoraram o design da aplicação. O padrão *Facade* é um bom exemplo para demonstrar isso.

O problema que ele busca resolver é quando existe um conjunto de componentes distintos que precisam funcionar juntos. A solução é definir uma interface, em um nível mais alto, que facilite a utilização de componentes de um nível menor.

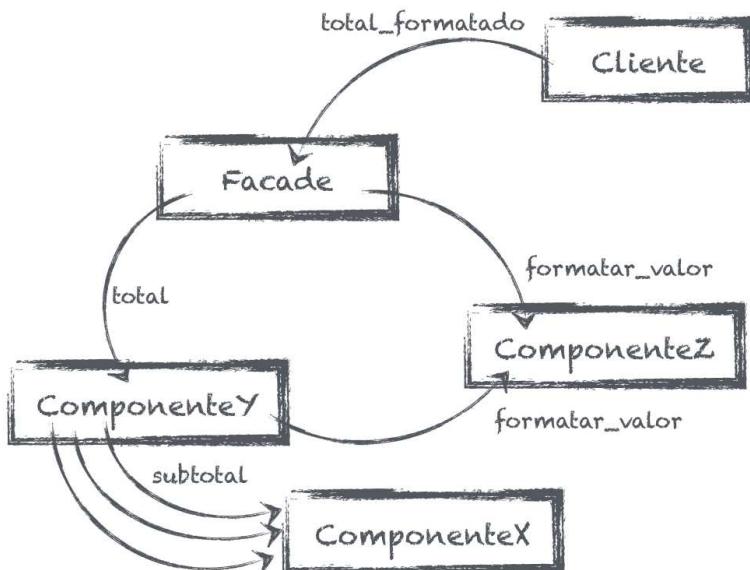


Fig. 12.2: Utilização do padrão Facade

Esse parece um problema bem comum; então, por que não utilizar o padrão *Facade*?

O padrão cria uma maneira mais fácil de usar um código que seria uma bagunça completa. Mas facilitar o acesso não resolve a bagunça completa!

É bem fácil olhar para o código do cliente e, vendo apenas uma chamada sendo feita, achar o código bonito, simples e claro. Mas, quando for preciso alterar esse código, a bagunça continuará lá.

Jogar a sujeira para debaixo do Facade não vai trazer nenhuma melhoria para sua aplicação. O problema não é com o padrão em si, mas com a aplicação fora de contexto.

Dado um conjunto de componentes que devem funcionar juntos e estão bem organizados, então o Facade facilita a utilização provendo pontos mais simples. Outra vantagem dele é que alterações nos componentes internos não se propagam até o código do cliente, facilitando a manutenção.

## 12.3 PADRÓES QUE NINGUÉM DEVERIA UTILIZAR

Ainda existe um outro conjunto de padrões considerado antipadrões, pois o problema que eles resolvem não deveria ser resolvido! Vejamos o padrão Singleton.

Sua ideia é garantir que uma classe tenha apenas um instância em toda a aplicação, fornecendo um único ponto de acesso global a ela.

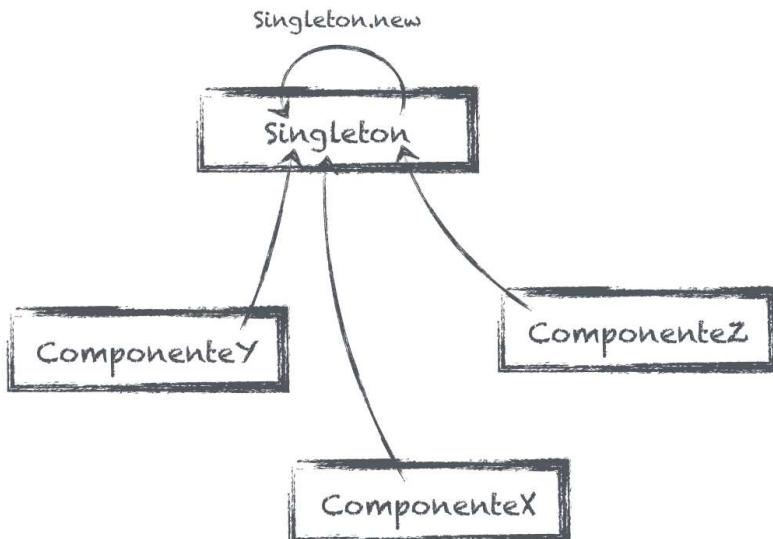


Fig. 12.3: Utilização do padrão Singleton

O problema é que fornecer um ponto de acesso global para uma única instância é basicamente criar uma instância global. Criar variáveis globais é um grande problema pois, dado um ponto específico da aplicação, é muito difícil ter certeza do que vai estar lá. Então, o código que usa uma instância global precisa sempre ficar se perguntando o estado dessa instância antes de tomar ações.

Isso não quer dizer que utilizar o Singleton seja ruim. Caso você realmente precise de uma instância global, o padrão propõe uma boa solução. Mas ter e acessar uma instância global não deveria ser um problema no seu código.



## CAPÍTULO 13

# Padrões de projeto e linguagens dinâmicas

Alguns acreditam que a lista original de padrões é muito grande e que muitos deles podem ser simplificados quando aplicados em linguagens com características dinâmicas e/ou funcionais. Uma das principais fontes é a apresentação de Peter Norvig (<http://www.norvig.com/design-patterns/>) que mostra que, devido às características das linguagens Lisp e Dylan, 16 dos 23 padrões são mais simples ou mesmo invisíveis.

Mesmo concordando com essa opinião, vale ressaltar que a ideia original de um padrão de projeto é propor uma solução para um problema em um dado contexto. Isso não tem nada a ver com a quantidade de classes que serão utilizadas ou com como é o diagrama UML (*Unified Modelling Language*) da sua solução!

## 13.1 PADRÕES MAIS SIMPLES

Ruby possui características dinâmicas e funcionais, além da base com Orientação a Objetos. Isso quer dizer que nem sempre será necessário criar uma nova classe para implementar um padrão. Para exemplificar a discussão, vamos analisar o padrão *Command* e como ele poderia ser implementado em Ruby.

O problema que o padrão resolve é quando você precisa de mais flexibilidade ao chamar métodos, seja porque o cliente só saberá o que deve ser chamado em tempo de execução, ou porque é preciso guardar informações sobre os métodos ou até mesmo enfileirar essas chamadas.

A solução proposta é criar um objeto que é responsável apenas por executar o comando (chamar um determinado método em um objeto); assim, em vez de chamar o método no objeto diretamente, utilizamos o objeto *Command* que fará isso. O contexto do problema deve permitir essa separação entre chamada e objeto a ser chamado.

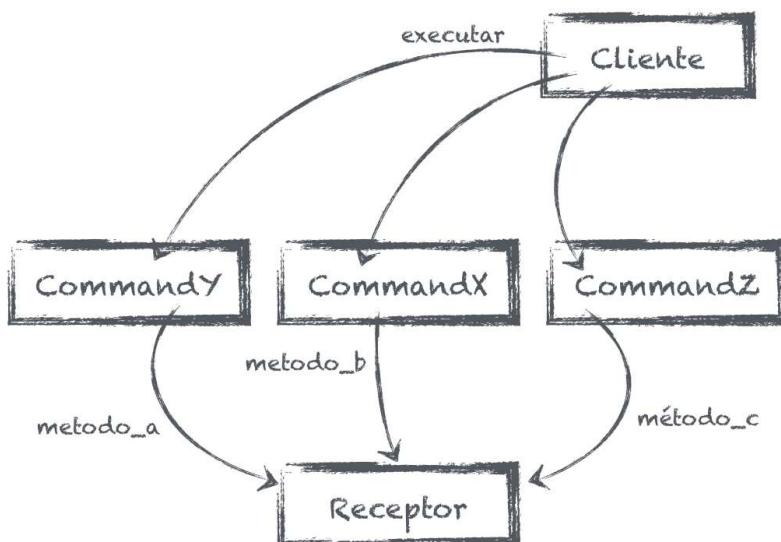


Fig. 13.1: Utilização do padrão Command

De acordo com a figura anterior, como temos objetos `Command` que vão executar os métodos no `Receptor`, podemos agora criar um *array* com uma lista de comandos a serem executados, passar um comando como parâmetro etc.

No entanto, em Ruby, podemos usar o método `send` para executar um método qualquer em outro objeto, basta passar o nome do método como um símbolo. Assim, a separação entre chamar um método e o objeto é feita de maneira bem mais simples do que a criação de uma estrutura de classes para representar objetos `Command`.

```
#commands
COMMAND_Y = :metodo_a
COMMAND_X = :metodo_b
COMMAND_Z = :metodo_c

Receptor.send(COMMAND_X) #comando como parâmetro
lista_de_comandos = [COMMAND_Z, COMMAND_Y, COMMAND_X, COMMAND_Y]
executar_comandos(lista_de_comandos) #comandos como lista
```

Apesar de não criar um objeto específico para executar os métodos no receptor, a ideia de desacoplar a chamada do objeto para ganhar mais flexibilidade continua presente!

Estendendo o problema para que o objeto `Command` faça mais do que apenas delegar chamadas para o receptor (por exemplo, escrever em um arquivo de *log*), nossa solução precisa ser um pouco mais robusta do que apenas o nome do método.

Em Ruby, funções são objetos de primeira classe, como visto na seção 3.2. Isso quer dizer que você pode guardar funções em variáveis, passar como argumento e qualquer outra coisa que um objeto comum possa fazer.

```
soma = lambda do |a, b| #função em variável
  a + b
end # retorna um Proc

soma.call(1, 2) #executar variável

executar(soma) #passar como parâmetro
```

Utilizando essa funcionalidade, basta que nossos comandos guardem um Proc que será executado depois:

```
COMMAND_Y = lambda do
    log.info("Executando metodo_a no Receptor")
    Receptor.metodo_a
end

COMMAND_Y.call
```

## 13.2 PADRÕES INVISÍVEIS

Da mesma forma que temos padrões simplificados, também temos padrões que fazem parte do coração da linguagem. Nós os usamos o tempo todo sem nem mesmo saber. Um bom exemplo é o padrão *Iterator*.

O problema que o Iterator resolve é quando você precisa de um meio para acessar, sequencialmente, os elementos de um objeto agregado sem expor sua estrutura interna. A solução do padrão é criar um novo objeto que vai percorrer o objeto agregado, que internamente vai ter um array ou lista, e retornar cada um dos elementos desse conjunto.

O objeto `Iterador` contém lógica para obter elementos da lista, com um método `proximo`, que retorna o próximo elemento da sequência, e outro `acabou?`, que é utilizado para encerrar o laço. Assim, conseguimos isolar a implementação da lista de livros promocionais e fornecer acesso apenas aos nomes dos livros.

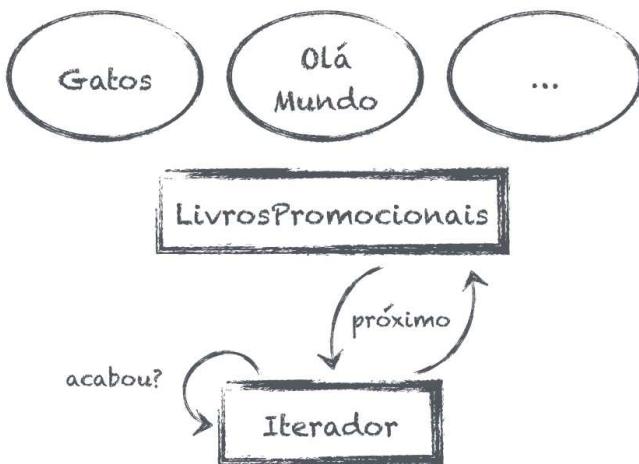


Fig. 13.2: Funcionamento do padrão Iterador

O código a seguir mostra como seria a utilização de um `Iterador` em um laço. Enquanto o iterador tiver elementos para serem percorridos, basta pegar o próximo da lista:

```
iterador = Iterador.new

while (!iterador.acabou?) do
  atual = iterador.proximo
  # código para tratar o elemento da lista
end
```

Se você já está acostumado com Ruby, deve ter pensado que implementar uma estrutura assim não faz o menor sentido. Podemos simplesmente usar o método `each` e vamos obter exatamente o mesmo comportamento.

O código a seguir implementa uma lista de livros em promoção, utilizando a classe `LivrosPromocionais`. Dentro dessa classe, é definido um array `livros_em_promocao` que vai armazenar uma lista de strings com os

nomes dos livros. Como o objetivo do padrão é esconder a estrutura de agregação interna, podemos implementar o nosso próprio método `each`, que vai apenas delegar a chamada para o array `livros_em_promocao`.

```
class LivrosPromocionais
  def initialize(livros)
    # flatten garante que sempre teremos um array
    @livros_em_promocao = [livros].flatten
  end

  def each(&bloco)
    @livros_em_promocao.each(&bloco)
  end
end

livros_promocionais =
  LivrosPromocionais.new(["Alo mundo", "Gatos"])
livros_promocionais.each do |livro|
  puts "livro em promoção: #{livro}"
end
```

A principal motivação de esconder a estrutura interna do objeto agregador é facilitar a troca da sua implementação, por exemplo, trocar um *array* por um *hash*. No entanto, devido ao *duck typing*, não precisamos nos preocupar com qual o tipo do objeto, contanto que ele responda aos mesmos métodos. Outra solução é apenas expor a lista `livros_em_promocao` e chamar o método `each` diretamente nela.

```
class LivrosPromocionais
  attr_reader :livros_em_promocao

  def initialize(livros)
    # flatten garante que sempre teremos um array
    @livros_em_promocao = [livros].flatten
  end
end

livros_promocionais =
  LivrosPromocionais.new(["Alo mundo", "Gatos"])
```

```
livros_promocionais.livros_em_promocao.each do |livro|
  puts "livro em promoção: #{livro}"
end
```

Assim, mesmo que `livros_em_promocao` precise ser qualquer outro tipo de estrutura de dados, basta que ele responda ao método `each`, que a interface continuará a mesma.



## CAPÍTULO 14

# Conclusão

Parabéns por ter chegado até o final do livro! Espero que você tenha gostado de acompanhar a jornada dos nossos heróis enquanto utilizava padrões de projeto para refatorar seus códigos. Mas, principalmente, espero que você tenha conseguido usar as técnicas apresentadas no seu código atual.

Agora que você já conhece vários padrões e sabe como aplicá-los, é preciso ter cuidado para não sair aplicando-os apenas por aplicar. Como já falei no começo do livro, é preciso considerar o contexto antes de aplicar um padrão para garantir que o código vai realmente melhorar.

### **14.1 DESIGN EVOLUCIONÁRIO**

A discussão sobre planejar o design no começo do projeto ou deixar que ele evolua a partir das necessidades de crescimento da aplicação não é nova. No artigo *Is Design Dead* (<http://martinfowler.com/articles/designDead.html>) ,

Martin Fowler mostra como as práticas do *Extreme Programming* (XP) permitem que a evolução do design seja viável. O ponto principal do artigo é a importância de deixar o código o mais simples possível.

Segundo Kent Beck (*Extreme Programming Explained: embrace change*, 1999), simplicidade de código pode ser definida a partir das seguintes características:

- Todos os testes passam;
- Sem duplicação;
- Mostra as intenções;
- Possui o menor número de classes ou métodos.

Ao aplicar padrões de projeto, não necessariamente estamos simplificando o código. Com certeza diminuímos duplicações ao refatorá-lo e separar as responsabilidades. Ao separar responsabilidades, também deixamos as intenções de cada parte dele mais claras e fáceis de serem utilizadas. No entanto, aumentamos a quantidade de classes e métodos.

Em teoria, ao tentar deixar o código simples seguindo as ideias apresentadas por Kent Beck, o design da aplicação vai evoluir naturalmente para o uso de padrões. No entanto, Martin Fowler sugere que a utilização de padrões seja mais consciente, aprendendo sobre os padrões de projetos e tendo uma ideia de para onde o design está indo.

Ainda no mesmo artigo, Martin Fowler sugere os seguintes pontos para aproveitar ao máximo padrões de projeto, que resumem perfeitamente o que eu acho que devemos ter em mente ao aprender sobre eles:

- 1) Invista tempo aprendendo sobre padrões;
- 2) Concentre-se em aprender quando aplicá-los (não muito cedo);
- 3) Concentre-se em como implementar padrões na sua forma mais simples, e adicionar complexidade depois;
- 4) Se você adicionar um padrão, e depois perceber que ele não está ajudando, não tenha medo de removê-lo.