

# Design Patterns com Java

Projeto orientado a objetos guiado por padrões



Casa do  
Código

EDUARDO GUERRA

© 2013, 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)



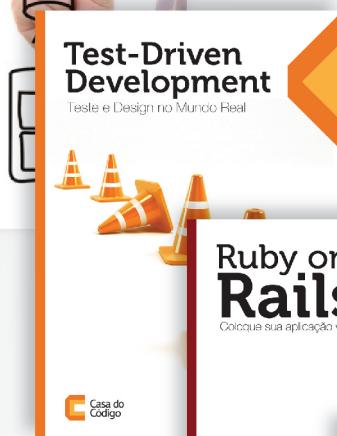
Caelum:  
Cursos de TI presenciais e online

[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)

E-book gerado especialmente para Henrique Simões de Andrade - [cassitos@gmail.com](mailto:cassitos@gmail.com)



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



# Sumário

<b>1 Entendendo Padrões de Projeto</b>	<b>1</b>
1.1 Conceitos da Orientação a Objetos . . . . .	2
1.2 Mas esses conceitos não são suficientes? . . . . .	7
1.3 O Primeiro Problema: Cálculo do Valor do Estacionamento . . . . .	8
1.4 Strategy: o Primeiro Padrão! . . . . .	14
1.5 O que são padrões? . . . . .	16
1.6 Como o Livro Está Organizado . . . . .	19
<b>2 Reuso Através de Herança</b>	<b>21</b>
2.1 Exemplo de Padrão que Utiliza Herança - Null Object . . . . .	22
2.2 Hook Methods . . . . .	26
2.3 Revisando modificadores de métodos . . . . .	28
2.4 Passos diferentes na Mesma Ordem - Template Method . . . . .	29
2.5 Refatorando na Direção da Herança . . . . .	35
2.6 Criando Objetos na Subclasse - Factory Method . . . . .	40
2.7 Considerações Finais do Capítulo . . . . .	44
<b>3 Delegando Comportamento com Composição</b>	<b>47</b>
3.1 Tentando Combinar Opções do Gerador de Arquivos . . . . .	48
3.2 Bridge - Uma Ponte entre Duas Variabilidades . . . . .	51
3.3 Hook Classes . . . . .	56
3.4 State - Variando o Comportamento com o Estado da Classe . . . . .	59
3.5 Substituindo Condicionais por Polimorfismo . . . . .	66
3.6 Compondo com Múltiplos Objetos - Observer . . . . .	69
3.7 Considerações Finais do Capítulo . . . . .	77

<b>4 Composição Recursiva</b>	<b>79</b>
4.1 Compondo um Objeto com sua Abstração . . . . .	80
4.2 Composite - Quando um Conjunto é um Indivíduo . . . . .	83
4.3 Encadeando Execuções com Chain of Responsibility . . . . .	89
4.4 Refatorando para Permitir a Execução de Múltiplas Classes . . . . .	95
4.5 Considerações Finais do Capítulo . . . . .	98
<b>5 Envolvendo Objetos</b>	<b>99</b>
5.1 Proxies e Decorators . . . . .	100
5.2 Exemplos de Proxies . . . . .	106
5.3 Extrairindo um Proxy . . . . .	111
5.4 Adaptando Interfaces . . . . .	114
5.5 Considerações Finais do Capítulo . . . . .	120
<b>6 Estratégias de Criação de Objetos</b>	<b>121</b>
6.1 Limitações dos Construtores . . . . .	122
6.2 Criando Objetos com Métodos Estáticos . . . . .	124
6.3 Um Único Objeto da Classe com Singleton . . . . .	127
6.4 Encapsulando Lógica Complexa de Criação com Builder . . . . .	129
6.5 Relacionando Famílias de Objetos com Abstract Factory . . . . .	136
6.6 Considerações Finais do Capítulo . . . . .	139
<b>7 Modularidade</b>	<b>141</b>
7.1 Fábrica Dinâmica de Objetos . . . . .	142
7.2 Injeção de Dependências . . . . .	147
7.3 Service Locator . . . . .	155
7.4 Service Locator versus Dependency Injection . . . . .	161
7.5 Considerações Finais do Capítulo . . . . .	163
<b>8 Indo Além do Básico</b>	<b>165</b>
8.1 Frameworks . . . . .	166
8.2 Utilizando Tipos Genéricos com os Padrões . . . . .	171
8.3 Padrões com Test-driven Development . . . . .	173
8.4 Posso Aplicar esses Padrões na Arquitetura? . . . . .	178
8.5 Comunidade de Padrões . . . . .	180
8.6 E agora? . . . . .	182

## Índice Remissivo

**183**

Versão: 15.9.16



## CAPÍTULO 1

# Entendendo Padrões de Projeto

*“Eu diria algo como ‘Software está gerenciando o mundo’. Nossa trabalho é apenas poliniza-lo ...”*

– Brian Foote

Imagine que uma pessoa tenha aprendido diversas técnicas de pintura. A partir desse conhecimento ela saberá como pegar um pincel, como misturar as cores e como trabalhar com diferentes tipos de tinta. Será que somente com esse conhecimento ela conseguirá pintar um quadro? Note que ela sabe tudo que se precisa para realizar a pintura, porém todo esse conhecimento não é válido se a pessoa não souber como utilizá-lo. Para realizar essa tarefa, é necessário, além de conhecimento, ter habilidade, que é algo que só se aprende com muita prática e treino. Saber as técnicas é apenas o primeiro passo...

Com programação acontece um fenômeno similar. Quando se aprende uma linguagem orientada a objetos e seus recursos, isso é equivalente a se aprender a pegar em um pincel. Saber, por exemplo, como utilizar herança e polimorfismo não é o suficiente para distinguir em quais situações eles devem ser empregados de forma

apropriada. É preciso conhecer os problemas que podem aparecer durante a modelagem de um sistema e saber quais as soluções que podem ser implementadas para equilibrar requisitos, muitas vezes contraditórios, com os quais se está lidando.

Felizmente existe uma forma de conseguir esse conhecimento sem precisar passar pelo caminho tortuoso de errar várias vezes antes de aprender a forma correta. É um prazer poder apresentar nesse livro os *Design Patterns*, padrões de projeto na tradução mais utilizada, uma forma de se documentar uma solução para um problema de modelagem. Mais do que isso, padrões não são novas soluções, mas soluções que foram implementadas com sucesso de forma recorrente em diferentes contextos. A partir deles, é possível aprender com a experiência de outros desenvolvedores e absorver esse conhecimento que eles levaram diversos anos para consolidar.

O objetivo desse capítulo é fazer uma revisão dos principais conceitos da orientação a objetos e apresentar como os padrões de projeto podem ser utilizados para adquirir habilidade em modelagem de software. Ao seu final, será apresentada uma visão geral de como esses padrões serão apresentados no decorrer do livro.

## 1.1 CONCEITOS DA ORIENTAÇÃO A OBJETOS

Em linguagens mais antigas, não havia nenhuma separação dos elementos de código. Ele era criado em único bloco, onde para se executar desvios no fluxo de execução, como para a criação de loops e condicionais, era preciso executar saltos através do temido comando *goto*. Isso tornava o código muito difícil de ser gerenciado e reutilizado, tornando complicado seu desenvolvimento e manutenção. Foi dessa época que surgiu o termo “*código macarrônico*”, que se referia a dificuldade de compreensão do fluxo de execução desses programas.

Em seguida, para resolver esse problema, foi criada a **programação estruturada**. Nesse novo paradigma foram criadas estruturas de controle que permitiam a criação de comando condicionais, como o *if* e o *switch*, e comandos iterativos, como o *for* e o *while*. Uma outra adição importante foram as funções. A partir delas era possível dividir a lógica do programa, permitindo sua modularização. Isso facilita que essas funções possam ser reutilizadas em outras aplicações. A maior dificuldade nesse paradigma era como lidar com os dados e o comportamento associado a eles. Era comum a criação de variáveis globais, que facilmente poderiam acabar causando problemas por estarem com valores inconsistentes. Outro problema ocorria com dados relacionados que podiam ser livremente modificados e facilmente ficarem inconsistentes.

Como uma evolução da programação estruturada surgiu a **programação orientada a objetos**. Além da estruturação da lógica, com esse paradigma era possível a estruturação dos dados e conceitos do software. Sendo assim, é possível colocar toda lógica relacionada a um conjunto de dados junto com ele. Além disso, a partir da possibilidade de abstração de conceitos consegue-se desacoplar componentes, obtendo um maior reúso e uma maior flexibilidade.

Os tópicos a seguir exploram com maior detalhe os principais conceitos da programação orientada a objetos.

## Classes e Objetos

Na programação orientada a objetos são definidos novos tipos através da criação de **classes**, e esses tipos podem ser instanciados criando **objetos**. A ideia é que um objeto represente uma entidade concreta enquanto sua classe representa uma abstração dos seus conceitos. Se, por exemplo, tivermos uma classe Gato, o Garfield e o Frajola seriam objetos dessa classe. Adicionalmente, uma classe CarteiraDeIdentidade, que representa uma abstração, teria como instâncias a minha e a sua carteira de identidade.

Eu vejo muitas pessoas utilizarem a metáfora "*a classe seria a forma e o objeto seria o pão*" para explicar a relação entre classes e objetos. Pessoalmente, não gosto dessa analogia pois ela leva a entender que a classe é o que produz o objeto, o que seria mais próximo do conceito de fábrica (que será visto no Capítulo 6 desse livro). É importante perceber que os objetos representam entidades concretas do seu sistema, enquanto as classes abstraem suas características.

A classe possui estado e comportamento, que são representados respectivamente pelos atributos e métodos definidos. Enquanto uma classe possui uma característica, um objeto possui um valor para aquela característica. Por exemplo, um Gato possui a cor do pelo, enquanto o Garfield possui seu pelo laranjado. A classe possui um comportamento e o objeto pode realizar aquele comportamento. Seguindo o mesmo exemplo, enquanto um Gato pode correr, o Frajola realmente corre para tentar pegar o Piu-piu.

Projetar um sistema orientado a objetos consiste em definir suas classes e como elas colaboram para conseguir implementar os requisitos de uma aplicação. É importante ressaltar que não é só porque uma linguagem orientada a objetos está sendo utilizada, que se estará modelando de forma orientada a objetos. Se suas classes não representam abstrações do sistema e são apenas repositórios de funções, você na verdade está programando segundo o paradigma estruturado.

## Herança

Se uma aplicação precisa de um conjunto de objetos, devem haver classes que abstraem esses conceitos. Porém, esses mesmos objetos podem precisar ser tratados em partes diferentes do software a partir de diferentes níveis de abstração. O Garfield e o Frajola podem ser tratados como gatos em uma parte do sistema, porém outra, que precisar também lidar com os objetos Pica-pau e Mickey Mouse, pode precisar de um conceito mais abstrato, como animal ou personagem.

A **herança** é uma característica do paradigma orientado a objetos que permite que abstrações possam ser definidas em diversos níveis. Considerando que você tem uma classe, ela pode ser especializada por uma outra classe que irá definir um conceito mais concreto. Por outro lado, um conjunto de classes pode ser generalizado por uma classe que representa um conceito mais abstrato. Quando uma classe estende outra, ela não só herda a estrutura de dados e o comportamento da superclasse, mas também o contrato que ela mantém com os seus clientes.

Um dos grandes desafios da modelagem orientada objetos é identificar os pontos onde essa abstração deve ser utilizada e será aproveitada de forma adequada pelo sistema. Pelo fato de um sistema de software ser uma representação de algum processo que acontece no mundo real, isso não significa que todas as abstrações existentes precisam ser representadas no software. Um avião, por exemplo, seria representado de forma completamente diferente em um simulador de voo, em um sistema de controle de tráfego aéreo e no sistema de vendas de uma companhia aérea. Saber qual a representação e as abstrações mais adequadas é um grande desafio.

## Encapsulamento

Sempre que falo sobre encapsulamento, inicio com a frase "*A ignorância é uma bênção!*". Não me entendam mal, mas com a complexidade das coisas que lidamos nos dias de hoje, é realmente muito bom não precisar saber como elas funcionam para utilizá-las. Fico feliz em poder assistir minha televisão, mudar seu canal, aumentar seu volume e não ter a menor ideia de como essas coisas estão funcionando. Imagine como seria complicado poder utilizar novas tecnologias se elas exigissem uma compreensão profunda de seu funcionamento interno.

Essa mesma questão acontece com software. Antes da programação estruturada, o desenvolvedor precisava conhecer os detalhes de implementação de cada parte do código. Com a criação das funções, houve um certo avanço em relação a divisão do código, porém a utilização de algumas funções ainda demandava um conhecimento maior.

mento sobre seu funcionamento interno, como que variáveis elas estão acessando e atualizando.

O **encapsulamento** é um conceito importante da orientação a objetos que diz que deve haver uma separação entre o comportamento interno de uma classe com a interface que ela disponibiliza para os seus clientes. Isso permite que o desenvolvedor que utilizar uma classe precise saber somente como interagir com ela, abstraindo seu comportamento interno. Este é um conceito muito poderoso que permite o desacoplamento entre as classes, podendo ser utilizado para uma melhor divisão do software em módulos.

Os métodos *getters* e *setters* que são utilizados respectivamente para acessar e modificar propriedades em um objeto são um exemplo do uso do encapsulamento. Para os clientes da classe, apenas uma informação está sendo recuperada ou modificada, porém a implementação por trás pode realizar outras coisas. Ao recuperar uma informação, ela pode ser calculada a partir de outros dados, e ao modificar uma informação, pode haver uma validação dos dados. É importante lembrar que o uso desses métodos de acesso é apenas um começo para começar o encapsulamento de uma classe. A estrutura de dados pode ser utilizada e manipulada também por outros métodos de forma totalmente transparente a classe cliente.

A linguagem Java provê as interfaces como um recurso de linguagem, que permite que a definição do contrato externo da classe possa ser feita de forma separada da implementação. Quando uma classe implementa uma interface é como se estivesse assinando um documento, no qual ela se compromete a implementar seus métodos. Dessa forma, os clientes da classe não precisam conhecer a classe onde está a implementação, mas apenas a interface. Adicionalmente, diferentes implementações podem ser utilizadas pelo cliente sem a modificação de seu código.

### INTERFACE OU CLASSE ABSTRATA?

Várias vezes já me fizeram a pergunta: “*quando devo utilizar uma classe abstrata e quando devo utilizar uma interface?*”. Tanto as classes abstratas e quanto as interfaces podem definir métodos abstratos que precisam ser implementados pelas classes que respectivamente a estende ou implementa. Porém apenas as classes abstratas podem possuir métodos concretos e atributos. Apesar dessa diferença, a resposta para pergunta é mais conceitual do que relacionada com questões de linguagem.

Quando a abstração que precisar ser criada for um conceito, ou seja, algo que possa ser refinado e espacializado, deve-se utilizar uma classe abstrata. Quando a abstração é um comportamento, ou seja, algo que uma classe deve saber fazer, então a melhor solução é a criação de uma interface. Imagine um jogo no qual existem naves que se movem. Se sua abstração representa uma nave, então você está representando um conceito e deve utilizar uma classe abstrata. Por outro lado, se sua abstração representa algo que se move, então o que está sendo abstraído é um comportamento e a melhor solução é usar uma interface.

## Polimorfismo

O **polimorfismo** é na verdade uma consequência da utilização de herança e da utilização de interfaces. Não faria sentido utilizar herança para criar novas abstrações se o objeto não pudesse ser visto como uma dessas abstrações. Equivalente mente, não faria sentido haver uma interface se a classe não pudesse ser tratada como algo que possui aquele comportamento. É através do polimorfismo que um objeto pode ser visto como qualquer uma de suas abstrações.

A palavra polimorfismo significa “*múltiplas formas*”, o que indica que um objeto pode assumir a forma de uma de suas abstrações. Em outras palavras, qualquer objeto pode ser atribuído para uma variável do tipo de uma de suas superclasses ou para o tipo de suas interfaces. Isso é um recurso extremamente poderoso, pois um código pode utilizar uma classe que não conhece, se souber trabalhar com uma de suas abstrações. Isso pode ter um impacto tremendo na reutilização e desacoplamento das classes.

Um bom exemplo para entender polimorfismo é a interface Comparable que é

provida pela API padrão da linguagem Java. Os algoritmos de ordenação presentes na classe `Collections` sabem ordenar listas de qualquer objeto que implementa essa interface, o que inclui classes como `Number`, `String` e `Date`. Dessa forma, se uma classe da sua aplicação implementar essa interface, então os algoritmos saberão ordenar uma lista de suas instâncias. O mais interessante é que esse algoritmo utiliza sua classe, mesmo tendo sido desenvolvido muito antes dela existir.

## 1.2 MAS ESSES CONCEITOS NÃO SÃO SUFICIENTES?

Sem mais delongas, já irei responder que não são suficientes. Na verdade, compreender bem esses conceitos é apenas o primeiro passo para a realização de um projeto orientado a objetos. Entender o significado da herança e saber como implementá-la na linguagem de programação não é suficiente para saber em que situações o seu emprego é adequado. Saber como uma classe implementa uma interface não basta para saber quando seu uso irá ajudar a resolver um problema.

Realizar a modelagem de um software é um imenso jogo de raciocínio, como em um tabuleiro de xadrez. Da mesma forma que muitas vezes é preciso entregar uma peça para se atingir um objetivo, no projeto de um software é comum abrir mão de certas características para se obter outras. Por exemplo, ao se adicionar uma verificação de segurança, o desempenho pode ser degradado. Ou mesmo, ao se adicionar flexibilidade, a complexidade de uso da solução pode aumentar.

São raros os benefícios que vêm sem nenhuma consequência negativa. Durante o projeto de um software o desenvolvedor está sentado em uma mesa de negociação, onde do outro lado estão sentados os requisitos que precisam ser atendidos. Ao colocar na mesa uma solução, deve ser avaliado os ganhos e as perdas obtidos para tentar equilibrar os atributos de qualidade do sistema. Muitas vezes é preciso combinar mais de uma solução para que o benefício de uma compense a desvantagem da outra.

Nesse jogo, as soluções que podem ser empregadas são as principais armas do desenvolvedor. O desenvolvedor com menor experiência conhece uma quantidade menor de soluções, o que muitas vezes o leva a adotar um projeto inadequado às necessidades do software. A elaboração de uma solução partindo do zero trás de forma intrínseca o risco da inovação, e muitas vezes acaba-se deixando de considerar outras alternativas.

Mas como conhecer diversas soluções sem precisar passar por vários anos alternando entre escolhas certas e erradas? Como saber o contexto em que essas soluções

são adequadas e quais são as contrapartidas dos benefícios da solução? Felizmente existe uma forma na qual desenvolvedores mais experientes expressam seu conhecimento em um determinado domínio. E então eu apresento os padrões de projeto, os *Design Patterns*!

### 1.3 O PRIMEIRO PROBLEMA: CÁLCULO DO VALOR DO ESTACIONAMENTO

Para ilustrar como o livro vai proceder, essa seção vai apresentar um primeiro problema de projeto, a partir do qual serão exploradas as alternativas de solução. Apesar de grande parte dos problemas ser de sistemas fictícios e criados para ilustrar a aplicação dos padrões, as situações apresentadas por eles são realistas e aplicáveis em softwares reais.

Considere o sistema de um estacionamento que precisa utilizar diversos critérios para calcular o valor que deve ser cobrado de seus clientes. Para um veículo de passeio, o valor deve ser calculado como R\$2,00 por hora, porém caso o tempo seja maior do que 12 horas, será cobrada uma taxa diária, e caso o número de dias for maior que 15 dias, será cobrada uma mensalidade. Existem também regras diferentes para caminhões, que dependem do número de eixos e do valor da carga carregada, e para veículos para muitos passageiros, como ônibus e vans. O código a seguir apresenta um exemplo de como isto estava desenvolvido.

```
public class ContaEstacionamento {

    private Veiculo veiculo;
    private long inicio, fim;

    public double valorConta() {
        long atual = (fim==0) ? System.currentTimeMillis() : fim;
        long periodo = inicio - atual;
        if (veiculo instanceof Passeio) {
            if (periodo < 12 * HORA) {
                return 2.0 * Math.ceil(periodo / HORA);
            } else if (periodo > 12 * HORA && periodo < 15 * DIA) {
                return 26.0 * Math.ceil(periodo / DIA);
            } else {
                return 300.0 * Math.ceil(periodo / MES);
            }
        }
    }
}
```

```
    } else if (veiculo instanceof Carga) {  
        // outras regras para veículos de carga  
    }  
    // outras regras para outros tipos de veículo  
}  
  
}
```

Como é possível observar, o código utilizado para calcular os diversos condicionais é complicado de se entender. Apesar de apenas parte da implementação do método ter sido apresentada, é possível perceber que a solução utilizada faz com que o método `valorConta()` seja bem grande. As instâncias da classe `ContaEstacionamento` relacionadas com os veículos que estão no momento estacionados são exibidas para o operador e tem seu tempo atualizado periodicamente.

Até então, o próprio desenvolvedor sabia que o código estava ruim, mas como sabia que o código estava funcionando, preferiu deixar da forma que está. Porém, o software começou a ser vendido para outras empresas e outras regras precisariam ser incluídas. Alguns municípios possuem leis específicas a respeito do intervalo de tempo para o qual um estacionamento definir sua tarifa. Além disso, diferentes empresas podem possuir diferentes critérios para cobrar o serviço de seus clientes.

A solução da forma como está não irá escalar para um número grande de regras! O código que já não estava bom poderia crescer de uma forma descontrolada e se tornar não gerenciável. O desenvolvedor sabia que precisaria refatorar esse código para uma solução diferente. O que ele poderia fazer?

## Usando Herança

A primeira ideia que o desenvolvedor pensou foi na utilização de herança para tentar dividir a lógica. Sua ideia era criar uma superclasse onde o cálculo do valor da conta seria representado por um método abstrato, o qual seria implementado pelas subclasses com cada uma das regras. A Figura 1.1 apresenta como seria essa solução.

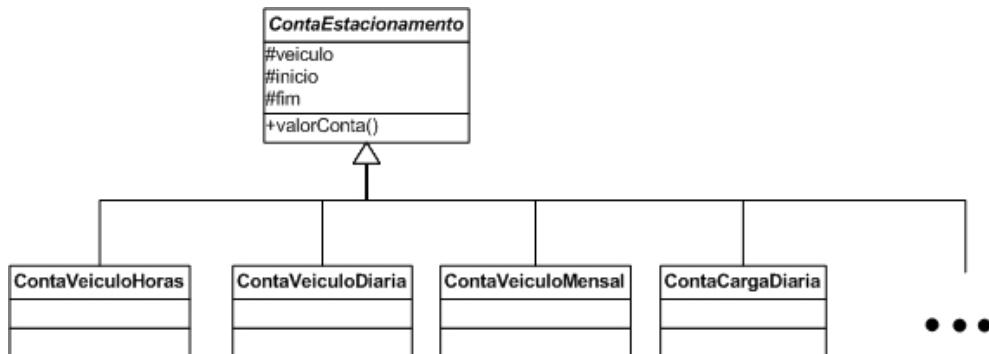


Figura 1.1: Utilização de herança para dividir a lógica

Um dos problemas dessa solução é a explosão de subclasses que vai acontecer, devido às várias possibilidades de implementação. Outra questão é que utilizando herança, depois não é possível alterar o comportamento depois que classe foi instanciada. Por exemplo, depois de 12 horas que um veículo estiver estacionado, o comportamento do cálculo da tarifa deve ser alterado da abordagem por hora para a abordagem por dia. Quando se cria o objeto como sendo de uma classe, para mudar o comportamento que ela implementa, é preciso criar uma nova instância de outra classe. Isso é algo indesejável, pois a mudança deveria acontecer dentro da própria classe!

A segunda solução que o desenvolvedor pensou foi utilizar a herança mas com uma granularidade diferente. Sua ideia era que cada subclasse possuísse o código relacionado a uma abordagem de cobrança para um tipo de veículo. Por exemplo, no caso acima, seria apenas uma subclasse para veículos de passeio e ela conteria os condicionais de tempo necessários. Dessa forma, a mesma instância seria capaz de fazer o cálculo.

Ao começar a seguir essa ideia, o desenvolvedor percebeu que nas subclasses criadas ocorria bastante duplicação de código. Um dos motivos é que a cada pequena diferença na abordagem, uma nova subclasse precisaria ser criada. Se a mudança fosse pequena, o resto do código comum precisaria ser duplicado. A Figura 1.2 ilustra essa questão. Uma das classes considera a tarifa de veículo de passeio como apresentado anteriormente e a outra considera que a primeira hora precisa ser dividida em períodos de 15 minutos (uma lei que existe no município de Juiz de Fora) e em seguida faz a cobrança por hora como mostrada anteriormente. Observe que se houvesse uma classe com os períodos de 15 minutos mais a cobrança por dia e por mês,

mais código ainda seria duplicado.

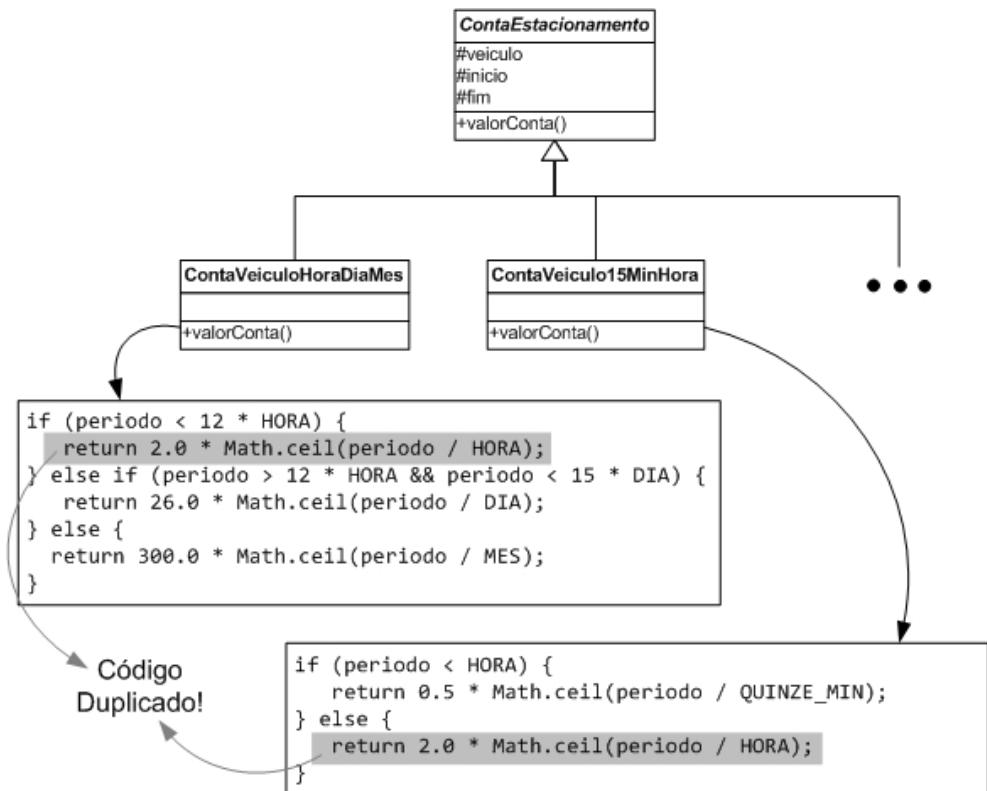


Figura 1.2: Duplicação de código com a implementação menos granular

Mas será que é possível ter uma forma de manter a mesma instância de `ContaEstacionamento` sem precisar de duplicar código?

## Pensando em Composição

Ao pensar melhor, o desenvolvedor decide que a herança não é a melhor abordagem para resolver o problema. Ele precisa de uma solução que permita que diferentes algoritmos de cálculo de tarifa possam ser utilizados pela classe. Adicionalmente, é desejável que não haja duplicação de código e que o mesmo algoritmo de cálculo possa ser utilizado para diferentes empresas. Além disso, uma classe deve poder iniciar a execução com um algoritmo e o mesmo ser trocado posteriormente.

Uma solução que se encaixa nos requisitos descritos é a classe

`ContaEstacionamento` delegar a lógica de cálculo para a instância de uma classe que a compõe. Dessa forma, para trocar o comportamento do cálculo do valor do estacionamento, basta trocar a classe que está o compondo. Essa classe também poderia ser parametrizada e ser reutilizada no contexto de diversas empresas. A figura 1.3 mostra o diagrama que representa essa solução.

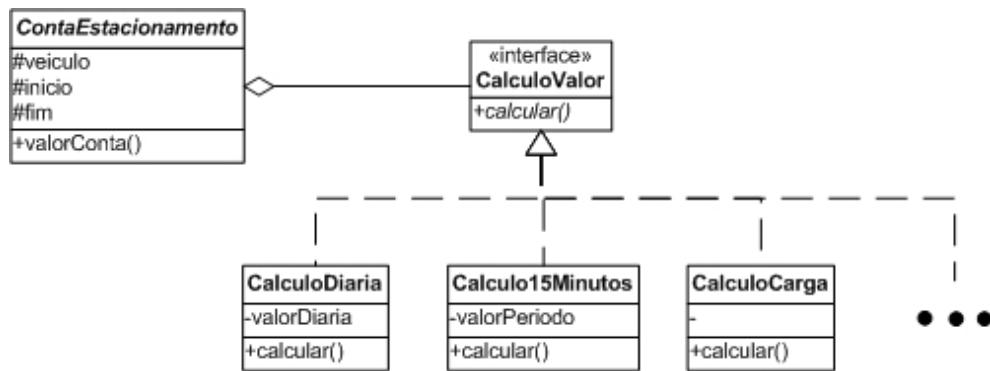


Figura 1.3: Utilizando a composição para permitir a troca do algoritmo

Agora apresentamos a classe `ContaEstacionamento` delegando para uma classe que a compõe o cálculo do valor do estacionamento. A interface `CalculoValor` abstrai o algoritmo de cálculo da tarifa. Observe que existe um método que permite que o atributo `calculo` seja alterado, permitindo a mudança desse algoritmo depois que o objeto foi criado.

```

public class ContaEstacionamento {

    private CalculoValor calculo;

    private Veiculo veiculo;
    private long inicio;
    private long fim;

    public double valorConta() {
        return calculo.calcular(fim-inicio, veiculo);
    }

    public void setCalculo(CalculoValor calculo){
        this.calculo = calculo;
    }
}
  
```

```
    }  
}
```

A seguir, a classe `CalculoDiaria` mostra um exemplo de uma classe que faz o cálculo da tarifa por dia. Observe que essa classe possui um atributo que pode ser utilizado para parametrizar partes do algoritmo. Dessa forma, quando a estratégia for alterada para o cálculo do valor por dia, basta inserir a instância dessa classe em `ContaEstacionamento`. Vale também ressaltar que essa mesma classe pode ser reaproveitada para diferentes empresas em diferentes momentos, evitando assim a duplicação de código.

*Listagem 1.1:*

```
public class CalculoDiaria implements CalculoValor {  
  
    private double valorDiaria;  
  
    public CalculoDiaria(double valorDiaria){  
        this.valorDiaria = valorDiaria;  
    }  
  
    public double calcular() {  
        return valorDiaria * Math.ceil(periodo / HORA);  
    }  
}
```

## Reconhecendo a Recorrência da Solução

Após implementar a solução, o desenvolvedor, orgulhoso de sua capacidade de modelagem, começa a pensar que essa mesma solução pode ser utilizada em outras situações. Quando houver um algoritmo que pode variar, essa é uma forma de permitir que novas implementações do algoritmo possam ser plugadas no software. Por exemplo, o cálculo de impostos como pode variar de acordo com a cidade poderia utilizar uma solução parecida.

Na verdade, ele se lembrou que já tinha visto uma solução parecida em um sistema que havia trabalhado. Como o algoritmo de criptografia que era utilizado para enviar um arquivo pela rede poderia ser trocado, existia uma abstração comum a todos eles que era utilizada para representá-los. Dessa forma, a classe que enviava o arquivo, era composta pela classe com o algoritmo. Sendo assim, o desenvolvedor

decidiu conversar com o colega a respeito da coincidência a respeito da solução utilizada. Sua surpresa foi quando ele disse que não era uma coincidência, pois ambos haviam utilizado o padrão de projeto Strategy.

### EU JÁ USEI ISSO E NEM SABIA QUE ERA UM PADRÃO!

Uma reação comum quando certas pessoas tem seu primeiro contato com padrões é ver que já utilizou aquela solução anteriormente. Isso é normal, pois o próprio nome “*padrão*” significa que aquela é uma solução que já foi utilizada com sucesso em diversos contextos. Nesse momento, algumas pessoas pensam: *Para que eu preciso saber os padrões se eu posso chegar a essas soluções sozinho?*.

Observe que para o desenvolvedor chegar a essa solução ele demorou um certo tempo. Mesmo assim, podem haver outras soluções que ele deixou de explorar e considerar e podem haver consequências que muitas vezes não são levadas em conta. Ao aprender padrões, o desenvolvedor adquire o conhecimento não apenas da estrutura empregada, mas sobre o contexto em que ele é utilizado e as trocas feitas para se equilibrar os requisitos. Dessa forma, não se engane achando que os padrões não irão te acrescentar conhecimento, pois mesmo os desenvolvedores mais experientes aprendem muito com eles .

## 1.4 STRATEGY: O PRIMEIRO PADRÃO!

“Por mais bonita que seja a estratégia, ocasionalmente deve-se olhar os resultados.”  
– - Sir Winston Churchill

O Strategy é um padrão que deve ser utilizado quando uma classe possuir diversos algoritmos que possam ser utilizados de forma intercambiável. A solução proposta pelo padrão consiste em delegar a execução do algoritmo para uma instância que compõe a classe principal. Dessa forma, quando a funcionalidade for invocada, no momento de execução do algoritmo, será invocado um método da instância que a compõe. A Figura 1.4 apresenta um diagrama que mostra a estrutura básica do padrão.

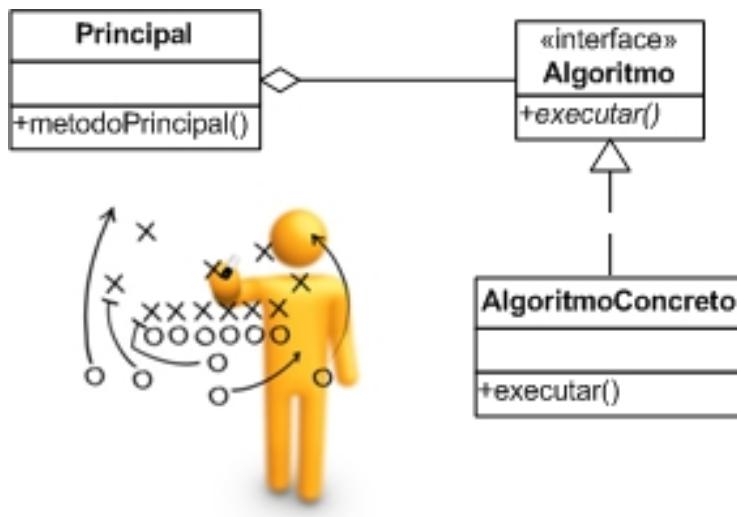


Figura 1.4: Estrutura do padrão Strategy

Uma das partes principais de um padrão diz respeito as suas consequências, pois é a partir delas que o desenvolvedor vai avaliar se essa é uma boa escolha ou não para seus requisitos. No caso do **Strategy**, a principal consequência positiva é justamente o fato do algoritmo poder ser alterado sem a modificação da classe. A partir dessa estrutura, novas implementações do algoritmo podem ser criadas e introduzidas posteriormente.

Outro ponto positivo do padrão está no fato da lógica condicional na classe principal ser reduzida. Como a escolha do algoritmo está na implementação do objeto que está compondo a classe, isso elimina a necessidade de ter condicionais para selecionar a lógica a ser executada. Outra consequência positiva está no fato da implementação pode ser trocada em tempo de execução, fazendo que o comportamento da classe possa ser trocado dinamicamente.

No mundo dos padrões, vale a expressão "*nem tudo são flores*", e é importante conhecer as consequências negativas do padrão que está sendo utilizado. No caso do **Strategy**, isso acontece no aumento da complexidade na criação do objeto, pois a instância da dependência precisa ser criada e configurada. Caso o atributo seja nulo, a classe pode apresentar um comportamento inesperado. Outro problema dessa solução está no aumento do número de classes: há uma para cada algoritmo, criando uma maior dificuldade em seu gerenciamento.

## 1.5 O QUE SÃO PADRÕES?

Um padrão descreve um conjunto composto por um contexto, um problema e uma solução. Em outras palavras, pode-se descrever um padrão como uma solução para um determinado problema em um contexto. Porém um padrão não descreve qualquer solução, mas uma solução que já tenha sido utilizada com sucesso em mais de um contexto. Exatamente por esse motivo que a descrição dos padrões normalmente sempre descreve alguns de seus usos conhecidos. Um padrão não descreve soluções novas, mas soluções consolidadas!

A ideia dos padrões vem do trabalho de Christopher Alexander na área de arquitetura de cidades e construções [?]. Neste livro, cada padrão trazia uma solução adequada a um problema que poderia ser reutilizada e adaptada para diversas situações. A disseminação dos padrões na comunidade de desenvolvimento de software iniciou-se com o conhecido livro "*Design Patterns: Elements of Reusable Object-Oriented Software*" [?]. Esse livro descrevia soluções de projeto orientado a objetos que são utilizadas até hoje por desenvolvedores de todo mundo. Esse livro também é conhecido como **GoF**, um acrônimo de *Gang of Four*, uma referência aos quatro autores do livro.

Para ser um padrão, uma solução não basta ser recorrente, mas precisa ser uma boa solução (caso contrário é um anti-padrão!). Além disso, é comum que ela traga outras partes, como a estrutura da solução, como funciona sua dinâmica e as consequências positivas e negativas de sua aplicação. Outra parte importante é o relacionamento com os outros padrões, pois mostra outros padrões que oferecem uma outra alternativa, ou padrões que podem complementar essa solução para compensar suas desvantagens.

Muitos desenvolvedores acham que o padrão se resume a sua estrutura, normalmente representada através de um diagrama de classes, e esquecem de todo o resto de sua descrição. Isso é um uso inadequado dos padrões, pois se a solução for utilizada no contexto errado, ela pode atrapalhar mais que ajudar. Se observarmos todos os padrões, é possível observar que a estrutura de alguns padrões são bastante similares, porém o problema que essas estruturas estão sendo utilizadas para resolver podem ser completamente diferentes. Por outro lado, a estrutura apresentada no padrão é apenas uma referência, pois é possível aplicar o padrão com diversas adaptações na estrutura de acordo com as necessidades da aplicação. É por esse motivo que existem fortes críticas a ferramentas que possuem uma abordagem de gerar a estrutura de padrões de forma automática.

Padrões não se refletem em pedaços de código ou componentes que são reuti-

lizados de forma igual em diversas aplicações, eles são um conhecimento que deve estar na cabeça dos desenvolvedores. A partir desse conhecimento, os desenvolvedores devem avaliar o contexto e o problema que querem resolver e adaptar e combinar padrões de forma a resolver o problema equilibrando as forças envolvidas. É a partir desse conhecimento que os padrões ajudam os desenvolvedores a desenvolverem sua habilidade em modelagem orientada a objetos.

### **QUANTO MAIS PADRÕES EU UTILIZAR, MELHOR VAI FICAR O MEU CÓDIGO?**

Claro que não! Como o padrão é uma solução para um problema, aplicar o padrão onde o problema não existe irá apenas complicar as coisas. Isso seria como para uma pessoa que acabou de aprender a usar um martelo, ver todos os problemas como se fosse um prego. Por mais que um parafuso possa parecer com um prego, a solução é bem diferente.

O mesmo vale para os padrões! Não é só porque você aprendeu um novo padrão, que precisa ir utilizando ele em todas as soluções. Lembre-se que eles também possuem consequências negativas que podem sobrepor as vantagens em alguns casos. Procure possuir diversos padrões em sua caixa de ferramentas e utilizar o mais adequado para a cada situação. A utilização desnecessária de padrões pode ser desastrosa e gerar uma sobre-engenharia do sistema, o que, no mínimo, dificulta a manutenção do código.

Os padrões também adicionam a equipe uma terminologia para se referir a soluções de modelagem, criando um vocabulário compartilhado. Dessa forma, quando alguém mencionar o nome de um padrão, todos saberão do que se trata. Esse nome não se refere apenas a estrutura, mas a todas informações agregadas ao padrão, como sua dinâmica e suas consequências.

Este livro não irá descrever os padrões em um formato tradicional, mas irá utilizá-los para exemplificar técnicas podem ser utilizadas para projetar um software. Cada capítulo irá focar em uma técnica ou em um tipo de problema, e então serão apresentados os padrões relacionados com exemplos que irão ilustrar sua implementação. Também serão discutidas as diferentes alternativas de solução e as consequências positivas e negativas trazidas por cada uma delas. O objetivo é apre-

sentar importantes técnicas de projeto orientado a objetos a partir da aplicação de padrões.

Todos os padrões apresentados no livro serão apresentados utilizando este estilo, então toda vez que vir um nome nesse estilo, saiba que está fazendo referência a um padrão. O nome dos padrões serão mantidos em inglês devido ao fato de serem referenciados dessa forma pela comunidade de desenvolvimento de software no Brasil, sendo que para muitos deles não existe uma tradução adequada. Os capítulos do livro podem ser lidos de forma independente, porém eles podem fazer referência a padrões explicados em capítulos anteriores. Para desenvolvedores inexperientes na aplicação de padrões, recomenda-se a leitura dos capítulos na ordem de apresentação.

Grande parte dos padrões utilizados nesse livro são os contidos no [?], porém importantes padrões de outras fontes também estão incluídos. Quando nenhuma referência for provida com a primeira citação ao padrão, isso significará que é um padrão desse livro.

## Chegando aos Padrões pela Refatoração

Existem basicamente duas formas para se implementar os padrões no seu código. A primeira delas, é a introdução do padrão na modelagem antes do código ser desenvolvido. Nesse caso, é feita uma análise preliminar do problema, onde é identificado o contexto em que é adequada a aplicação do padrão. A outra abordagem é através da refatoração [?], uma técnica no qual o código é reestruturado de forma a não haver modificação de seu comportamento. Nesse contexto, o código já foi desenvolvido porém apresenta algum tipo de deficiência em termos de projeto, situação conhecida como **mau cheiro**. Sendo assim, o código vai sendo alterado em pequenos passos até que se chegue ao padrão alvo.

Saber como utilizar refatoração para se chegar a padrões é uma técnica importante [?], pois nem sempre a solução mais adequada é a que é empregada na primeira vez. Esse livro irá apresentar, juntamente com os padrões, como pode surgir a necessidade de sua aplicação em um código existente e quais os passos de refatoração que podem ser feitos para sua implementação. No exemplo apresentado nesse capítulo, foi mostrado um código problemático já existente que foi refatorado para implementar o padrão **Strategy**. Os passos da refatoração para esse padrão serão mostrados no Capítulo 3.

## 1.6 COMO O LIVRO ESTÁ ORGANIZADO

Esta seção irá fechar esse capítulo inicial, falando um pouco sobre a organização do livro. Este livro apresenta os padrões de uma forma distinta de outros livros. Os capítulos são organizados de acordo com o princípio de design utilizado pelo padrão. Dessa forma, os padrões servirão para exemplificar tipos de problema que podem ser resolvidos com aquela técnica. Serão apresentados exemplos, que além de mostrar a aplicação do padrão de forma prática, também irão ilustrar como eles podem ser combinados.

O Capítulo 2 começa explorando de forma mais profunda a utilização de herança e quais são os padrões que fazem uso desse princípio para permitir a extensão de comportamento. De forma complementar, o Capítulo 3 contrasta o uso da herança com o uso da composição, ressaltando as principais diferenças entre elas. Nesse ponto, diversos padrões que fazem uso da composição serão apresentados, inclusive de forma combinada com a herança.

Seguindo a linha de raciocínio, o Capítulo 4 apresenta a composição recursiva, e como ela pode ser utilizada para facilitar o reaproveitamento de código. A partir dos padrões que utilizam esse princípio é possível combinar o comportamento de diversas classes de formas diferentes, obtendo uma grande quantidade de possibilidades de comportamento final.



## CAPÍTULO 2

# Reuso Através de Herança

*“Entidades de software devem ser abertas a extensão, mas fechadas a modificação.”*

– Bertrand Meyer

A herança é uma das principais funcionalidades de linguagens orientadas a objetos. É a partir dela que é possível grande parte do potencial de reúso em uma modelagem orientada a objetos. O problema é que muita gente que diz isso para por aí ainda busca o reuso através da herança de forma errada.

A primeira pensamento que normalmente temos ao se estender uma classe é reutilizar a lógica da superclasse na subclasse. Será que isso é mesmo verdade? O que uma subclasse pode reutilizar da superclasse? A estrutura de dados? Seus métodos? A reutilização da estrutura de dados na herança não é muito importante, pois simplesmente instanciando e utilizando uma classe isso é possível de ser feito. A utilização dos métodos da superclasse pela subclasse é equivalente ao reuso de funções na programação estruturada, não sendo também grande novidade.

O potencial de reuso possível com herança está em outro local! Ele pode estar sim no reúso de código da superclasse, porém não é com a subclasse chamando mé-

todos da superclasse, mas com a superclasse chamando código da subclasse. Quando um método da superclasse chama um método que pode ou deve ser implementado na subclasse, isso permite que um mesmo algoritmo possa ser reutilizado com passos alterados. Essa flexibilidade aumenta o potencial de reutilização pois permite a sua adaptação para necessidades mais específicas.

Outro local onde o código pode ser reusado é nas classes que utilizam uma variável com o tipo da superclasse. Nesse caso, como as instâncias das subclasses podem ser atribuídas a essa variável, é possível adaptar o comportamento segundo a instância utilizada. Resumindo em apenas uma palavra: polimorfismo! Por exemplo, o algoritmo de ordenação de listas pode ser reutilizado em diversas aplicações para ordenação de listas de diversas classes. Isso só é possível pois todas as classes compartilham a mesma abstração do tipo Comparable.

O objetivo desse capítulo é mostrar como a herança pode ser utilizada em um design orientado a objetos para permitir adaptação de comportamento e consequentemente um maior reuso. Isso será feito apresentando padrões que utilizam os princípios da herança descritos e como eles podem ser utilizados para criação de soluções.

## 2.1 EXEMPLO DE PADRÃO QUE UTILIZA HERANÇA - NULL OBJECT

*“O que é invisível para nós é também crucial para nosso bem estar.”*

– Jeanette Winterson

Para começar a falar de herança, essa seção apresenta um padrão bem simples, porém bastante útil. Apesar de não ter sido incluído no GoF, em uma entrevista recente, um dos autores confessou que o colocaria em uma hipotética segunda edição do livro [?]. Ele irá ilustrar como com o uso da herança é possível “enganar” o código que utiliza a classe, introduzindo um novo comportamento que irá eliminar a necessidade do uso de condicionais. Como para o código cliente o comportamento da classe é invisível, o dinamismo desse comportamento pode ser a chave para se lidar com situações diferentes.

Vou começar citando uma situação que certamente muitos já viram em algum código que trabalharam. A classe `ApresentacaoCarrinho` apresentada na listagem a seguir chama o método `criarCarrinho()` da classe `CookieFactory` para recuperar um carrinho previamente criado pelo usuário a partir dos cookies armazenados em seu navegador. A partir dos valores recuperados, são atribuídos atributos para

a exibição das informações do carrinho na parte superior da página do usuário. O grande problema desse código é que caso nenhum carrinho tenha sido criado, ele retorna um valor nulo. Daí é preciso adicionar condicionais para configurar os valores adequados para quando o carrinho for nulo.

```
public class ApresentacaoCarrinho{  
  
    public void colocarInformacoesCarrinho(HTTPServletRequest request) {  
        Carrinho c = CookieFactory.criarCarrinho(request);  
        if(c != null) {  
            request.setAttribute("valor", c.getValor());  
            request.setAttribute("qtd", c.getTamanho());  
        } else {  
            request.setAttribute("valor", 0.0);  
            request.setAttribute("qtd", 0);  
        }  
        if(request.getAttribute("username") == null) {  
            if (c != null) {  
                request.setAttribute("userCarrinho",  
                    c.getNomeUsuario());  
            } else {  
                request.setAttribute("userCarrinho",  
                    "<a href=login.jsp>Login</a>");  
            }  
        } else {  
            request.setAttribute("userCarrinho",  
                request.getAttribute("username"));  
        }  
    }  
}
```

Se esse problema se tornar recorrente, ele pode se tornar uma verdadeira catástrofe na legibilidade do código de uma aplicação. É comum ver condicionais repetidos para tratar em vários pontos a possibilidade de uma determinada variável ser nula. Esses condicionais garantem a segurança do código apenas naquele ponto, pois nada garante que esse valor será tratado adequadamente em outros locais. Mas como então fazer para proteger minha aplicação de um `NullPointerException` sem precisar recorrer a esse tipo de condicional?

## Criando uma Classe para Representar Valores Nulos

O padrão Null Object [?] propõe a criação de uma classe para representar objetos nulos em uma aplicação. Essa classe deve estender a classe original e implementar seus métodos de forma a executar o comportamento esperado da aplicação quando um valor nulo for recebido. Dessa forma, ao invés de se retornar um valor nulo, retorna-se uma instância dessa nova classe.

A classe CarrinhoNulo, apresentada na listagem a seguir, exemplifica a criação de um Null Object para o contexto apresentado. Observe que os métodos retornam exatamente os valores que eram configurados para o caso do carrinho ser nulo. Nesse exemplo, apenas valores são retornados, porém em outros casos é preciso ter uma lógica a ser executada dentro dos métodos.

```
public class CarrinhoNulo extends Carrinho{  
    public double getValor(){  
        return 0.0;  
    }  
  
    public int getTamanho(){  
        return 0;  
    }  
  
    public String getNomeUsuario(){  
        return "<a href=login.jsp>Login</a>";  
    }  
}
```

Na listagem a seguir, é possível observar como o código fica mais simples com a eliminação da parte responsável pelo tratamento dos valores nulos. Outra coisa que precisaria ser alterada é a própria classe CookieFactory que deve retornar uma instância de CarrinhoNulo ao invés de null quando nenhum carrinho do usuário for encontrado.

```
public class ApresentacaoCarrinho{  
    public void colocarInformacoesCarrinho(HTTPServletRequest request) {  
        Carrinho c = CookieFactory.criarCarrinho(request);  
        request.setAttribute("valor", c.getValor());  
        request.setAttribute("qtd", c.getTamanho());  
  
        if(request.getAttribute("username") == null) {
```

```
        request.setAttribute("userCarrinho", c.getNomeUsuario());
    } else {
        request.setAttribute("userCarrinho",
            request.getAttribute("username"));
    }
}
}
```

Uma consequência interessante da aplicação desse padrão é que ele resolve o problema do tratamento de valores nulos em qualquer ponto da aplicação que utilize essa classe. Por mais que o método apresentado no exemplo tratasse esses valores, a mesma situação pode acontecer em partes do código que não o fazem, gerando a possibilidade de erro. Apesar das vantagens, o tratamento dos valores nulos não ficam explícitos e isso pode gerar uma certa confusão na hora de ler e dar manutenção nesse código.

## O Uso da Herança no Null Object

Uma das coisas interessantes desse exemplo é como que o código que utilizava o `Null Object` desconhecia completamente que ele estava ali. Ele conhecia apenas a interface da superclasse, mas utilizou as funcionalidades da subclasse. Na verdade, esse é o principal motivo para a utilização da herança no projeto de um software: o uso do polimorfismo. Com ele é possível reutilizar o código cliente para diversas implementações. Inclusive, um bom teste para verificar se o uso de herança é adequado em um determinada situação, é ver se faz sentido substituir a implementação por uma de suas subclasses em todos os contextos.

## HERDAR OU NÃO HERDAR DE JPanel? EIS A QUESTÃO!

Eu vejo essa questão da utilização ou não da herança é no desenvolvimento de aplicações desktop em Swing. Ao se criar uma nova tela para aplicação, uma dúvida comum é se essa classe deve estender a classe JPanel ou simplesmente utilizá-la. A extensão de JPanel pode fazer sentido a princípio, visto que a nova classe será um painel (obedecendo a famosa regra "*é um*") e que pode ser adicionada em uma interface gráfica.

Por outro lado, se pensarmos nos princípios da herança, uma subclasse deve poder ser utilizada no lugar de sua superclasse. Raciocinando dessa forma, é muito fácil pensar em diversas situações em que a classe com a tela de uma aplicação não faria sentido de ser utilizada no lugar de um painel genérico. Isso também quebraria o contrato da superclasse, visto que diversos métodos, como para adição de componentes e configuração de layout deixariam de fazer sentido.

Somente verificar se a subclasse é *uma* superclasse pode não ser suficiente. Antes de utilizar herança verifique se o polimorfismo faz sentido, ou seja, se qualquer subclasse pode ser utilizada no lugar da superclasse. Em caso negativo, isso é um indício que a herança está sendo utilizada de forma inadequada. Esse é conhecido como o **Princípio de Substituição de Liskov**, que defende que se uma classe é um subtipo de outra, então os objetos dessa classe podem ser substituídos pelos objetos do subtipo sem que seja necessário alterar as propriedades do programa.

O NullObject é um padrão que demonstra bem essa características da herança, pois ele permite que o código cliente possa ser utilizado, mesmo para o caso de um objeto nulo. Observe que outras implementações de Carrinho, como um que talvez acessa informações remotamente, também poderiam ser retornadas pela fábrica e utilizadas livremente.

## 2.2 HOOK METHODS

Um importante uso que pode ser feito da herança é para permitir a especialização de comportamento. Dessa forma, a superclasse pode fornecer uma base para uma

determinada funcionalidade, a qual invoca um método o qual somente é definido pela superclasse. Esse método funciona como um ponto de extensão do sistema é chamado de método-gancho, ou em inglês, **hook method**.

A Figura 2.1 representa o conceito de *hook method*. A superclasse possui um método principal público que é invocado pelos seus clientes. Esse método delega parte de sua execução para o *hook method*, o qual é um método abstrato que deve ser implementado pela subclasse. Esse método funciona como um “gancho” no qual uma nova lógica de execução para a classe pode ser “pendurada”. Cada subclasse implementa esse método provendo uma lógica diferente. Como essa lógica pode ser invocada a partir do mesmo método público, definido na superclasse, os *hook methods* permitem que de acordo com a subclasse instanciada, o objeto possua um comportamento diferente.

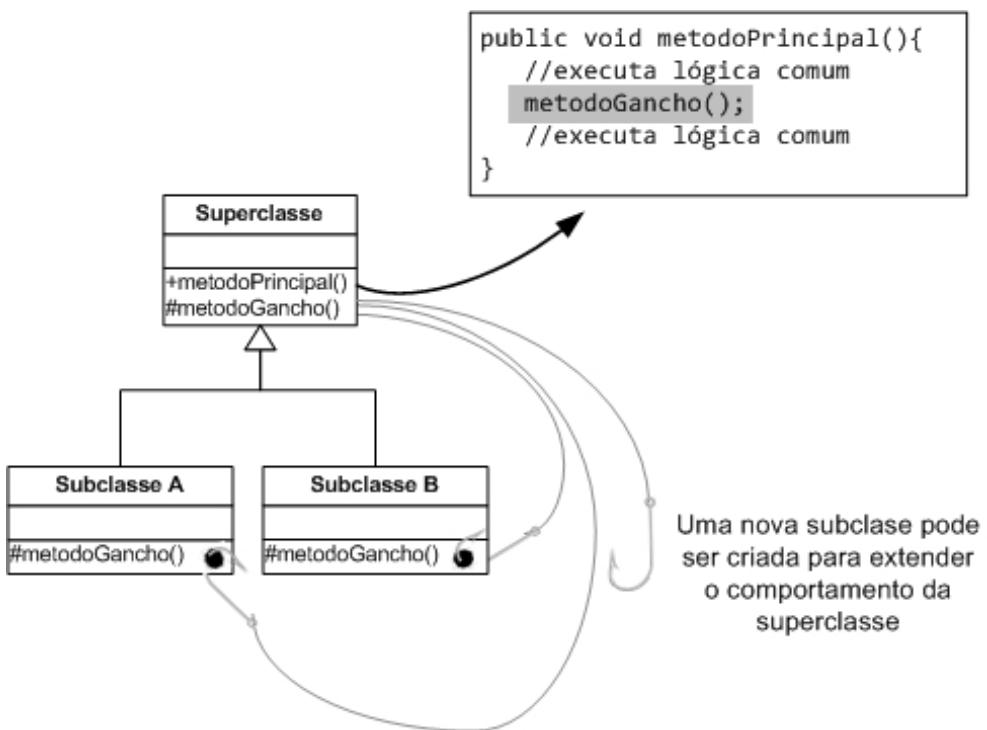


Figura 2.1: Representação de Hook Methods

Essa prática é muito utilizada em frameworks para permitir que as aplicações possam especializar seu comportamento para seus requisitos. Nesse caso, o fra-

mework provê algumas classes com *hook methods* que precisam ser especializadas pela aplicação. Sendo assim, a aplicação deve estender essa classe e implementar o *hook method* de forma a inserir o comportamento específico de seu domínio. Imagine, por exemplo, um framework que realiza o agendamento de tarefas. Usando essa estratégia ele poderia prover uma classe para ser estendida, na qual precisaria ser implementado um *hook method* para a definição da tarefa da aplicação. Enquanto isso, na classe do framework estaria a lógica de agendamento que chamaria o método definido pela aplicação no momento adequado.

Um exemplo de uso dessa técnica está na Servlets API [?], na qual um servlet precisa estender a classe `HTTPServlet`. Essa classe possui o método `service()` que é invocado toda vez que ele precisa tratar uma requisição HTTP. Esse método chama outros métodos, como `doPost()` ou `doGet()`, que precisam ser implementados pela subclasse. Nesses métodos ela deve inserir a lógica a ser executada para tratar a requisição recebida. Nesse caso, esses métodos possuem uma implementação *default* vazia e precisam ser implementados somente se necessário.

## 2.3 REVISANDO MODIFICADORES DE MÉTODOS

Quando aprendemos o básico da linguagem, aprendemos modificadores de acesso e outros tipos de modificadores de métodos e qual o seu efeito nas classes. O que nem todos compreendem é quando cada um deles precisa ser utilizado e qual o tipo de mensagem que se passa com cada um em termos de design. A seguir estão relacionados alguns modificadores de acesso e qual o recado que uma classe manda para duas subclasses quando utiliza cada um deles:

- **abstract**: Um método abstrato precisa obrigatoriamente ser implementado em uma subclasse concreta. Isso significa que esse é um *hook method* que foi definido na superclasse e obrigatoriamente precisa ser definido.
- **final**: De forma contrária, um método do tipo final não pode ser sobreescrito pelas subclasses e por isso nunca irá representar um *hook method*. Esses métodos representam funcionalidades da classe que precisam ser imutáveis para seu bom funcionamento. Costumam ser os métodos que invocam os *hook methods*.
- **private**: Os métodos privados só podem ser invocados dentro da classe que são definidos. Dessa forma, eles representam métodos de apoio internos da classe e nunca poderão ser substituídos pela subclasse como um *hook method*.

- `protected` e `public`: Todos os métodos públicos e protegidos, desde que não sejam `final`, são candidatos a *hook method*. É isso mesmo! Qualquer método que pode ser sobreescrito na subclasse pode ser utilizado para a inserção de comportamento. Algumas classes possuem implementações vazias, caracterizando

Ao criar uma estrutura de classes, é importante colocar os modificadores corretos nos métodos de forma a passar a mensagem correta para as classes que forem estendê-la. Por exemplo, se um método que coordena a chamada de *hook methods* for sobreescrito, eles podem não ser chamados e algumas premissas assumidas na superclasse podem não ser mais verdadeiras. Por isso é preciso muito cuidado ao prover uma classe para ser estendida como parte da API de um componente ou framework.

## 2.4 PASSOS DIFERENTES NA MESMA ORDEM - TEMPLATE METHOD

*“No meio do caminho tinha uma pedra / tinha uma pedra no meio do caminho / tinha uma pedra / no meio do caminho tinha uma pedra.”*

– Carlos Drummond de Andrade

O principal padrão que utiliza *hook methods* como técnica é o `Template Method`. Este padrão é aplicável quando se deseja definir um algoritmo geral, que define uma série de passos para cumprir um requisito da aplicação. Porém, os passos desse algoritmo podem variar e é desejável que a estrutura da implementação forneça uma forma para que eles possam ser facilmente substituídos.

Imagine, por exemplo, como seria se fossemos definir um algoritmo para as pessoas acordarem e irem ao trabalho. Esse algoritmo envolveria ações como acordar, ir ao banheiro, comer alguma coisa, trocar de roupa e se locomover até o trabalho. Dependendo da pessoa cada um dos passos pode ser diferente, como o tipo de roupa que colocam para ir ao trabalho ou o que comem pela manhã. Na locomoção ao trabalho, alguns podem ir de transporte público, outros podem ir em seu carro e até mesmo tem os que vão a pé ou de bicicleta. Por mais que cada um dos passos seja diferente, o algoritmo que os utiliza acaba sendo o mesmo.

### Estrutura do Template Method

Para entender a estrutura do `Template Method` vamos utilizar como metáfora algo familiar a muitas pessoas: modelos de documento. Conforme ilustrado na Fi-

gura 2.2, um modelo de documento define algumas partes que são fixas e outras partes que devem ser introduzidas quando o documento propriamente dito for criado. São lacunas que precisam ser completadas e que irão variar de documento para documento. O modelo de documento provê uma estrutura que pode ser facilmente reaproveitada, facilitando a criação do documento.

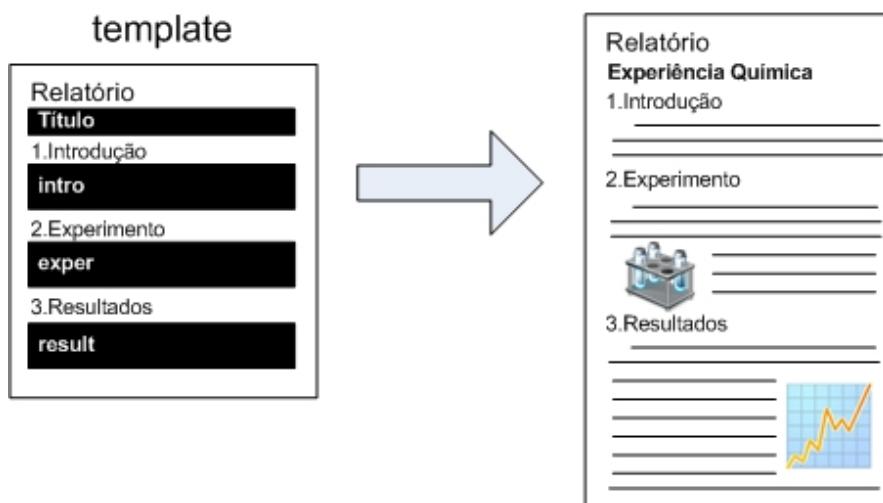


Figura 2.2: Uso de Templates para Documentos

De forma similar, um *Template Method* é um modelo de algoritmo que possui algumas partes fixas e algumas partes variáveis. As partes variáveis são lacunas que precisam ser completadas para que o algoritmo faça realmente sentido. As lacunas são representadas como *hook methods* que podem ser implementados nas subclasses. Caso seja uma lacuna obrigatória, o método deve ser definido como abstrato e caso a implementação seja opcional, o método pode ser concreto e normalmente possui uma implementação vazia. O algoritmo é representado através de um método na superclasse que coordena a execução dos *hook methods*.

A Figura 2.3 apresenta a estrutura do padrão *Template Method*. A *ClasseAbstrata* representa a superclasse que implementa o *TemplateMethod* e que define quais são os *hook methods*. A *ClasseConcreta* representa a classe que herda o *Template Method* da *ClasseAbstrata* e define uma implementação concreta dos *hook methods*. A classe representada como *Cliente* invoca o *metodoTemplate()*. Observe que apesar do tipo da variável ser do tipo da classe abstrata, o tipo instan-

ciado é o da subclasse que implementa os passos concretos do algoritmo.

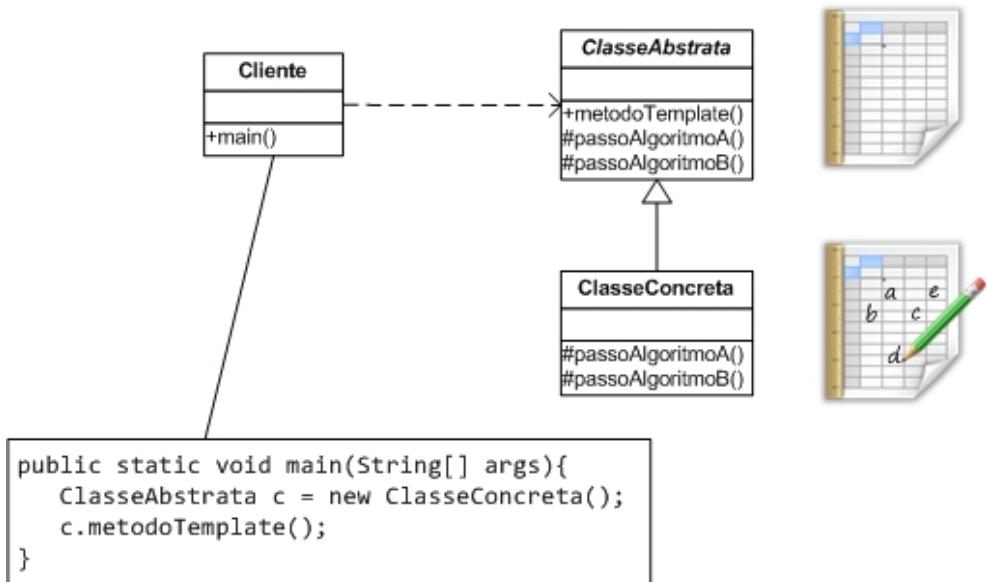


Figura 2.3: Estrutura do Padrão Template Method

### QUAL A DIFERENÇA ENTRE HOOK METHODS E O TEMPLATE METHOD?

Muitas pessoas nesse ponto podem estar achando que *Hook Methods* e o padrão *Template Method* são a mesma coisa. A grande diferença é que os *Hook Methods* são uma técnica para permitir a extensão de comportamento e o *Template Method*, como um padrão, é uma solução para um problema mais específico. Seria correto dizer que o *Template Method* utiliza *Hook Methods* em sua solução. O que é importante perceber é que o conceito de *Hook Method* é mais geral e inclusive é utilizado por outros padrões que serão vistos mais a frente nesse livro.

## Serializando Propriedades em Arquivos

Imagine que em uma aplicação precisamos pegar mapas de propriedades e serializar em arquivos. A questão é que o formato em que as propriedades precisam

ser estruturadas são diferentes. Por exemplo, pode ser possível estruturar as informações em arquivos XML e em arquivos de propriedades. Além disso, os arquivos podem precisar receber um pós-processamento, como uma criptografia ou uma compactação. Para o exemplo que será apresentado iremos considerar que existem duas possibilidades: a criação do arquivo XML compactado e a criação do arquivo de propriedades criptografado.

Como nesse caso temos um algoritmo base em que os passos podem ser modificados de acordo com a implementação, esse é um cenário adequado para implementação do padrão Template Method. A classe `GeradorArquivo`, apresentada na listagem a seguir, possui o método `gerarArquivo()` que define um algoritmo base para a criação do arquivo, invocando os dois métodos abstratos `processar()` e `gerarConteudo()`. Esse método possui o modificador `final` para que ele não possa ser sobreescrito nas subclasses. Note que o método `processar()` fornece uma implementação *default* que retorna o próprio array de bytes, representando o caso em que não existem processamento adicional após o arquivo ser gerado. Diferentemente, o método `gerarConteudo()` é abstrato e obrigatoriamente precisará ser implementado.

```
public abstract class GeradorArquivo {  
    public final void gerarArquivo(String nome,  
                                    Map<String, Object> propriedades)  
        throws IOException {  
        String conteudo = gerarConteudo(propriedades);  
        byte[] bytes = conteudo.getBytes();  
        bytes = processar(bytes);  
        FileOutputStream fileout = new FileOutputStream(nome);  
        fileout.write(bytes);  
        fileout.close();  
    }  
  
    protected byte[] processar(byte[] bytes) throws IOException{  
        return bytes;  
    }  
  
    protected abstract String  
        gerarConteudo(Map<String, Object> propriedades);  
}
```

As duas próximas listagens apresentam as classes `GeradorXMLCompactado` e `GeradorPropriedadesCriptografado` que implementam a classe `GeradorArquivo`,

fornecendo uma implementação para os passos do algoritmos definidos na superclasse. A primeira implementação, no método `gerarConteudo()` cria um arquivo XML no qual existe uma tag chamada `<properties>` e cada propriedade é um elemento dentro dessa tag. O método `processar()` usa a classe `ZipOutputStream` para gerar um arquivo compactado. Já na segunda listagem, o método `gerarConteudo()` cria a estrutura de um arquivo de propriedades em que cada linha possui o formato `propriedade=valor`. O método de processamento criptografa os bytes do arquivo utilizando um algoritmo simples chamado cifra de César, onde o valor de cada byte é deslocado de acordo com o parâmetro `delay`.

```
public class GeradorXMLCompactado extends GeradorArquivo {  
    protected byte[] processar(byte[] bytes) throws IOException {  
        ByteArrayOutputStream byteOut = new ByteArrayOutputStream();  
        ZipOutputStream out = new ZipOutputStream(byteOut);  
        out.putNextEntry(new ZipEntry("internal"));  
        out.write(bytes);  
        out.closeEntry();  
        out.close();  
        return byteOut.toByteArray();  
    }  
  
    protected String gerarConteudo(Map<String, Object> props) {  
        StringBuilder propFileBuilder = new StringBuilder();  
        propFileBuilder.append("<properties>");  
        for(String prop : props.keySet()){  
            propFileBuilder.  
                append("<" + prop + ">" + props.get(prop) + "</>" + prop + ">");  
        }  
        propFileBuilder.append("</properties>");  
        return propFileBuilder.toString();  
    }  
}  
  
public class GeradorPropriedadesCriptografado extends GeradorArquivo {  
    private int delay;  
  
    public GeradorPropriedadesCriptografado(int delay) {  
        this.delay = delay;  
    }  
  
    protected byte[] processar(byte[] bytes) throws IOException {
```

```

        byte[] newBytes = new byte[bytes.length];
        for(int i=0;i<bytes.length;i++){
            newBytes[i]= (byte) ((bytes[i]+delay) % Byte.MAX_VALUE);
        }
        return newBytes;
    }

protected String gerarConteudo(Map<String, Object> props) {
    StringBuilder propFileBuilder = new StringBuilder();
    for(String prop : props.keySet()){
        propFileBuilder.append(prop+"="+props.get(prop)+"\n");
    }
    return propFileBuilder.toString();
}
}

```

## Consequências do Uso do Template Method

Pelo exemplo apresentado e pela descrição do padrão, é possível perceber que com o Template Method podemos reaproveitar o código relativo a parte comum de um algoritmo, permitindo que cada passo variável possa ser definido na subclasse. Isso também é uma forma de permitir que a funcionalidade da classe que define o algoritmo básico seja estendida. Assim é possível definir uma funcionalidade mais geral onde pode ser facilmente incorporada a parte específica do domínio da aplicação. É importante lembrar que os modificadores adequados dos métodos devem ser utilizados para impedir que o contrato da superclasse com os seus clientes seja quebrado.

Porém, como foi dito no primeiro capítulo, o uso da herança nesse padrão também trás algumas limitações. A primeira é que a herança "*é uma carta que só pode ser jogada uma vez*", isso significa que uma classe que precise de comportamentos de duas outras classes, só poderá fazer o uso da herança para uma delas. Essa questão será melhor discutida no Capítulo 3. Outra questão é que depois que uma implementação for instanciada, não será mais possível alterar os passos do algoritmo.

Ao utilizar um padrão, é preciso avaliar para os requisitos de sua aplicação quais consequências pesam mais ou menos. A partir dessas informações é possível decidir se seu uso será ou não adequado. Ressalto que o maior problema de uma solução possui limitações é quando elas são desconhecidas pelo desenvolvedores, pois quando se tem consequência de sua existência é possível gerenciar o risco ou tratá-las, muitas

vezes a partir de outros padrões.

## 2.5 REFATORANDO NA DIREÇÃO DA HERANÇA

Muitas vezes, os requisitos que direcionam o desenvolvimento para o uso da herança não surgem todos ao mesmo tempo. Outras vezes, simplesmente não se visualiza no momento que se está programando, que aquele padrão pode ajudar na solução. Nesses casos, o padrão pode ser alcançado a partir de refatorações. A ideia é conseguir a partir de pequenos passos, ir modificando aos poucos a estrutura da solução, sem modificar o seu comportamento. Vale ressaltar que é extremamente recomendado que a refatoração seja apoiada por uma suíte de testes automatizados, que devem ser executados a cada passo da refatoração para verificar se o comportamento foi mantido.

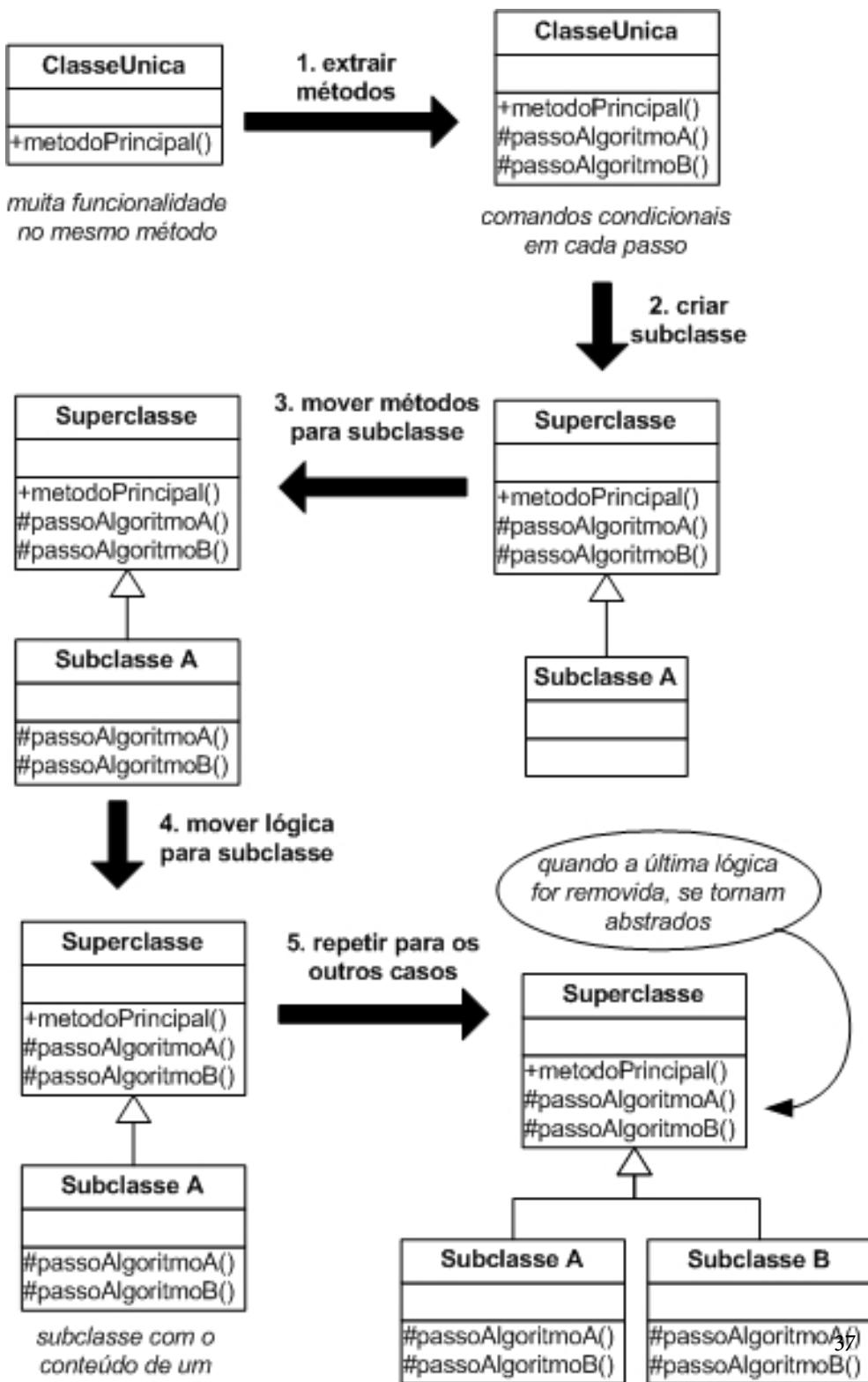
### DEMOLINDO TUDO QUE FOI CRIADO E RECONSTRUINDO DO ZERO

Um grande erro ao se refatorar um código é destruir toda a solução existente e reconstruir do zero uma nova solução. Isso é ruim pois durante um longo período de tempo perde-se a referência dos testes automatizados, que dão o feedback se o comportamento está sendo mantido inalterado. Um dos grandes desafios da refatoração é conseguir ir caminhando na direção da estrutura desejada, mantendo o comportamento a cada passo. No decorrer desse livro, serão apresentados os passos que devem ser seguidos para se refatorar a solução na direção dos padrões apresentados. Com isso, pretende-se mostrar como com uma sequência de pequenas mudanças, pode-se alcançar um grande impacto a qualidade do código e na modelagem da aplicação.

No caso de necessidade de se refatorar para o uso da herança, existem dois cenários diferentes. No primeiro, toda a funcionalidade foi acumulada em apenas uma classe, que possui um grande número de condicionais e um código de difícil manutenção. O segundo cenário ocorre quando funcionalidades com similaridades nos algoritmos são implementadas em diferentes classes, gerando duplicação de código.

A Figura 2.4 apresenta os passos para a refatoração quando uma única classe concentra em um método todas as possibilidades do algoritmo utilizando condi-

onais. A primeira etapa é extrair os métodos de acordo com os passos que devem ser realizados pelo algoritmo. Após a extração, cada método ainda possuirá a lógica condicional necessária para todos os possíveis caminhos. Na etapa seguinte do processo de refatoração é criada uma subclasse para representar uma alternativa de execução do algoritmo e os passos seguintes consistem em mover a funcionalidade para essa subclasse. Esse processo vai sendo repetido até que cada possível caminho seja movido para uma subclasse e somente a lógica mais geral esteja concentrada na superclasse.



Para que isso seja feito, a primeira coisa que deve ser feita é sobreescriver os métodos extraídos na nova subclasse criada. A princípio, esses métodos ainda irão delegar a execução da sua funcionalidade para a superclasse. Caso exista algum parâmetro setado na superclasse para a escolha dos passos do algoritmo, ele pode ser configurado de forma fixa na subclasse, de acordo com o que será implementado nela. A etapa seguinte consiste em mover para a subclasse a lógica relativa aos métodos com os passos do algoritmo. Deve ser retirado o condicional na superclasse juntamente com o código a executado e inserido no método da subclasse, que não deve mais delegar sua execução para superclasse. Esse procedimento deve ser feito método por método. Em seguida, os mesmos passos devem ser repetidos para a criação de novas subclasses, até que não este mais lógica nesses métodos da superclasse. No caso de não haver uma implementação *default*, eles devem se tornar abstratos.

A outra possibilidade de refatoração deve ser realizada quando existirem duas classes que implementam algoritmos similares e possuam código duplicado. Conforme está apresentado na Figura 2.5, a refatoração irá levar a refatoração na direção da definição de uma superclasse comum que implementa um Template Method. A primeira coisa que precisa ser feita é a extração de métodos com as partes que diferenciam um algoritmo do outro. Depois dessa etapa, o código interno dos métodos principais deve ser igual e todas as diferenças devem estar nos métodos extraídos. Em seguida, para uniformizar as classes, é preciso renomear os métodos para que fiquem com mesmo nome e mesma assinatura em ambas as classes (incluindo tipos de parâmetros e exceções). Deve-se buscar nomes que representem bem a parte do algoritmo abstraída por eles.

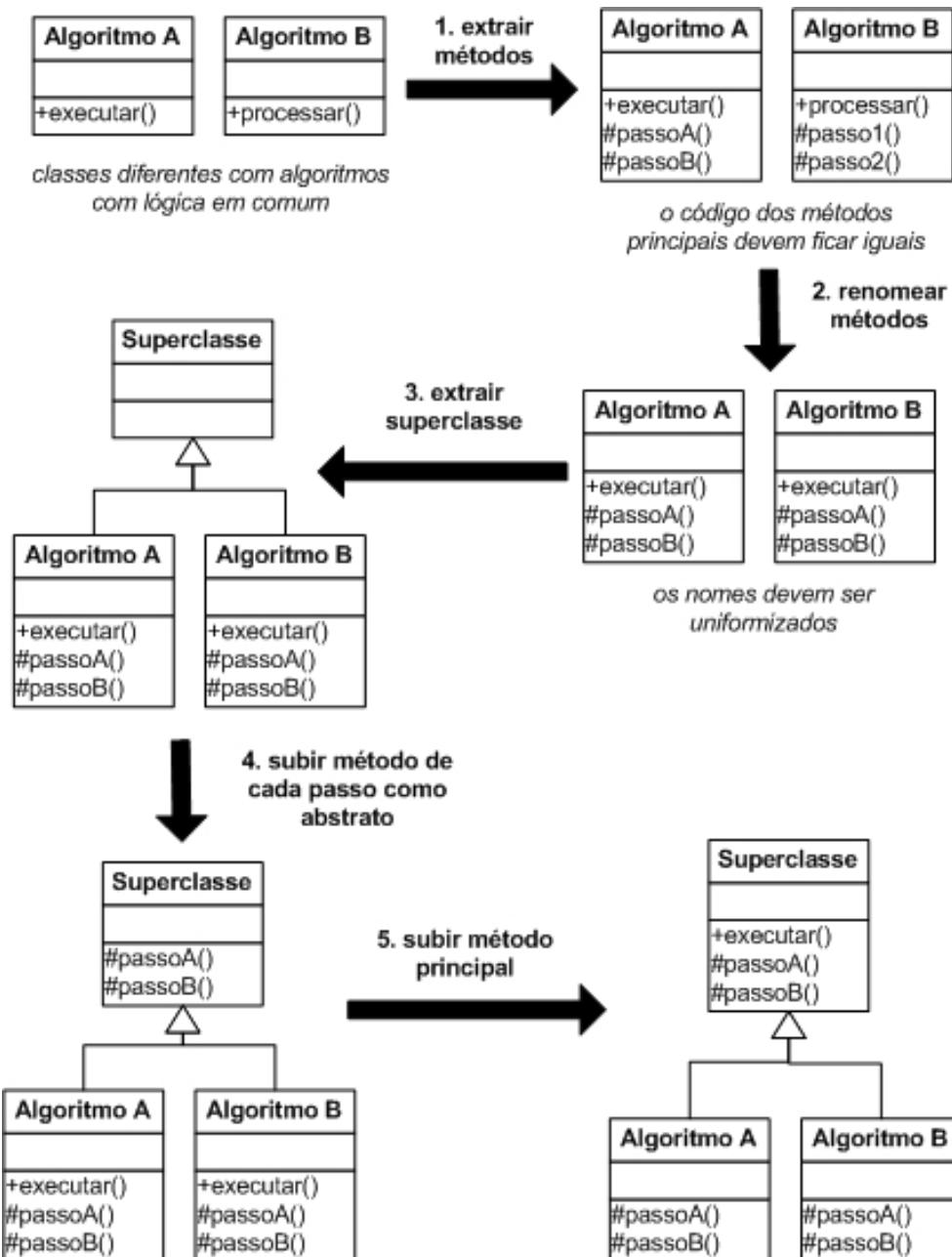


Figura 2.5: Criando superclasse para duas classes com código duplicado

Para seguir em direção a implementação do **Template Method**, o passo seguinte é criar uma superclasse que seja especializada por ambas as classes. A princípio essa classe estará vazia para que não conflite com a implementação de cada uma. Em seguida, deve-se subir para superclasse os métodos extraídos, porém como métodos abstratos. Isso permitirá que na etapa seguinte o método principal seja totalmente movido para superclasse e retirado de todas as subclasses.

## Mau cheiro de código: Missing Template Method

Na terminologia de design de software, um **mau cheiro**, ou *bad smell*, se refere a algum indício de problemas no código que podem significar deficiências em sua modelagem. Um mau cheiro é diferente de um bug, pois não necessariamente ele causa um erro na aplicação, mas trás outras consequências negativas como dificuldade de manutenção e falta de flexibilidade. Existem ferramentas que fazem a detecção automática de maus cheiros, usando como base suas métricas e características. O livro “Refatoração: aperfeiçoando o projeto de código existente” [?] trás uma lista com alguns maus cheiros.

Um dos mau cheiros que são frequentemente detectados por essas ferramentas se chama *“Missing Template Method”*. Esse mau cheiro se refere a dois componentes que possuem um número significante de similaridades, mas não compartilham de uma mesma abstração, como uma interface, ou de uma mesma implementação, como um *Template Method*. Ao se deparar com essa situação, seja detectada por uma ferramenta ou não, é indicada a refatoração descrita.

## 2.6 CRIANDO OBJETOS NA SUBCLASSE - FACTORY METHOD

“Pequenas surpresas após cada esquina, mas nada perigoso.”

– Willy Wonka, A Fantástica Fábrica de Chocolates

O conceito de *hook method* não se aplica somente ao padrão **Template Method**. Existem outros padrões que fazem uso do mesmo princípio para solucionar problemas diferentes. Eles ainda pode combinar isso com outras técnicas para compor soluções mais elaboradas. Essa seção apresenta o padrão **Factory Method**, que é utilizado para resolver um problema relacionado a criação de objetos.

Quando estamos desenvolvendo uma aplicação, é comum termos classes de diferentes hierarquias se relacionando. Por exemplo, uma classe que representa uma entidade do sistema pode se relacionar com a classe que faz a sua validação, ou uma

classe que representa um documento pode se relacionar com a classe que gera sua representação em PDF. Quando isso ocorre, acabamos tendo que criar uma classe em função da outra que está sendo utilizada. No exemplo citado, sabemos que todo documento possui seu gerador de PDF, mas como relacionar as duas classes? Para a classe que estará trabalhando com a lógica de geração de arquivos, não interessa de qual classe é a instância que ele está recuperando, mas apenas que essa instância seja relacionada com a classe que está trabalhando.

## Relacionando DAOs e Classes de Serviço

Para ir mais fundo nesse problema da criação de objetos, vamos considerar um exemplo bastante comum em todo tipo de aplicação. Quando temos uma arquitetura de camadas definida, é comum que objetos de uma camada precisem criar os objetos respectivos da camada seguinte. Nesse exemplo, consideraremos duas camadas da aplicação: a camada DAO (nome que vem do padrão Data Access Object) que realiza o acesso ao banco de dados e a camada de serviço, onde estão implementadas as regras de negócio. Nesse caso, a camada de serviço relacionada com uma entidade deve criar o DAO para a mesma entidade.

A interface DAO define os métodos gerais de acesso a dados que serão disponibilizados, como para recuperação, gravação e exclusão de entidades. Essa interface precisará ser implementada para cada entidade do sistema para a criação dessas operações para cada uma. Note que a interface possui um parâmetro genérico que é utilizado para determinar a entidade associada ao DAO. Esse parâmetro genérico é utilizado para que essa entidade seja utilizada nos parâmetros e retornos dos métodos, variando de acordo com que for setado na implementação.

```
public interface DAO<E> {  
    public E recuperarPorId(Object id);  
    public void salvar(E entidade);  
    public void excluir(Object id);  
    public List<E> listarTodos();  
}
```

A classe ServicoAbstrato representa uma classe da camada de negócios que contém serviços relacionados a uma entidade. Para prover sua funcionalidade, ela precisa da colaboração do DAO relacionado a mesma entidade. Essa classe possui métodos com serviços gerais que são provados a todas as entidades. Um exemplo é o método gravarEntidadeEmArquivo(), que recupera uma entidade a partir de seu

identificador e chama a instância de GeradorArquivo ( aquela classe criada anteriormente nesse capítulo) para criação de um arquivo com as propriedades da classe. Essas propriedades são obtidas através do método `getMapa()` definido para o tipo Entidade. Vale ressaltar que qualquer subclasse de GeradorArquivo pode ser utilizada, pois ela é passada no construtor da classe.

A grande questão é que os métodos gerais dessa classe abstrata precisam acessar o DAO para executar sua funcionalidade, porém qual será a instância é algo que só será definido na subclasse, onde a entidade com a qual se trabalha já terá sido definida. Dessa forma, a classe definiu um método abstrato chamado `getDAO()`, que retorna a instância do DAO para ser utilizada. Dessa forma, as subclasses devem implementar esse método de forma a retornar a instância correta. Esse é um *hook method* com o objetivo de criar um objeto.

```
public abstract class ServicoAbstrato<E>{
    public GeradorArquivo gerador;

    public ServicoAbstrato(GeradorArquivo gerador){
        this.gerador = gerador;
    }

    public abstract DAO<E> getDAO();

    //Serviço geral
    public void gravarEntidadeEmArquivo(Object id, String nomeArquivo){
        E entidade = getDAO().recuperarPorId(id);
        gerador.gerarArquivo(nomeArquivo,
            ((Entidade)entidade).getMapa());
    }
}
```

A classe `ServicoProduto`, apresentada na próxima listagem, representa um exemplo de uma subclasse da classe `ServicoAbstrato` para a entidade `Produto`. Observe que ela implementa o *hook method* definido e cria a instância do DAO relacionado com a classe `Produto`. No exemplo, a instância é criada quando o primeiro acesso é feito ao método. Essa classe pode possuir métodos específicos de produto, porém os métodos herdados da superclasse que utilizam o método `getDAO()` também poderão ser utilizados.

```
public class ServicoProduto extends ServicoAbstrato<Produto>{
    private DAO<Produto> dao;
```

```
public DAO<Produto> getDAO(){  
    if(dao == null){  
        dao = new ProdutoDAO();  
    }  
    return dao;  
}  
  
//Funcionalidade específica  
public double getPrecoProduto(Object produtoId){  
    Produto p = getDAO().recuperarPorId(id);  
    return p.getPreco();  
}  
}
```

Vale observar a utilização dos tipos genéricos nesse exemplo. Como a classe `ServicoProduto` especificou de forma explícita o tipo genérico de sua superclasse, todos os elementos da classe que usavam o parâmetro do tipo genérico serão fixados. Como o método `getDAO()` precisará agora retornar um `DAO<Produto>`, haverá um erro de compilação caso um `DAO` com outro tipo genérico for retornado.

## Entendendo o Factory Method

O exemplo de criação do `DAO` nas classes de serviço, ilustram bem o cenário onde a classe da instância utilizada depende da subclasse. O padrão `Factory Method` utiliza um *hook method* para delegar a criação da instância para a subclasse. Isso permite que métodos mais gerais na superclasse possam utilizar essa instância mesmo sem conhecer a classe concreta. Isso pode ser feito invocando o método abstrato de criação que é implementado na subclasse.

A Figura 2.6 apresenta um diagrama com a estrutura do padrão `Factory Method`. Nesse padrão, a classe principal possui uma dependência com uma abstração de uma hierarquia de classes. No diagrama, `Dependencia` está representada como uma interface, porém não há nada que impede de ser uma classe. O método `metodoFabrica()` possui a responsabilidade de criar uma instância de `Dependencia`, porém na classe principal ele é um método abstrato. Dessa forma, nas subclasses esse método precisa ser implementado e deve ser criado um objeto de uma classe concreta que possua a abstração da dependência.

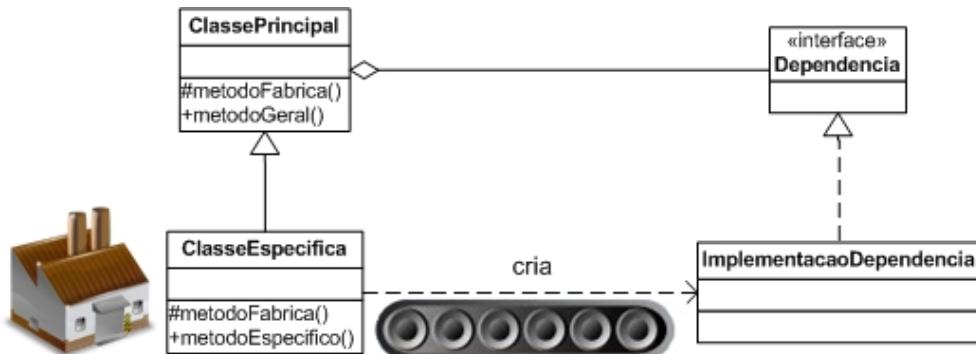


Figura 2.6: Estrutura do padrão Factory Method

A partir desse padrão, é possível desacoplar a superclasse da criação de uma dependência. Com a criação das instâncias na subclasse, apenas elas ficam acopladas as classes concretas da hierarquia. Dessa forma, caso uma nova instância da dependência precise ser utilizada pela superclasse, basta criar uma nova subclasse que retorne aquela instância.

## 2.7 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Esse capítulo abordou a utilização da herança para a reutilização de código. O primeiro padrão apresentado foi `Null Object`, que mostra como o polimorfismo pode ser utilizado para que o código dos clientes de uma classe possa ser adaptados com suas extensões. Nesse padrão foi visto como a extensão de uma classe pode ser sua representação nula, fazendo com que o código cliente consiga lidar com ela de forma transparente.

Em seguida foi mostrado como o uso da herança pode ser feito para permitir a especialização do código da superclasse. A partir de *hook methods* é possível que as subclasses insiram comportamento em métodos que estão implementados na superclasse. Foram apresentados os padrões `Template Method` e `Factory Method` que fazem uso desse princípio.

Apesar desses padrões possuírem o uso da herança como parte principal de sua solução, o uso dos princípios apresentados não são sua exclusividades. Em qualquer outro contexto em que os requisitos puderem ser solucionados a partir da especialização da funcionalidade da superclasse, os mesmo princípios apresentados podem

ser utilizados. O próximo capítulo irá explorar o uso da composição e mostrar como ela pode ser utilizada em conjunto com a herança e quando seu uso é mais adequado.



## CAPÍTULO 3

# Delegando Comportamento com Composição

*“A primeira regra do gerenciamento é a delegação. Não tente fazer tudo sozinho porque você não consegue.”*

– Anthea Turner

Ao pensar em objetos, uma coisa muito natural é imaginar em como são combinados. Na verdade, não existem grandes construções do homem que não são feitas a partir da combinação de diversos objetos. Uma casa, por exemplo, é feita a partir da combinação de diversos blocos. Olhando outros exemplos, um carro é composto de peças e um computador é feito de componentes eletrônicos. O que é mais interessante é que a medida que vamos descendo, muitas vezes o padrão vai se repetindo: uma peça pode ser composta por outras, e uma placa de computador é feita com capacitores, resistências e chips. É a quantidade infinita de combinações desses elementos que torna possível a construção de diversas coisas diferentes a partir do mesmo material.

Quando a programação orientada a objetos foi criada, um dos objetivos era simplificar a abstração dos conceitos do mundo real para o software. Dessa forma, um elemento existente no domínio da aplicação, poderia ser representado utilizando um objeto no software. Da mesma forma que as coisas interagem no mundo real para a realização de um objetivo, dentro de um software não poderia ser diferente. Do mesmo jeito que diversas peças são combinadas em vários níveis para compor um componente físico, os componentes de software também podem ser resultado da combinação de outros componentes mais granulares.

Apesar de não estar entre os quatro elementos principais de uma linguagem orientada a objetos, a composição é um conceito que vem implícito quando se fala em objetos. O Capítulo 1 já apresentou o uso da composição através do padrão **Strategy**, e este capítulo irá explorar o conceito mais a fundo e mostrar como ele pode ser utilizado para a extensão de comportamento. Será mostrado como a composição combinada com o uso de abstrações pode ser eficaz na solução de diferentes problemas. A primeira seção começa revisitando o exemplo do gerador de arquivos do capítulo anterior para ilustrar cenários onde as limitações da herança são evidentes.

## 3.1 TENTANDO COMBINAR OPÇÕES DO GERADOR DE ARQUIVOS

No exemplo do capítulo anterior, a classe `GeradorArquivo` implementava o padrão `Template Method` para permitir que as subclasses pudessem especializar parte do comportamento, como o formato da geração do conteúdo e um pós-processamento a ser feito nos bytes do arquivo. O método principal `gerarArquivo()` invocava os *hook methods* `processar()` e `gerarConteudo()`, que eram implementados somente na subclasse. As duas subclasses de `GeradorArquivo`, `GeradorXMLCompactado` e `GeradorPropriedadesCriptografado`, respectivamente implementam os métodos para gerar em XML compactados e de propriedade criptografados.

O problema começa ao tentarmos ter uma implementação que combina o comportamento dessas subclasses, como para a geração de arquivos XML e criptografados. Se tentarmos utilizar a herança, acabamos com a duplicação de alguma parte do código. A Figura 3.1 mostra o que acontece quando criamos uma nova classe que define o gerador de XML como uma superclasse com duas subclasses que especializam apenas a parte do pós-processamento. Apesar de ter sido criada uma abstração referente ao formato do arquivo, a ausência de uma abstração para o pós-processamento faz com que haja duplicação de código.

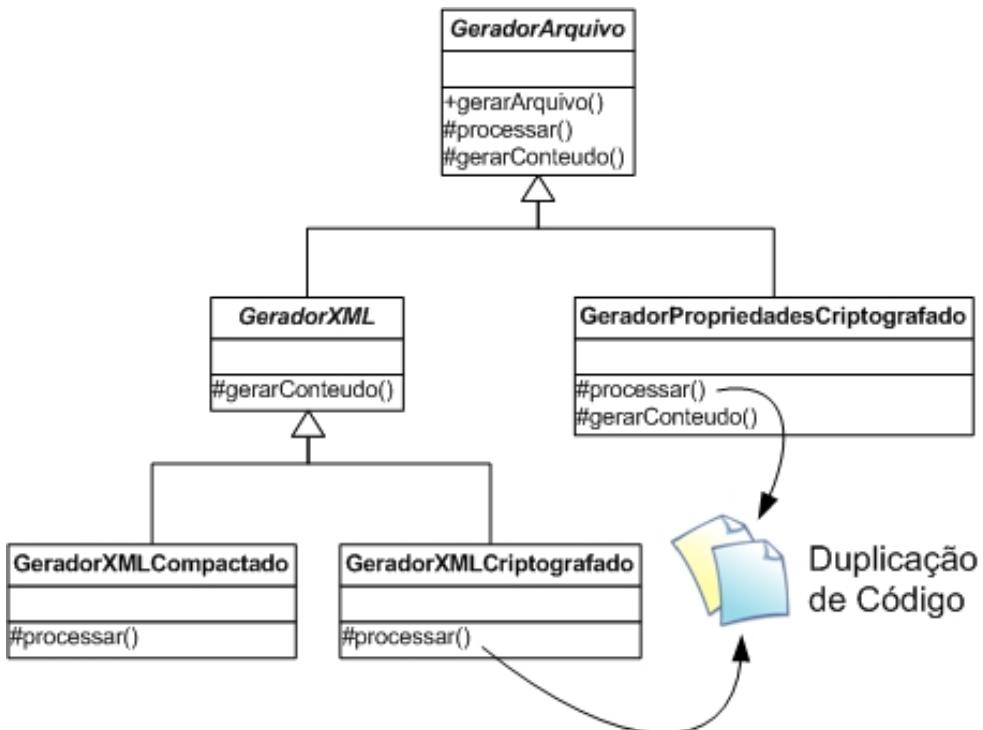


Figura 3.1: Duplicação de código ao se tentar criar um novo nível na hierarquia

A Figura 3.2 mostra que se tentarmos estruturar a hierarquia de uma outra forma para eliminar a duplicação anterior, continuaremos com duplicação em uma outra parte do código.

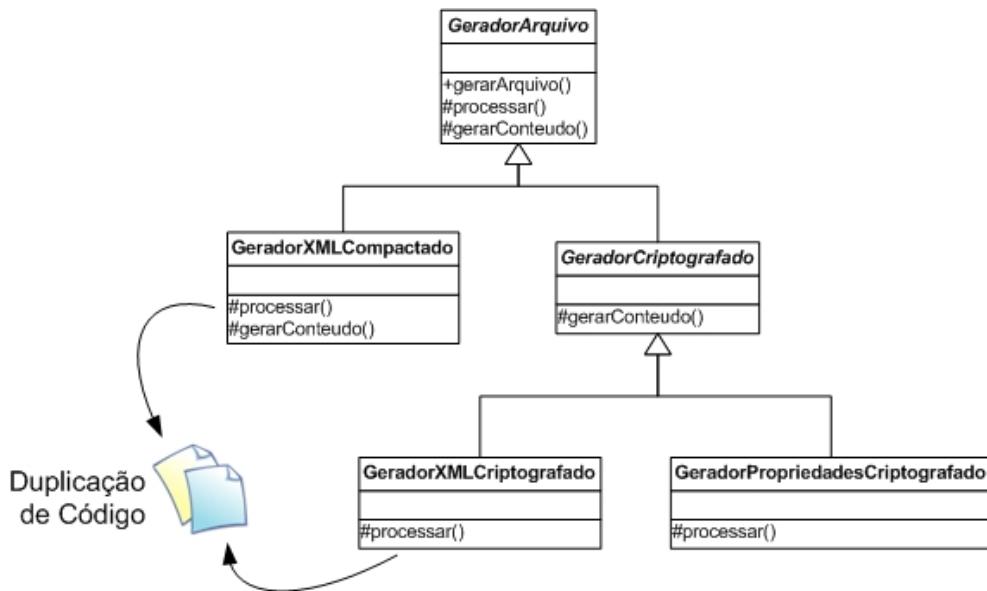


Figura 3.2: Manutenção da duplicação ao se tentar criar a hierarquia pelo outro lado

A limitação do uso da herança, que fica evidente nesse problema, é o fato dela poder ser utilizada apenas uma vez. Se fosse possível ter herança múltipla em Java, uma classe poderia obter a implementação do formato do arquivo de uma superclasse e a do pós-processamento de outra. Como isso não é possível, quando precisamos de uma implementação de outro ramo da hierarquia, acaba havendo uma duplicação de código.

Quando mais de um *hook method* são definidos em uma superclasse e implementados por uma subclasse, essas implementações ficam ligadas uma a outra devido ao fato de terem sido implementadas na mesma classe. Por mais que seja possível ir implementando cada *hook method* em um nível diferente da hierarquia, a duplicação ainda pode acontecer, principalmente se esses fatores que pode variar, chamados de **variabilidade**, são independentes um do outro. O problema vai se tornando ainda mais crítico quando se aumenta o número de variabilidades e o número de possíveis implementações de cada uma delas. A quantidade de classes necessárias para implementar todas as possibilidades tende a aumentar de forma exponencial!

## 3.2 BRIDGE - UMA PONTE ENTRE DUAS VARIABILIDADES

*“A ponte não é de concreto, não é de ferro / Não é de cimento / A ponte é até onde vai o meu pensamento / A ponte não é para ir nem pra voltar / A ponte é somente pra atravessar / Caminhar sobre as águas desse momento”*

– Lenini

Como todos já devem desconfiar pelo foco do capítulo, a composição será utilizada como forma de resolver o problema descrito. Como com a pura utilização de herança a implementação de uma variabilidade fica ligada a outra, a ideia da nova solução seria separar a implementação de uma delas em uma outra classe. Da mesma forma mostrada no padrão Strategy no Capítulo 1, essa classe irá compor a classe principal, no caso GeradorArquivo. Dessa forma, o método implementado nessa hierarquia de classes a parte será invocado como parte da implementação do algoritmo. A Figura 3.3 mostra como ficaria a estrutura de classes com a utilização da composição para os algoritmos de pós-processamento dos arquivos.

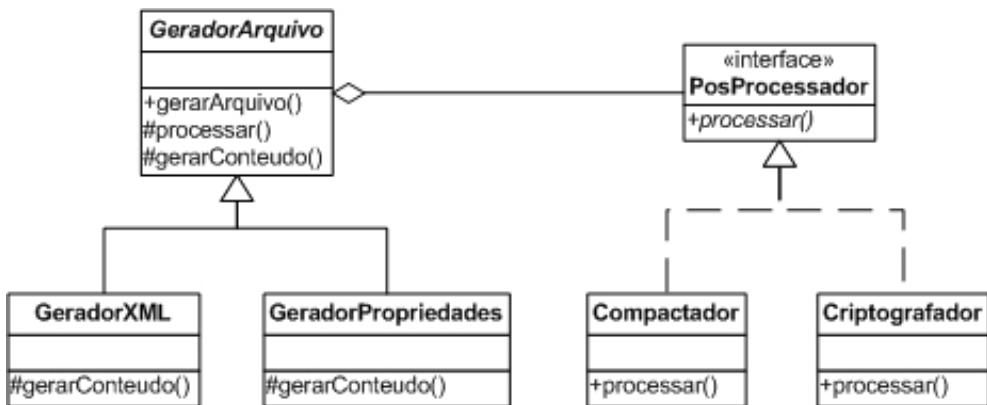


Figura 3.3: Utilização de Composição no GeradorArquivo

A listagem a seguir apresenta o código do GeradorArquivo após a aplicação dessa solução. O método `processar()` não está mais na mesma classe e sim na instância de `PosProcessador` que compõe `GeradorArquivo`. Dessa forma, qualquer implementação dessa interface pode ser utilizada para realizar o pós-processamento do arquivo. A grande motivação dessa solução no cenário apresentado é que o processador pode ser configurado independente da subclasse que está sendo utilizada, permitindo as duas variabilidades variar de forma independente.

```
public abstract class GeradorArquivo {  
    private PosProcessador processador;  
  
    public void setProcessador(PosProcessador processador){  
        this.processador = processador;  
    }  
  
    public final void gerarArquivo(String nome,  
                                   Map<String, Object> propriedades)  
        throws IOException {  
        String conteudo = gerarConteudo(propriedades);  
        byte[] bytes = conteudo.getBytes();  
        bytes = processador.processar(bytes);  
        FileOutputStream fileout = new FileOutputStream(nome);  
        fileout.write(bytes);  
        fileout.close();  
    }  
  
    protected abstract String  
        gerarConteudo(Map<String, Object> propriedades);  
}
```

### MAS E O CASO EM QUE NÃO HÁ PÓS-PROCESSAMENTO?

Na implementação original apresentada no capítulo anterior, o método `processar()` possuía uma implementação *default* para o caso de não haver nenhum pós-processamento. Esse caso deve ser tratado aqui nessa solução também! O que deve ser feito? Deve-se colocar um condicional para verificar se o processador foi configurado corretamente? Essa é uma solução possível, porém podemos utilizar o `Null Object` para solucionar esse problema.

Para fazer isso, uma implementação de `PosProcessador` chamada `NullProcessador` poderia ser criada com uma implementação que retornasse o próprio array de bytes recebido como parâmetro sem nenhum pós-processamento. Sendo assim, para garantir que não ocorra um `NullPointerException` caso o conteúdo da variável `processador` seja nulo, uma instância dessa classe poderia ser atribuída a ela em algum momento da inicialização do objeto.

Esse exemplo mostra como é importante combinar as soluções dos padrões para chegar a uma estrutura final. Enxergue um padrão como um elemento que pode ser incluído para compor a sua solução e não a solução completa!

## Entendendo o padrão Bridge

Essa solução caracteriza o padrão `Bridge`, que cria uma ponte entre duas hierarquias ligadas por uma relação de composição permitindo que ambas variem de forma independente. Nesse caso, a ponte é caracterizada pela relação de composição entre a classe `GeradorArquivo` e a interface `PosProcessador`. Um cenário onde esse tipo de solução é comum é quando temos uma hierarquia de abstrações e outra com implementações, permitindo que cada uma possa variar independentemente. Neste caso, o `GeradorArquivo` e suas subclasses são uma abstração que caracteriza algoritmos de geração de arquivos em determinados formatos, porém ainda sem definir a implementação do pós-processamento.

A Figura 3.4 apresenta uma estrutura mais geral para o padrão `Bridge`. A classe `Abstracao` representa um conceito que precisa ser representado em um sis-

tema e possui duas variabilidades independentes, ou seja, comportamentos não dependentes que podem ser estendidos e/ou modificados. A classe representada como `AbstracaoRefinada` especializa o comportamento da abstração através da implementação de seus *hook methods*. Já a abstração representada pela interface `Componente` compõe a classe `Abstracao` e representa uma de suas variabilidades. Dessa forma, quando essa operação precisar ser invocada em `Abstracao`, o método na classe que a compõe será chamado.

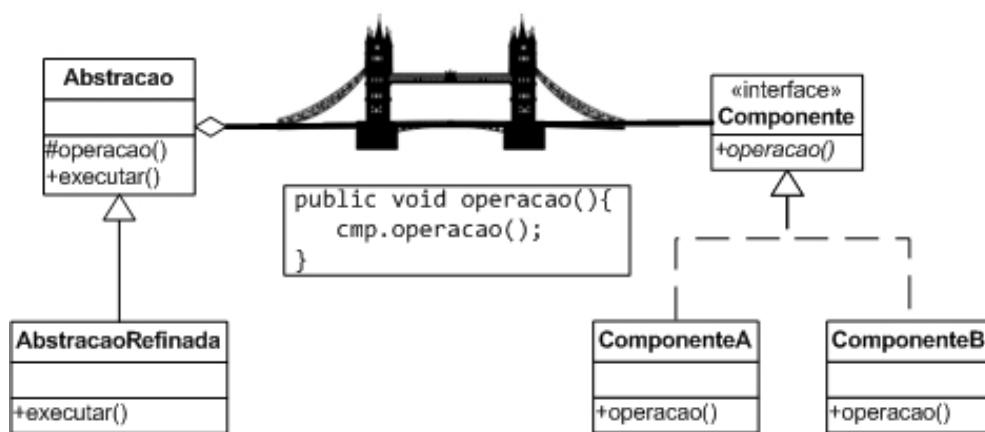


Figura 3.4: Estrutura Geral do Padrão Bridge

No exemplo apresentado, a classe `Abstracao` é representada pela classe `GeradorArquivo`, que representa uma abstração de como gerar arquivos a partir de mapas de propriedades. Seus refinamentos são classes que a especializam para que a geração dos arquivos seja feita em um formato específico, como XML e arquivos de propriedades. A interface `Componente` é representada pela interface `PosProcessador` e fica claro a partir de suas implementações que ela representa uma operação de pós-processamento do arquivo.

## Os Efeitos do Bridge

A principal motivação na implementação do padrão Bridge é tornar independentes os fatores que podem variar na solução. Qualquer subclasse de `Abstracao` pode ser facilmente composta com qualquer implementação da interface `Componente`, permitindo a combinação de ambas livremente sem duplicação de

código. Caso, por exemplo, uma nova implementação de qualquer uma das duas hierarquias for criada, ela será facilmente combinada com o que já existe.

Um efeito colateral muito interessante é que as classes que foram separadas, também podem ser utilizadas em outro contexto. No exemplo, os algoritmos de pós-processamento poderia ser utilizados na geração de outros tipos de arquivos ou mesmo para o envio de informações pela rede. Esse tipo de questão nos faz refletir se a princípio, pensado apenas nas responsabilidades, se essa funcionalidade já não deveria ter sido atribuída a outra classe. Imagine acessar uma classe chamada `GeradorArquivo` para criptografar dados para serem enviados em uma rede. Dessa forma, uma consequência desse padrão também é o desacoplamento de responsabilidades, permitindo mais facilmente o seu reuso em outros contextos.

Um ponto que acho interessante no `Bridge` é o fato de sua solução utilizar ao mesmo tempo herança e composição. Escuto muitas pessoas dizerem para utilizar sempre a composição no lugar da herança, porém acho que a palavra "**sempre**" é muito complicada no contexto de design de software, pois não existe bala de prata nem uma solução mágica que irá resolver todos os seus problemas. A composição tem sim suas vantagens que serão apresentadas no presente capítulo, porém é importante sempre avaliar com cuidado o problema e os requisitos da solução para que se busque a solução mais adequada.

### O PADRÃO NÃO É O DIAGRAMA!

Um erro comum ao se aprender os primeiros padrões é achar que a implementação deve ser sempre exatamente como é mostrada no diagrama geral do padrão. O diagrama é apenas uma referência, pois a mesma solução pode ser implementada de diversas outras formas. Por exemplo, dependendo do contexto, `Implementacao` poderia ser implementada como uma classe abstrata ou a classe `Abstracao` poderia possuir mais de um *hook method* a ser implementado por suas subclasses. É por esse motivo, que ferramentas que geram código ou provêem uma estrutura de classes para a implementação de padrões são bastante criticadas por especialistas da área. É mais importante se preocupar em ter um solução adequada ao seu problema, do que seguir cegamente a estrutura do padrão.

### 3.3 HOOK CLASSES

No capítulo anterior foi introduzido o conceito de *Hook Methods*, que são métodos que podem ser sobreescritos pelas subclasses como forma de estender e especializar o comportamento da classe. Nesse capítulo estamos vendo uma outra forma de alterar o comportamento das classes a partir da composição. Ao invés dos pontos em que o comportamento pode variar serem definidos na mesma classe, eles são definidos em uma outra classe que compõe a classe principal. A essa classe que pode ser alterada damos o nome de *Hook Class*. A Figura 3.5 ilustra o conceito apresentado.

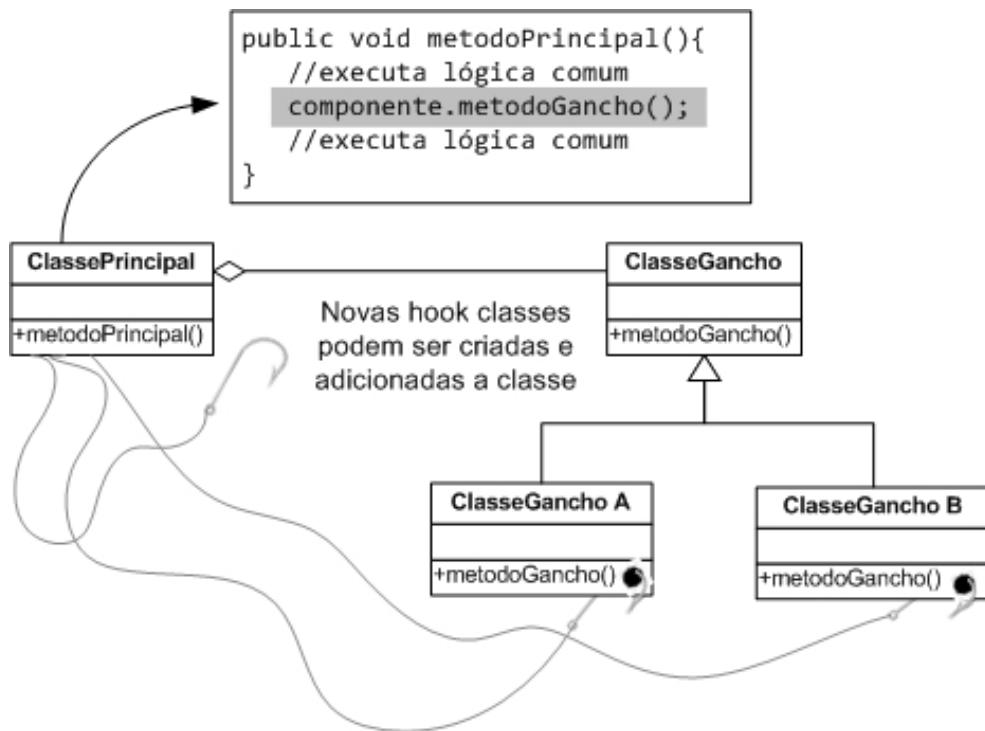


Figura 3.5: Representação do Conceito de Hook Classes

Da mesma forma que os *hook methods*, as *hook classes* são uma técnica utilizada pelos padrões para chegar a uma solução para um problema mais específico. O padrão Bridge, por exemplo, utiliza ao mesmo tempo um *hook method* e uma *hook class* como forma de separar dois pontos de extensão cujo comportamento pode variar de forma independente. Dessa forma, é importante não apenas conhecer os

padrões, mas entender os princípios por trás de sua estrutura para entender melhor as consequências trazidas por uma decisão de design.

## Diferenças Entre os Tipos de Hooks

Quando entendemos a ideia por trás dos *hook methods* e das *hook classes*, eles até que são bem parecidos. Em ambos os casos, a classe principal chama um método cuja implementação pode variar. No caso dos *hook methods* esse método está na mesma classe, podendo a implementação variar com a subclasse, e no caso das *hook classes* o método está em um objeto que compõe a classe, fazendo com que a implementação varie com a instância. Apesar de alguma semelhança, a utilização de uma técnica ou outra possui consequências bem diferentes.

A primeira delas está no momento em que a implementação que será utilizada é escolhida. Quando um *hook method* é utilizado, a escolha é feita no momento em que o objeto é instanciado. Isso ocorre pois ao criarmos um novo objeto devemos escolher qual a classe concreta que será utilizada, e fazendo isso estamos escolhendo a sua implementação dos *hook methods*. Já no caso da utilização de *hook classes*, a implementação pode ser trocada a qualquer momento, bastando para isso trocar a instância que foi configurada. Observe que isso não é obrigatório, como no exemplo da classe `GeradorArquivo` apresentada nesse capítulo, que permitia a configuração da classe que a compunha apenas no construtor.

Nesse ponto, vale a pena retomar o exemplo do primeiro capítulo, onde era necessário poder trocar a lógica que calculava o valor do estacionamento de acordo com o tempo e com o tipo de veículo. No cenário apresentado era extremamente importante poder trocar a implementação do cálculo após a criação do objeto `ContaEstacionamento`. Esse foi um dos motivos que fizeram a escolha do padrão `Strategy`, que utiliza uma *hook class*, ser adequado para a resolução do problema.

Outra questão que deve ser levada em consideração é o grau de dependência de diferentes pontos de extensão. Caso os *hook methods* sejam definidos na mesma classe, sendo eles na própria classe ou em uma *hook class*, as implementações ficam ligadas. Foi esse o problema apresentado no exemplo do `GeradorArquivo`, onde pontos de extensão eram independentes mas ficavam definidos na mesma classe. Quando usamos apenas *hook methods*, pelo fato de não haver herança múltipla em Java, eles sempre ficam interligados. Ao utilizarmos *hook classes*, se colocarmos os métodos na mesma classe o mesmo problema irá acontecer, porém se colocarmos em classes diferentes, cada implementação poderá ser configurada de forma independente. O padrão `Bridge` aborda justamente essa questão, utilizando diferentes

estratégias de extensão para que as implementações possam variar de forma independente.

## Evoluindo o Modelo de Classes

Em um artigo clássico sobre padrões para evolução de frameworks [?], é mostrado que normalmente os pontos de extensão são primeiramente definidos utilizando herança, o que é conhecido como *whitebox framework*, para só em seguida evoluir para o uso de composição, chamado de *blackbox framework*. Por mais que a composição pareça ser mais vantajosa que a herança por suas características estruturais, é muito mais fácil deixar sua classe aberta a extensão através de herança.

O termo *whitebox framework* se refere a frameworks cujas classes que permitem sua extensão a partir da herança. Devido ao fato das subclasses terem acesso a detalhes internos da classe que está sendo estendida, sua estrutura fica exposta. O termo *whitebox*, ou caixa branca, é uma metáfora a essa visibilidade que o desenvolvedor precisa ter de como as coisas são na classe internamente. Com a utilização desse paradigma, qualquer método público ou protegido que não seja final, é um potencial *hook method* a ser especializado pelas subclasses. Dessa forma, quando não se sabe exatamente o que será necessário ser estendido, a herança é uma boa alternativa por possibilitar que vários pontos fiquem em aberto. Uma desvantagem dessa abordagem, é que quem desenvolver a classe que faz a extensão deve conhecer a estrutura interna da superclasse, tornando a tarefa mais complexa.

Por outro lado, o termo *blackbox framework* se refere a frameworks cujo mecanismo de extensão que se baseia em composição. Nesse caso, o desenvolvedor deve criar implementações que obedecem a uma interface fornecida e utilizá-las para compor a classe principal. O termo *blackbox*, ou caixa preta, é utilizado porque o desenvolvedor não precisa conhecer a parte interna das classes do framework para estendê-las. Apesar de possuir as vantagens da composição, esse tipo de ponto de extensão é mais difícil de ser identificado, pois é preciso saber a porção exata da funcionalidade que será delegada. Em compensação, depois que esses pontos forem identificados, é mais fácil para outros desenvolvedores de alterarem o comportamento, pois eles só precisam entender as interfaces que estão implementando.

Na verdade, nada precisa ser completamente preto ou branco. O caminho natural é, ao criar a primeira classe, deixar alguns possíveis *hook methods* em aberto para que seja possível realizar a extensão do comportamento. A medida que as extensões forem sendo criadas e os principais pontos onde o comportamento é especializado com frequência forem identificados, a estrutura pode ser refatorada para o uso de

*hook classes.*

### 3.4 STATE - VARIANDO O COMPORTAMENTO COM O ESTADO DA CLASSE

*“Eu costumava partir por semanas em um estado de confusão.”*

– Albert Einstein

Um cenário bastante comum ao desenvolvermos um software é uma determinada entidade mudar de estado durante o decorrer de sua execução, alterando seu comportamento de acordo com esse estado. Uma conta corrente em um banco, por exemplo, comporta-se diferente caso o saldo esteja negativo. Em um jogo, as mesmas ações podem gerar diferentes efeitos em um personagem quando ele está em estados diferentes, como quando pega algum item ou está dentro d'água.

A implementação comum nesses casos é criar uma série de condicionais onde, em diversos pontos da classe, verifica-se o estado do objeto e executa-se a lógica mais adequada. Seguindo essa ideia, alguns estados são suficientes para deixar o código bastante confuso. É comum que a mesma sequência de condicionais seja encontrada em diversos pontos do código, onde o estado do objeto possui influência. Além disso, essa estrutura torna complicada a adição de mais estados, pois isso demandará alterações no código da própria classe e irá aumentar ainda mais a quantidade de condicionais. Como resolver isso? Conheça o padrão State!

#### A Estrutura do State

Essa seção irá apresentar o padrão State, o qual utiliza composição para permitir essa variação de comportamento de acordo com o estado de uma entidade do sistema. Nesse padrão, os estados são representados a partir de classes que obedecem a uma abstração comum, podendo ser uma interface ou uma superclasse. Dessa forma, a entidade é composta por um objeto do tipo dessa abstração e, nos métodos de negócio, delega para ele o comportamento de toda parte que depende do seu estado. A Figura 3.6 representa a estrutura desse padrão.

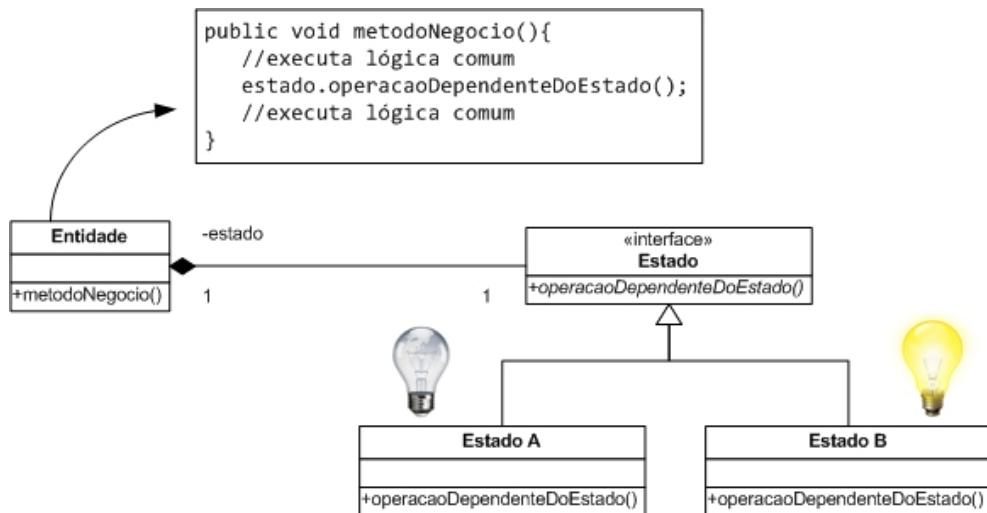


Figura 3.6: Estrutura do padrão State

Sendo assim, apenas a lógica comum será mantida na entidade e o comportamento do objeto específico de cada estado estará definido em cada subclasse. Quando o estado for alterado, basta trocar a instância utilizada para caracterizar o estado, que consequentemente o comportamento da entidade será alterado. Nesse caso, o estado é uma *hook class* para a qual está sendo delegado comportamento. Uma consequência desse padrão é a simplicidade em alterar estados existentes, ou mesmo inserir novos estados no ciclo de vida da entidade. A desvantagem é que, devido a divisão da lógica dos estados, fica mais difícil ter uma visão global dos estados e transições que podem ser realizadas.

## Busca em Profundidade

Calma! Você não está em um livro de algoritmos! Acho muito interessante quando é possível resolver um problema clássico na área de projeto de algoritmos e estrutura de dados utilizando algum padrão. Isso nos mostra que eles são úteis para qualquer tipo de software. O algoritmo abordado nessa seção é a busca em profundidade em grafos. Nesse algoritmo, a medida que os nós do grafo vão sendo percorridos, eles mudam de cor e o comportamento do algoritmo depende da cor do nó que está sendo percorrido. Dessa forma, o State será utilizado para configurar o comportamento de cada nó de acordo com seu estado.

A busca em profundidade é um algoritmo de busca de grafos, no qual o inicia-se com um nó raiz e vai se explorando cada um de seus ramos, sempre na maior profundidade possível, antes de retroceder e processar os outros ramos. Para quem não se lembra do algoritmo de busca em largura em grafos, segue uma breve explicação de como ele funciona. A execução inicia-se a partir de um nó e todos iniciam-se com a cor branca. Quando um nó começa a ser processado, ele adquire a cor cinza que significa que ele está sendo processado. Ao adquirir a cor cinza, são processados todos os seus nós adjacentes que tenham a cor branca. Ao finalizar a execução o nó passa da cor cinza para cor preta, que indica que seu processamento foi concluído. O algoritmo termina quando o nó raiz onde o algoritmo iniciou adquire a cor preta.

A Figura 3.7 apresenta um exemplo de execução do algoritmo passo a passo em um grafo. Nesse exemplo, a medida que os nós vão mudando de estado, eles adquirem uma cor diferente e as arestas que já tiverem sido percorridas são colocadas em negrito. A partir do exemplo é possível entender de uma forma mais concreta como os nós são percorridos e a situação em que mudam de estado.

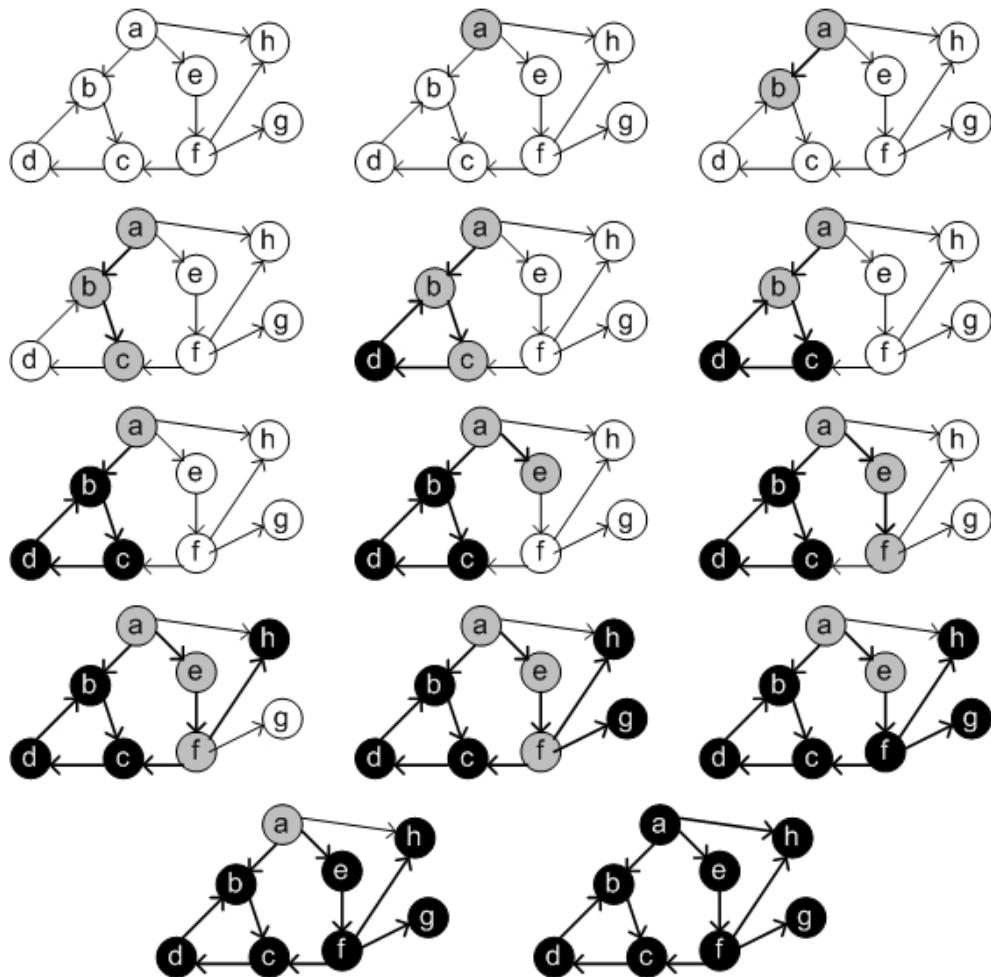


Figura 3.7: Busca em Profundidade Passo a Passo

### Implementação da Busca em Profundidade Utilizando State

O primeiro passo da implementação é definir um nó e quais os pontos da execução que diferem com o estado. No contexto do algoritmo, as informações relevantes de um nó é o seu nome, seus nós adjacentes e sua cor. Essa classe possui um método chamado `buscaProfundidade()` que irá efetivamente executar o algoritmo. Os pontos identificados que podem variar com o estado são: (a) a própria execução da busca em profundidade, pois no caso do nó ser preto ou cinza ela não deve ser executada;

e (b) quando o nó assume uma cor. A implementação realizada para classe No está representada na listagem a seguir.

```
public class No {  
    private Set<No> adjacentes = new HashSet<>();  
    private Cor cor;  
    private String name;  
  
    public No(String name){  
        this.name = name;  
        cor = new Branco();  
    }  
    public void buscaProfundidade(List<No> list){  
        cor.busca(this, list);  
    }  
    public Set<No> getAdjacentes() {  
        return adjacentes;  
    }  
    public void addAdjacentes(No adj) {  
        adjacentes.add(adj);  
    }  
    public void setCor(Cor cor, List<No> list) {  
        this.cor = cor;  
        cor.assumiu(this ,list);  
    }  
    public String toString() {  
        return name;  
    }  
}
```

O atributo `cor` representa o estado do nó na execução do algoritmo. Observe que métodos são invocados no objeto desse atributo quando o método `buscaProfundidade()` é invocado e quando uma nova cor é configurada a partir do método `setCor()`. O método `buscaProfundidade()` recebe uma lista que deve ser populada com os nós do grafo na ordem que terminaram de ser executados pelo algoritmo.

A listagem a seguir apresenta a classe abstrata `Cor` e suas três subclasses. Observe que cada cor implementa os *hook methods* da definição do estado, de acordo com a descrição do algoritmo. Quando o nó no estado Branco recebe a chamada da busca, o mesmo deve passar para a cor Cinza. Essa, por sua vez, ao ser assumida pelo nó

inicia o processamento de todos os nós adjacentes, assumindo a cor Preto ao seu final. Finalmente, ao se tornar Preto, o nó deve ser adicionado na lista recebida como parâmetro pelo algoritmo.

```
//Classe que abstrai o estado de um nó
public abstract class Cor {
    void busca(No no, List<No> list){}
    void assumiu(No no, List<No> list){}
}

//Implementações dos três estados possíveis
public class Branco extends Cor {
    public void busca(No no, List<No> list) {
        no.setCor(new Cinza(), list);
    }
}

public class Cinza extends Cor {
    void assumiu(No no, List<No> list) {
        for(No adj : no.getAdjacentes())
            adj.buscaProfundidade(list);
        no.setCor(new Preto(), list);
    }
}

public class Preto extends Cor {
    void assumiu(No no, List<No> list) {
        list.add(no);
    }
}
```

A listagem a seguir mostra a criação do grafo apresentado na Figura 3.7 e a execução do algoritmo. O seu final, os nós são impressos na ordem que seu processamento foi finalizado.

```
public static void main(String[] args) {
    No a = new No("A");      No b = new No("B");
    No c = new No("C");      No d = new No("D");
    No e = new No("E");      No f = new No("F");
    No g = new No("G");      No h = new No("H");

    a.addAdjacentes(b);     b.addAdjacentes(c);
    c.addAdjacentes(d);     d.addAdjacentes(b);
```

```
a.addAdjacentes(e);      e.addAdjacentes(f);
f.addAdjacentes(c);      f.addAdjacentes(g);
f.addAdjacentes(h);      a.addAdjacentes(h);

List<No> l = new ArrayList<>();
a.buscaProfundidade(l);
for(No n : l)
    System.out.println(n);
}
```

Nesse algoritmo, diversas ações dependem do estado do nó, como se ele deve ser adicionado na lista ou se os nós adjacentes devem ser processados. Apesar disso, observe que nenhum comando condicional foi utilizado na implementação!

## Utilizando um Enum para Implementar o State

Quando temos que representar estados de um objeto em um software, frequentemente utilizamos enumerações. O problema é que muitas vezes as enumerações possuem apenas a definição dos estados, ficando a lógica específica de cada estado presa dentro dos condicionais na classe. Construções do tipo `enum` podem definir métodos, inclusive abstratos, que podem ser sobreescritos na definição de cada estado. Segue na listagem a seguir, como as cores dos nós do grafo poderiam ser definidas utilizando uma enumeração. Em relação ao resto do código, somente a inicialização do atributo `cor` da classe `No` precisaria ser modificada.

```
public enum Cor {

    BRANCO{
        public void busca(No no, List<No> list) {
            no.setCor(CINZA, list);
        }
    }, CINZA{
        void assumiu(No no, List<No> list) {
            for(No adj : no.getAdjacentes())
                adj.buscaProfundidade(list);
            no.setCor(PRETO, list);
        }
    },
    PRETO{
        void assumiu(No no, List<No> list) {
            list.add(no);
        }
    }
}
```

```
    }
};

void busca(No no, List<No> list){}
void assumiu(No no, List<No> list){}
}
```

Porém existem algumas situações onde a utilização de enumerações é desaconselhada na representação de um estado. A primeira situação é quando é desejável que o estado seja um ponto de extensão e que novos estados possam ser definidos. Nesse caso, como os possíveis estados são definidos dentro de uma enumeração fixa, não é possível adicionar um novo sem a modificação do código do próprio enum. Outra situação é quando algum estado precise armazenar uma informação específica do objeto que está sendo composto por ele. Nesse caso, como cada instância do enum é compartilhada por todos que a possuem, a informação não poderia ser diferente para cada instância.

## 3.5 SUBSTITUINDO CONDICIONAIS POR POLIMORFISMO

No exemplo apresentado, os nós já possuíam um estado explícito segundo a descrição do algoritmo, porém um cenário para utilização do padrão State muitas vezes é difícil de ser identificado inicialmente. O conceito do que significa um estado para uma determinada entidade do software pode começar a ficar explícito somente no momento da codificação. A repetição de condicionais similares em diversos pontos da mesma classe pode ser um sinal de que seria adequada a refatoração do código em direção ao padrão State.

O mesmo vale para outros padrões que utilizam composição. Um exemplo é quando uma classe possuir um método grande que utiliza condicionais para selecionar dentre alternativas de implementação para um mesmo algoritmo. Nesse caso, a refatoração poderia ser na direção do padrão Strategy. O exemplo apresentado na introdução do livro ilustra bem uma refatoração em direção a composição.

A Figura 3.8 apresenta o processo que pode ser utilizado para extrair a classe que será utilizada para a composição e para a criação das implementações de cada alternativa. O primeiro passo consiste na identificação dos locais com a lógica condicional como descrito acima e na sua extração para métodos que a princípio ficarão na própria classe. Em seguida, uma nova classe deve ser criada e um atributo do tipo dessa classe deve ser introduzido na classe principal que se deseja refatorar. Note

que a princípio essa classe estará vazia e nenhum uso dela é feita a partir da classe principal.

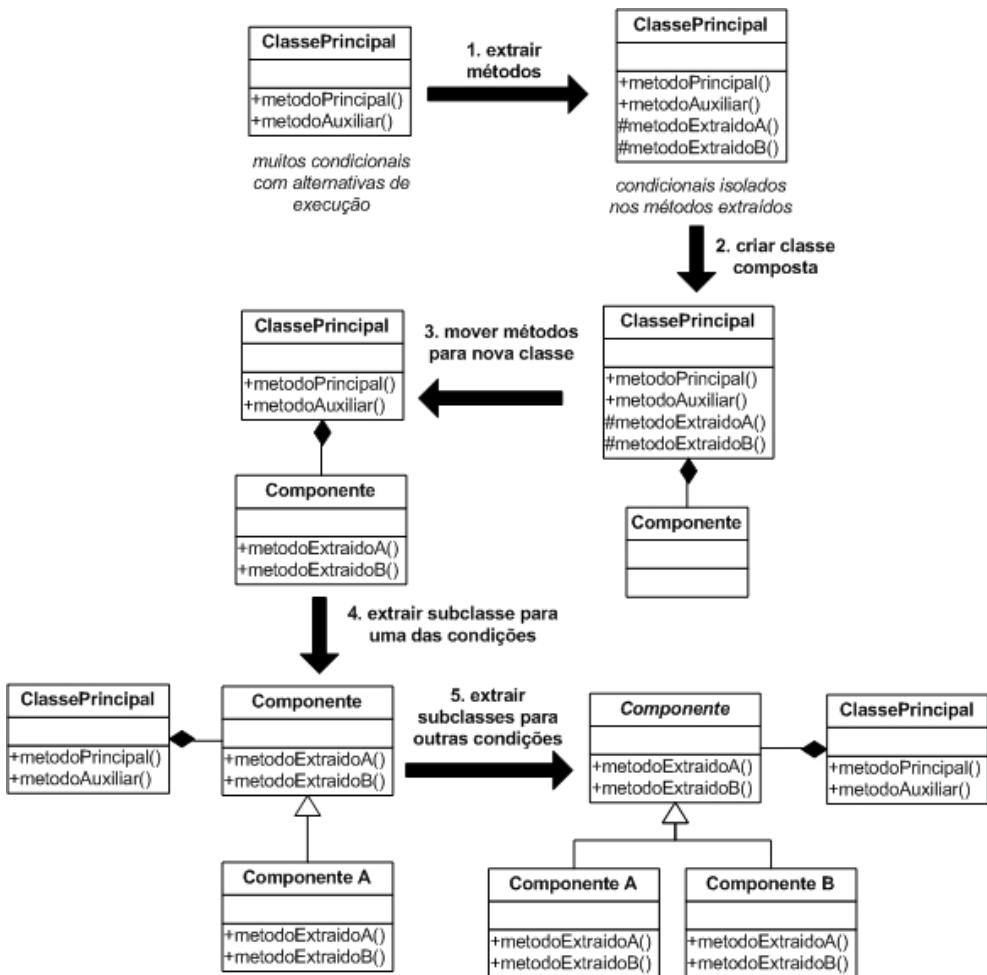


Figura 3.8: Refatorando em Direção a Composição

Com a nova classe criada, o próximo passo é mover os métodos extraídos na primeira etapa da refatoração para ela. Cada método deve ser passado para a nova classe como método público e os locais onde era invocado devem agora chamá-lo a partir do atributo criado. É aconselhável que cada método seja migrado separadamente, executando testes em cada passo. Uma questão que deve ser considerada

nesse ponto é relacionada aos dados que são utilizados pela porção da lógica que está indo para outra classe. Quando a informação for um parâmetro relacionado com o algoritmo em si, normalmente ele é incluído como um atributo da classe que está sendo extraída. Quando é uma informação da classe principal, normalmente ele é passado como parâmetro, ou a própria instância da classe é passada como parâmetro.

Depois que a lógica condicional for migrada para outra classe, é possível começar a extrair subclasses com cada uma das implementações. O conteúdo de um condicional na superclasse deve ser movido para uma subclasse. Adicionalmente, é preciso atribuir uma instância dessa classe na classe principal quando ela precisar ser invocada. Essa troca de instância é realizada normalmente no momento que a variável utilizada nos condicionais é alterada. Por exemplo, se a refatoração for para o padrão State, então a mudança da instância deve ocorrer no momento da mudança de estado. Com isso, a lógica que antes era selecionada com condicional, agora será invocada através de polimorfismo. As duas listagens a seguir apresentam respectivamente o antes e o depois da criação de uma nova subclasse.

```
public class Componente {  
    public void metodoExtraido() {  
        if (possibilidadeA) {  
            //lógica referente ao cenário A  
        } else if(possibilidadeB) {  
            //lógica referente ao cenário B  
        } else if(possibilidadeC) {  
            //lógica referente ao cenário C  
        }  
    }  
}  
  
public class Componente {  
    public void metodoExtraido() {  
        if (possibilidadeB) {  
            //lógica referente ao cenário B  
        } else if(possibilidadeC) {  
            //lógica referente ao cenário C  
        }  
    }  
}  
  
public class ComponenteA extends Componente {  
    @Override
```

```
public void metodoExtraido() {  
    //lógica referente ao cenário A  
}  
}
```

Depois que uma subclasse for extraída, a refatoração deve ser concluída com a extração das outras subclasses. É importante não tentar criar todas as subclasses de uma só vez. A cada subclasse extraída, deve-se executar os testes para verificar se ela foi realizada corretamente. Ao finalizar a extração, a superclasse pode ficar com algum comportamento *default*, caso exista, ou os métodos podem ser transformados em abstratos caso toda lógica tenha sido movida.

## 3.6 COMPONDO COM MÚLTIPLOS OBJETOS - OBSERVER

*“A visão de um observador imparcial é uma janela para o mundo.”*

– Kenneth L. Pike

Nos exemplos vistos até o momento, a composição ocorre apenas com uma instância. Nessa seção será apresentado um padrão que utiliza a composição com múltiplos objetos: o `Observer`. Esse é um padrão muito importante e que frequentemente é necessário em aplicações. Esse padrão é aplicável quando existe um objeto cujos eventos precisam ser observados por outros objetos. Um exemplo seria um componente gráfico cujos eventos precisam ser tratados pela aplicação. Um outro exemplo, talvez menos óbvio, é um objeto de dados cujas mudanças demandam mudanças em outras partes do sistema.

A próxima seção apresenta um exemplo do padrão `Observer`, onde mudanças em um objeto são notificadas a outros objetos interessados. Em seguida esse padrão será analisado de forma mais profunda, avaliando sua estrutura básica e suas consequências. Por fim, serão mostradas algumas APIs existentes que fazem uso desse padrão.

### Notificando mudanças em uma Carteira de Ações

O exemplo que será abordado envolve um objeto que representa uma carteira de ações. Essa carteira possui um mapa com os códigos das ações e sua respectiva quantidade. De acordo com as ações do usuário ou com gatilhos configurados na aplicação, ações podem ser compradas ou vendidas alterando a quantidade de ações. A questão é que existem diversas classes interessadas em saber quando uma informação é alterada nessa classe para poder executar a sua lógica. Por exemplo, um

componente que exibe um gráfico com a quantidade de cada ação da carteira precisa saber quando houve uma mudança para ser atualizado. Outro exemplo seria um componente que fizesse um log das alterações realizadas. Poderia também existir um componente para fazer a auditoria dos dados.

Caso as notificações sejam feitas individualmente para cada necessidade, a classe que representa a carteira de ações ficaria acoplada as classes que precisaria notificar. Isso não é desejável. Além de dificultar mudanças nas classes envolvidas, tornaria complicado a adição de novas classes interessadas em mudanças nessas informações.

Para resolver esse problema, o padrão `Observer` será implementado na solução. Será criada uma interface `Observador`, apresentada na listagem a seguir, a qual deve ser implementada pelas classes que desejam receber os eventos de mudança na quantidade das ações. Implementações dessa interface poderão ser registradas na classe `CarteiraAcoes`, cuja listagem também está apresentada, através do método `addObservador()`. Dessa forma, todos os observadores registrados receberão uma notificação quando uma mudança for realizada. Observe que o método `notificar()` chama o método `mudancaQuantidade()` da interface `Observador` em todas as instâncias registradas.

```
public interface Observador {  
    void mudancaQuantidade(String acao, Integer qtd);  
}  
  
public class CarteiraAcoes {  
    private Map<String, Integer> acoes = new HashMap<>();  
    private List<Observador> obs = new ArrayList<>();  
  
    public void adicionaAcoes(String acao, Integer qtd) {  
        if(acoes.containsKey(acao)){  
            qtd += acoes.get(acao);  
        }  
        acoes.put(acao, qtd);  
        notificar(acao, qtd);  
    }  
  
    private void notificar(String acao, Integer qtd) {  
        for(Observador o: obs)  
            o.mudancaQuantidade(acao, qtd);  
    }  
}
```

```
    public void addObservador(Observador o) {  
        obs.add(o);  
    }  
}
```

A seguir, são apresentadas duas listagens de classes que implementam a interface Observador: AcoesLogger e GraficoBarras. A classe AcoesLogger possui uma implementação bem simples e apenas registra a nova quantidade recebida no console. Já a classe GraficoBarras utiliza o componente de geração de gráficos JFreeChart para a criação de uma janela com um gráfico de barras que é atualizado toda vez que uma mudança é notificada.

```
public class AcoesLogger implements Observador {  
    public void mudancaQuantidade(String acao, Integer qtd) {  
        System.out.println("Alterada a quantidade da ação "  
                           + acao + " para " + qtd);  
    }  
}  
  
public class GraficoBarras implements Observador {  
  
    private DefaultCategoryDataset dataset;  
    private JFrame frame = new JFrame();  
  
    public GraficoBarras() {  
        dataset = new DefaultCategoryDataset();  
        JFreeChart chart = ChartFactory.createBarChart(  
            "Carteira de Ações", "Siglas",  
            "Quantidade", dataset, PlotOrientation.VERTICAL,  
            false, true, false);  
        ChartPanel chartPanel = new ChartPanel(chart);  
        frame.setContentPane(chartPanel);  
        frame.setSize(500, 270);  
        frame.setVisible(true);  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
  
    public void mudancaQuantidade(String acao, Integer qtd) {  
        dataset.setValue(qtd, "Quantidade", acao);  
    }  
}
```

## UTILIZANDO O JFREECHART

Se você deseja executar o exemplo apresentado, precisa baixar o JFreeChart no endereço <http://www.jfree.org/jfreechart/> e adicionar os arquivos jfreechart-1.x.x.jar e jcommon-1.x.x.jar no classpath. Os gráficos desse framework sempre são divididos em duas partes: o conjunto de dados e a sua representação visual. No exemplo apresentado, o atributo dataset da classe DefaultCategoryDataset representa o conjunto de dados e a variável local chart representa a parte visual. No caso, com a alteração do conjunto de dados associado ao gráfico, a representação também é atualizada.

Para finalizar o exemplo, a listagem a seguir apresenta um método main() que associa os observadores na carteira de ações e adiciona ações a carteira. No início do código, as instâncias de GraficoBarras e AcoesLogger são criadas e associadas ao objeto da classe CarteiraAcoes. Em seguida, o método Thread.sleep() é utilizado para gerar um intervalo entre cada quantidade inserida na carteira de ações, para ser possível ver de forma progressiva as alterações no gráfico e as modificações registradas no console. A Figura 3.9 apresenta o gráfico gerado no final da execução.

```
public static void main(String[] args) throws Exception {
    GraficoBarras gb = new GraficoBarras();
    AcoesLogger al = new AcoesLogger();
    CarteiraAcoes c = new CarteiraAcoes();
    c.addObservador(gb);
    c.addObservador(al);

    Thread.sleep(2000);
    c.adicionaAcoes("COMP02", 200);
    Thread.sleep(2000);
    c.adicionaAcoes("EMP34", 400);
    Thread.sleep(2000);
    c.adicionaAcoes("ACDF89", 300);
    Thread.sleep(2000);
    c.adicionaAcoes("EMP34", -200);
    Thread.sleep(2000);
    c.adicionaAcoes("COMP02", 150);
}
```

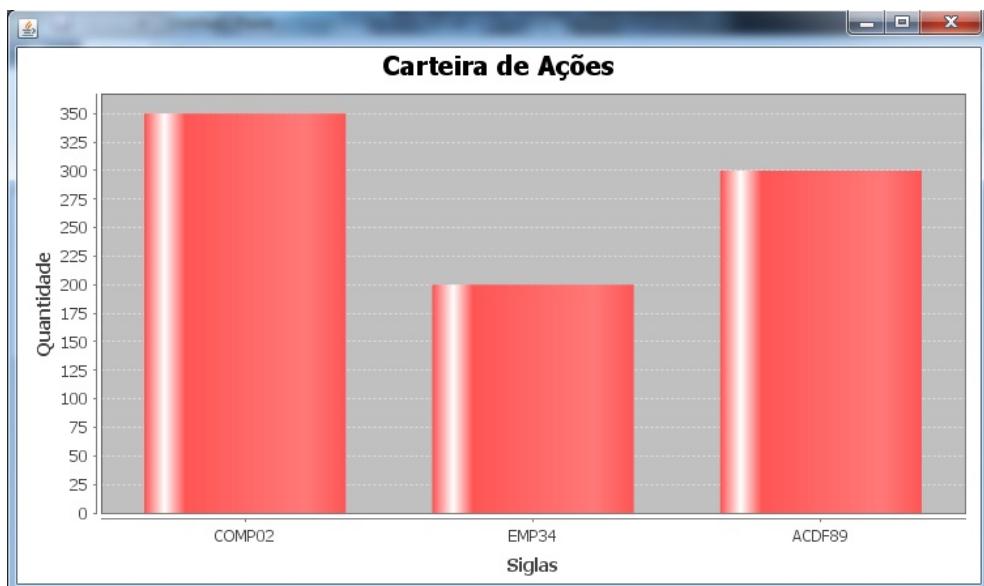


Figura 3.9: Exemplo do Gráfico Gerado pela Aplicação

### Indo a fundo no Padrão Observer

A estrutura do padrão **Observer** é bem simples: a classe que está sendo observada é composta por uma lista de observadores. Além disso, a classe normalmente oferece métodos que permitem a adição e remoção desses observadores em tempo de execução. Dessa forma, quando o estado da classe observada é alterado, todos os observadores são notificados. A Figura 3.10 apresenta um diagrama com a estrutura do padrão.

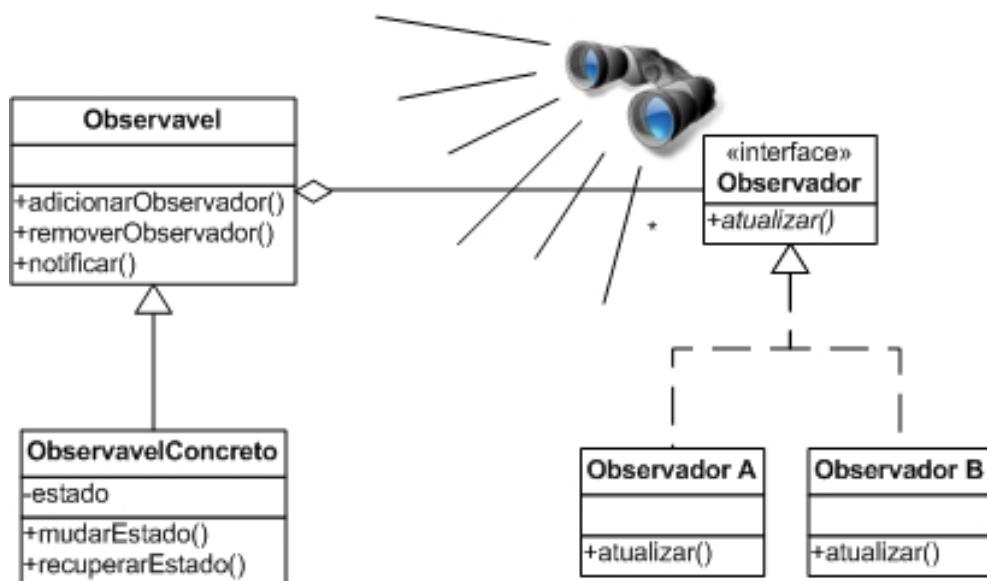


Figura 3.10: Estrutura do Padrão Observer

Observe que no diagrama existe uma hierarquia de classes composta pelas classes `Observable` e `ObservableConcreto` que não havia sido representada no exemplo, onde esses dois papéis são desempenhados pela mesma classe. Essa abstração de objetos observáveis pode ser extremamente útil quando houver mais de uma classe que puder emitir notificações. Imagine, por exemplo, diversos componentes gráficos que podem notificar a ocorrência de eventos gerados por ações do usuário. Nesse contexto pode ser importante a capacidade de abstrair esse comportamento e atribuir observadores ao mesmo tipo de classe.

Uma importante consequência desse padrão é o desacoplamento entre a classe que está sendo observada e a classe observadora. A interface que deve ser implementada pelos observadores é o contrato que une as implementações. Dessa forma, um mesmo observador pode ser adicionado em diferentes objetos, recebendo notificações de todos eles, e vários observadores diferentes podem receber notificações do mesmo objeto. No `Observer` ao invés de possuir uma *hook class*, a classe observada pode possuir um conjunto, sendo que o *hook method* de todas é chamado no momento onde deve ocorrer a notificação.

É importante ressaltar, especialmente ao se falar do `Observer`, que diversas va-

riações são possíveis. Por exemplo, podem existir vários métodos de notificação, para diferentes tipos de evento, que precisam ser implementados pela classe observadora. Nesse caso, cada método seria chamado em uma situação diferente. No exemplo apresentado, poderia haver um método para notificar quando uma ação é removida da carteira. Outro ponto onde pode haver variação é em como os observadores são gerenciados. Eles poderiam, por exemplo, serem passados no construtor e, outras vezes, não faz sentido haver um método de exclusão. Lembre-se que o padrão é apenas uma referência e cabe ao desenvolvedor adaptá-lo ao seu contexto!

Ao lidar com a composição de diversas *hook classes*, é importante levar em consideração como o que ocorre em uma influencia as outras. Por exemplo, o que acontece se uma das classes lança uma exceção? As outras devem continuar sendo executadas? Em caso positivo, deve haver um tratamento de erro separado para cada invocação. Outra questão seria em relação a possibilidade de demora na execução de um observador influenciar a velocidade de notificação dos outros. Dependendo dos requisitos da aplicação, pode ser necessário a chamada do *hook method* de cada classe em um *thread* diferente.

## Padrão Observer nas APIs Java

O padrão `Observer` pode ser visto com frequência em várias APIs da linguagem Java. Um dos motivos dele poder ser visto de forma tão explícita é porque diversas classes permitem que suas mudanças sejam recebidas pela aplicação através de observadores. Como foi dito, essa é uma boa forma de permitir o desacoplamento entre a classe que gera e a classe que recebe o evento.

A interface `ActionListener`, por exemplo, é utilizada no Swing nas classes que desejam tratar eventos de interface. No caso, os objetos que são observados são componentes de interface, como a classe `JButton`. O observador, que também é chamado *listener*, é adicionado no componente através do método `addActionListener()` e precisa implementar o método `actionPerformed()` para tratar os eventos.

Outro exemplo de implementação do `Observer`, agora nas APIs do Java EE, é o `MessageListener` para o recebimento de mensagens JMS. Essa interface precisa ser implementada por classes que desejam receber mensagens de um serviço de mensageria, as quais devem implementar o método `onMessage()` para tratar cada mensagem recebida. Uma instância de `MessageConsumer` com a fila ou o tópico de origem das mensagens é onde o *listener* deve ser configurado. Os Message Driven Beans, que são EJBs responsáveis por tratar mensagens JMS, também implementam essa interface.

Há dezenas de outros exemplos dentro das clássicas APIs do Java. Todos os listeners de sessão das servlets, os listeners de fase da JSF e os listeners de mudanças de valores em entidades na JPA são bons exemplos de implementação do `Observer`, que desacoplam seu código e facilitam a extensão. Todos os observadores podem trabalhar de forma totalmente independente, sem ter conhecimento da existência dos outros.

Uma coisa interessante em aprender padrões é que a partir deles fica mais fácil compreender o funcionamento de APIs e frameworks que utilizamos. É como se passássemos a olhar para eles com outros olhos. Imagino que seja algo comparável a quando estudamos física, química ou biologia e passamos a compreender melhor o mundo ao nosso redor. Ao perceber que um padrão foi implementado em uma solução, é mais fácil entender com utilizar aquela API e quais as consequências do uso daquela solução naquele contexto.

### A INTERFACE OBSERVER E A CLASSE OBSERVABLE

Uma curiosidade que poucos sabem é que existe desde a JDK 1.0 a interface `java.util.Observer` e a classe `java.util.Observable`. A interface `Observer` possui o método `update()` que recebe como parâmetro a instância de `Observable` que gerou o evento e mais um objeto que é o parâmetro passado para o método `notifyObservers()`. A ideia da classe `Observable` é que ela seja estendida por classes que queiram emitir notificações sobre mudanças em seu estado. Ela possui métodos implementados para gerenciar e notificar as instâncias de `Observer`.

O que é mais interessante é que essa classe não é utilizada em nenhum lugar da API da JDK, mesmo o padrão `Observer` sendo utilizado em diversos contextos. A questão principal é que essas classes consideram o `Observer` como uma implementação pronta e não como uma solução que pode ser adaptada para diversos contextos. Por mais que a solução apresentada por essas classes seja reutilizável, dificilmente ela será mais adequada do que uma implementação do padrão para um contexto específico.

### 3.7 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Essa capítulo apresentou como a composição pode ser utilizada para inserir pontos de extensão que permitem que uma porção de lógica possa ser inserida durante a execução de uma classe. Esses pontos podem ser utilizados para inserir uma lógica específica da aplicação ou de um domínio em uma classe mais genérica, tornado-a mais reutilizável. Utilizando *hook classes*, a adaptação de comportamento é feita através da combinação de objetos, podendo o desenvolvedor criar novos componentes para criação da configuração de classes final.

A extensão de comportamento pode ocorrer por diferentes motivos e em diferentes contextos. Pode ser delegada para outra classe a execução de um algoritmo que pode ser trocado, como é o caso do padrão *Strategy*. Outro cenário é a delegação da lógica relativa as diferenças de estado de um objeto, sendo esse o foco do padrão *State*. Finalmente, o padrão *Observer* associa a execução de métodos de diversas outras classes a mudança de estado ou a eventos ocorridos em um objeto.

Além de compreender os padrões e esses cenários que favorecem o uso da composição, é importante também conhecer os princípios e consequências por trás dessas decisões. A composição pode inclusive ser combinada com outros tipos de solução, como no padrão *Bridge* onde ela é combinada com o uso de herança em uma hierarquia diferente.



## CAPÍTULO 4

# Composição Recursiva

*“Recursão: ver recursividade. Recursividade: se ainda não entendeu, ver recursão”*

– Piada Nerd

No capítulo anterior foi visto como a composição pode ser utilizada para combinar classes de diferentes tipos para permitir que a funcionalidade seja estendida. Através do polimorfismo, a classe composta pode abstrair a implementação da classe que a está compondo, permitindo que diversas classes diferentes possam ser colocadas naquela posição. Agora imagine se uma classe puder ser composta pela sua própria abstração, ou, em outras palavras, se uma classe possuir um atributo de sua superclasse ou de uma de suas interfaces. Isso permitiria que uma instância da mesma classe poderia ser utilizada para compô-la, tornando possível a criação de uma grande estrutura dessa forma.

Eu sei que isso parece ser muito abstrato, mas vamos tentar pensar em um exemplo do mundo físico para ilustrar esse conceito. Imagine uma peça de Lego que provê na parte de cima uma forma para que outras peças possam ser encaixadas e, na parte de baixo, a forma do encaixe para que ela seja ligada na parte de cima. Como a

mesma peça possui as duas formas de encaixe, possível ligar e combinar várias peças do mesmo tipo em diferentes configurações. Quando consideramos que podem existir peças diferentes com o mesmo formato de encaixe, as possibilidades são ainda maiores.

Voltando ao mundo do software, é como se a forma que uma peça provê em sua parte superior fosse a interface da classe. Analogamente, o tipo de um atributo seria o encaixe disponibilizado para que uma outra peça possa se encaixar. Dessa forma, uma classe que implementa uma interface se encaixa em um atributo daquele tipo. Na composição simples, utilizando essa ideia é possível conectar apenas dois objetos. Mas quando uma classe provê uma interface e um atributo do mesmo tipo, é possível criar uma estrutura com diversos objetos. Nesse caso, as próprias possibilidades de configurações dessa estrutura já tornam o software mais flexível.

## 4.1 COMPONDÔ UM OBJETO COM SUA ABSTRAÇÃO

A recursividade é um conceito amplamente utilizado em programação. Uma função recursiva, por exemplo, é aquela que chama a si mesma como parte da sua lógica. Em um algoritmo recursivo, a função é chamada novamente para diminuir o problema que precisa ser resolvido. Dessa forma, ele vai sendo reduzido até o ponto que sua resolução é trivial. Esse é chamado de ponto de parada.

Um exemplo clássico e simples de função recursiva é o fatorial de um número, que consiste na multiplicação de todos os números inteiros e positivos menores ou iguais aquele número. A listagem a seguir apresenta a implementação do fatorial utilizando recursão. Observe que, pela definição de fatorial, o fatorial de um número  $n$  é  $n$  vezes o fatorial de  $n-1$ . O fato de que o fatorial de 1 é 1 é utilizado como critério de parada.

```
public static int factorial(int n){  
    if(n == 1)  
        return 1;  
    return n * factorial(n-1);  
}
```

Quando criamos uma estrutura com objetos, é possível utilizar também o conceito de recursividade para que ela seja definida utilizando sua própria definição. Isso é feito quando uma classe contém um atributo do próprio tipo da classe. Esse tipo de estrutura é muito utilizado para definir muitas estruturas de dados, como

listas ligadas, árvores e grafos. Por exemplo, no Capítulo 3 foi apresentada busca em profundidade de grafos, onde a classe No possuía uma lista de instâncias de No.

Para exemplificar vamos considerar o código do elemento de uma lista ligada definido a seguir. A classe Elemento possui um atributo proximo que possui o mesmo tipo. Dessa forma, é possível encadear diversas instâncias dessa classe formando uma lista. O método contar() é um método recursivo, no qual o próprio método é chamado novamente, porém na instância que estiver no atributo proximo. A condição de parada é quando um elemento não possui próximo, significando que ele é o último elemento da lista e deve retornar um. Se houver um próximo elemento, então a contagem deve retornar um mais a contagem do próximo.

```
public class Elemento{  
    private Object valor;  
    private Elemento proximo;  
  
    public Elemento(Object valor){  
        this.valor = valor;  
    }  
    public Object getValor(){  
        return valor;  
    }  
    public Elemento getProximo(){  
        return proximo;  
    }  
    public void setProximo(Elemento proximo){  
        this.proximo = proximo;  
    }  
    public int contar(){  
        if(proximo == null)  
            return 1;  
        return 1 + proximo.contar();  
    }  
}
```

Até então não tem muita diferença entre as estruturas que podem ser construídas em uma linguagem estruturada, como C, onde pode-se criar estruturas de dados utilizando a construção struct. O grande diferencial dessa definição recursiva de estruturas é a utilização do polimorfismo. Dessa forma, o tipo base utilizado na estrutura pode ser uma interface ou uma superclasse que possuem diversas implementações. Sendo assim, diversos tipos podem ser utilizados na criação da estrutura

e a implementação de um pode ser abstraída pelos outros. Adicionalmente, essa estrutura também pode ser facilmente estendida através da criação de novos tipos que possuam a mesma implementação.

A Figura 4.1 apresenta diferentes possibilidades para a implementação da composição recursiva. Uma delas é a composição recursiva ser definida na superclasse. Dessa forma, todas as subclasses irão poder ser compostas por uma classe da hierarquia. Outra possibilidade é quando a composição recursiva é definida em uma ou mais subclasses. Nessa opção, apenas alguns tipos podem ser utilizados para a implementação da recursividade e outros serão pontos finais na estrutura. Nesse capítulo serão apresentados padrões que utilizam essas diferentes alternativas de implementação.

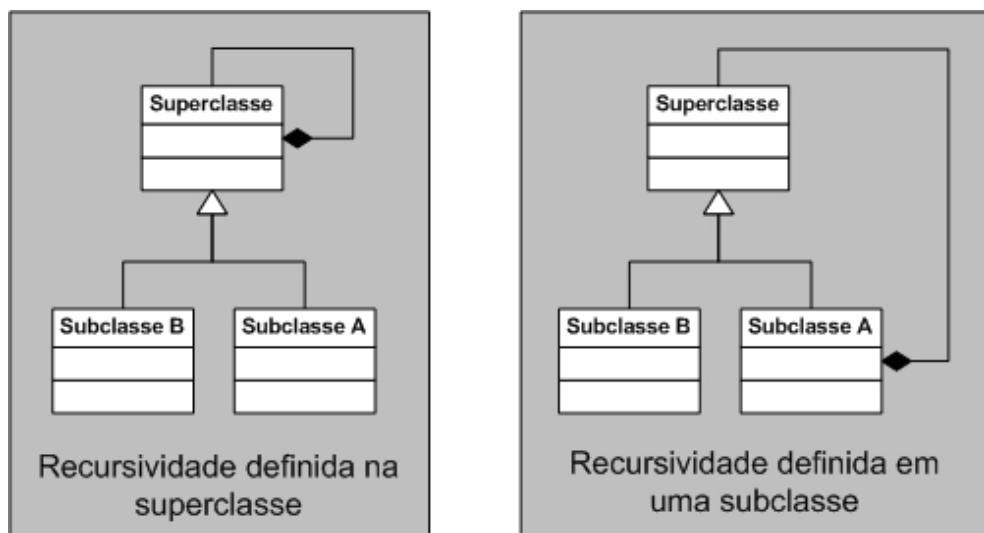


Figura 4.1: Possibilidades para Composição Recursiva

É importante ressaltar que existem variações das possibilidades apresentadas. A composição, por exemplo, pode ser de uma ou múltiplas instâncias, o que geraria soluções diferentes. Dependendo do domínio, pode também haver mais de um atributo que possua uma definição recursiva, mas que possua diferentes significados para a classe. Em uma árvore, por exemplo, uma lista de nós pode representar os nós filhos e um atributo simples pode representar o pai.

## 4.2 COMPOSITE - QUANDO UM CONJUNTO É UM INDIVÍDUO

*“Para sempre é composto de agora.”*

– Emily Dickinson

Um dos erros de pessoas que estão iniciando na orientação a objetos é modelar o coletivo entendendo o indivíduo. Uma cesta de maçãs **não é uma** maçã para que o uso de herança seja justificado. Similarmente, um conjunto de processos **não é um** processo, logo também não é correto utilizar herança para representá-lo. Normalmente isso é feito para a subclasse invocar diretamente os métodos da superclasse e, dessa forma, a subclasse define um novo conjunto de métodos, ignorando a interface pública da superclasse.

Apesar do que foi dito, existem situações onde o conjunto também representa um único indivíduo. Um conjunto de produtos pode ser um produto? A resposta é sim dependendo do contexto. Uma loja ou um supermercado frequentemente montam kits com diversos produtos que são vendidos como um produto único. Como representar isso em um sistema? A questão é que esse conjunto de produtos terá uma lógica diferente para diversos fatores, como para o cálculo do preço, por exemplo. Porém, em outros cenários, ele deve ser considerado como um produto comum.

Essa seção apresenta o padrão Composite cujo objetivo é prover uma solução para objetos que representam um conjunto de objetos, mas que compartilham a mesma abstração deles. Este padrão tem o potencial de encapsular uma lógica complexa, dividindo-a em uma hierarquia de classes e simplificando a solução final. Todos que já utilizaram um Composite em um projeto real, sabem que é um padrão realmente muito poderoso.

### Apresentando o Padrão Composite

O padrão Composite possui o objetivo de permitir que a mesma abstração possa ser utilizada para uma instância simples e para seu conjunto. A Figura 4.2 apresenta a estrutura básica que é utilizada no padrão. Uma abstração apresenta uma interface básica com as operações que devem ser executadas tanto nos objetos simples quanto nos compostos. A classe Simples representa um único indivíduo e a classe Composto representa o conjunto. Observe que existe uma relação de composição entre Composto e Abstracao, que mostra que um objeto da classe Composto pode ser composto por outro também da classe Composto.

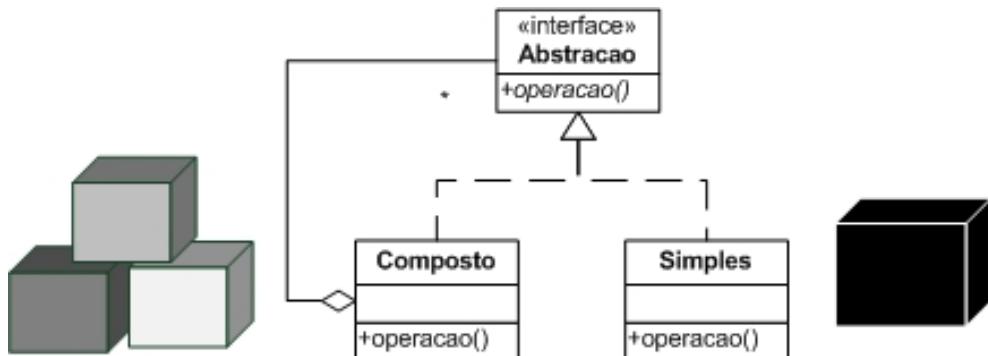


Figura 4.2: Estrutura Básica do Composite

As operações definidas na abstração devem ser implementadas em todas as subclasses. Para as subclasses simples, que não forem compostas por outras do tipo da abstração, pode ser uma implementação direta. Por exemplo, uma classe Produto pode disponibilizar um método para retornar o preço e possuir apenas o acesso a um atributo. O segredo do padrão Composite está na implementação dos métodos compostos, que devem invocar os métodos das instâncias que o compõe e executar uma lógica para criar um resultado único. No exemplo do produto, o produto composto, uma classe KitProdutos, para retornar o preço ele poderia somar o preço dos produtos que o compõe e dar um desconto em cima.

Os objetos organizados com o padrão Composite acabam seguindo uma estrutura de árvore. As folhas seriam as classes simples, que não possuem a composição. Apesar de no diagrama da Figura 4.2 apenas um tipo de classe simples ser representada, é importante ressaltar que podem haver diversas classes desse tipo com diferentes implementações. Os ramos seriam representados pelas classes compostas, que seria composta pelos seus filhos. O número de filhos de uma classe composta varia com a necessidade da aplicação, podendo ser duas instâncias ou um conjunto.

O grande benefício do Composite é tornar totalmente transparente para os clientes o fato de estarem trabalhando com um objeto simples ou composto. Isso torna muito simples adicionar uma implementação de um objeto composto em uma hierarquia já existente na aplicação.

## COMPOSITE EM COMPONENTES GRÁFICOS

Um contexto em que o padrão Composite é muito utilizado é para a criação de componentes gráficos. Muitos componentes são formados a partir da composição de outros componentes existentes, porém são tratados como um só. Por exemplo, um componente de seleção de arquivos normalmente possui botões e uma caixa de texto. Dessa forma, quando um método `setEnabled()` for chamado nesse componente composto, ele deverá coordenar a invocação desse método em todos os seus componentes internos.

Em Java, esse conceito é utilizado em suítes gráficas para aplicações desktop, como no Swing e no SWT, e em frameworks com componentes para páginas web, como o JSF. O framework SwingBean, por exemplo, possui um componente que representa um formulário e é composto por diversos outros componentes.

## Modelando a Representação de Trechos Aéreos

Para exemplificar a aplicação do padrão Composite, vamos modelar as classes que representam trechos aéreos em uma aplicação de turismo. Para esse exercício vamos considerar que as informações principais de um trecho aéreo são sua origem, seu destino e o seu preço. Muitas vezes, quando um cliente solicita um voo a partir de uma origem e um destino, não existe um voo direto que conecta aqueles dois aeroportos. Nesse caso, normalmente busca-se então um voo que pousa em um aeroporto intermediário, de onde o cliente pode pegar um outro voo até o seu destino. Nesse caso, além do preço dos dois trechos, deve-se adicionar a taxa de conexão cobrada pelo aeroporto intermediário. A questão é independente do trecho aéreo ser composto por mais de um trecho ou não, em certos pontos o sistema deve tratá-los de forma similar.

O primeiro passo é a definição de uma interface que irá definir quais serviços serão disponibilizados pela classe que representar o trecho simples e o composto. Pela descrição do problema, o trecho deve ser capaz de retornar a origem, o destino e o preço. Sendo assim, a listagem a seguir apresenta a interface `TrechoAereo` que define métodos para recuperação dessas informações. Em seguida, é apresentado o código da classe `TrechoSimple`, que possui uma implementação cuja lógica é ape-

nas o acesso a atributos.

```
public interface TrechoAereo {  
    public String getOrigem();  
    public String getDestino();  
    public double getPreco();  
}  
  
public class TrechoSimples implements TrechoAereo{  
    private String origem;  
    private String destino;  
    private double preco;  
  
    public TrechoSimples(String origem, String destino, double preco) {  
        this.origem = origem;  
        this.destino = destino;  
        this.preco = preco;  
    }  
    public String getOrigem() {  
        return origem;  
    }  
    public String getDestino() {  
        return destino;  
    }  
    public double getPreco() {  
        return preco;  
    }  
}
```

Para a implementação do trecho composto, será aplicado o padrão Composite. Nesse caso, o trecho composto possui dois atributos, respectivamente para representar o primeiro e o segundo trecho que o compõe. Também existe um atributo que representa a taxa de conexão a ser paga de forma adicional ao preço de cada trecho individualmente. Note que no construtor é feita uma verificação se o segundo trecho começa no mesmo local em que o primeiro termina. Pode ser observado que as operações que precisavam ser implementadas delegam parte da execução para os trechos que compõe o objeto.

```
public class TrechoComposto implements TrechoAereo {  
    private TrechoAereo primeiro;  
    private TrechoAereo segundo;
```

```
private double taxaconexao;

public TrechoComposto(TrechoAereo primeiro, TrechoAereo segundo,
                      double taxaconexao) {
    this.primeiro = primeiro;
    this.segundo = segundo;
    this.tacanexao = taxaconexao;
    if(primeiro.getDestino() != segundo.getOrigem())
        throw new RuntimeException("O destino do primeiro" +
                                   "não é igual a origem do segundo");
}
public String getOrigem() {
    return primeiro.getOrigem();
}
public String getDestino() {
    return segundo.getDestino();
}
public double getPreco() {
    return primeiro.getPreco() + segundo.getPreco() +
           taxaconexao;
}
}
```

A Figura 4.3 apresenta um exemplo de como um voo com três trechos seria representado utilizando a estrutura de classes apresentada. A seguir também é apresentada uma listagem que mostra o código para a criação dos trechos simples e compostos. Observe pelo exemplo, que `TrechoComposto` pode ser composto tanto por instâncias de `TrechoSimples` quanto por instâncias do próprio `TrechoComposto` de forma transparente, criando uma estrutura de árvore.

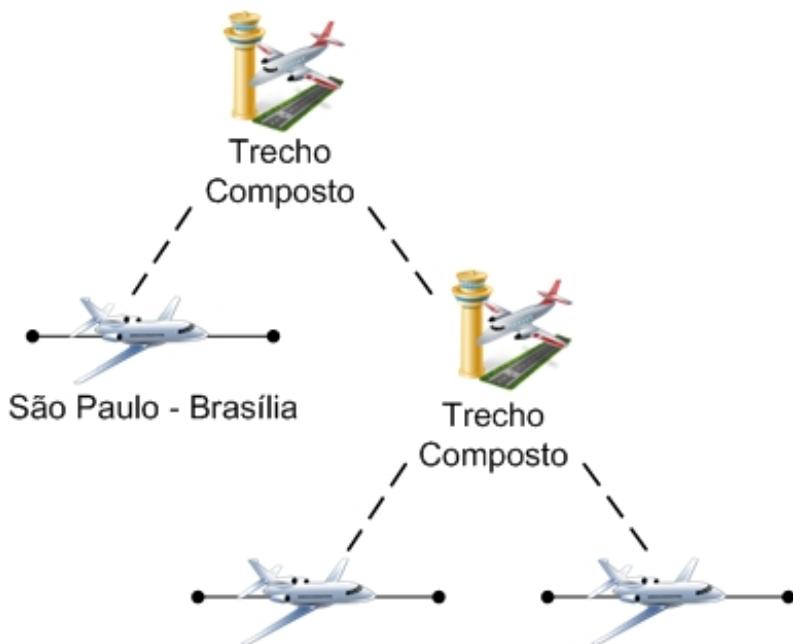


Figura 4.3: Representação de trechos aéreos

```
TrechoSimples ts1 = new TrechoSimples("São Paulo", "Brasília", 500);
TrechoSimples ts2 = new TrechoSimples("Brasília", "Recife", 300);
TrechoSimples ts3 = new TrechoSimples("Recife", "Natal", 350);
TrechoComposto tc1 = new TrechoComposto(ts2, ts3, 30);
TrechoComposto tc2 = new TrechoComposto(ts1, tc1, 50);
```

### **POR QUE TRECHO COMPOSTO NÃO PODE ESTENDER TRECHOSIMPLES?**

Pode parecer tentador a princípio fazer com que TrechoComposto estenda TrechoSimples ao invés de criar uma interface comum entre as duas. Se isso ocorrer, a classe TrechoComposto iria simplesmente ignorar a estrutura de dados da classe TrechoSimples e ainda teria que lidar com seu construtor. Isso geraria um mal cheiro de código chamado de *Refused Bequest*, que ocorre quando uma subclasse não utiliza a estrutura e os métodos herdados da sua superclasse. Para que aquela herança é dada para a classe se ela não a utiliza?

## **4.3 ENCADEANDO EXECUÇÕES COM CHAIN OF RESPONSIBILITY**

*“Nenhuma corrente pode ser mais forte que seu elo mais fraco.”*

– Provérbio Popular

Quando em um software uma funcionalidade é executada, normalmente existem diversos passos que precisam ser executados em sequência. A dificuldade ocorre quando precisa-se de flexibilidade na configuração desses passos. Por exemplo, pode ser necessária a modificação da ordem dos passos e até mesmo para a inserção de novos passos. Um outro objetivo seria a possibilidade de reutilização desses passos em outros processamentos ou para outras aplicações.

Chain of Responsibility é um padrão de projeto que cria uma cadeia de execução onde cada elemento processa as informações e em seguida delega a execução ao próximo da sequência. Em sua implementação tradicional, os elementos são percorridos até que um deles faça o tratamento da requisição, encerrando a execução depois disso. Como alternativa, também é possível criar uma cadeia de execução onde cada um executa sua funcionalidade até que a cadeia termine ou ela seja explicitamente finalizada por um dos elementos.

### **Recuperação de um Arquivo Remoto**

Para ficar mais concreto o entendimento do padrão, imagine que um arquivo, como uma lista de certificados digitais revogados, é atualizado de forma periódica

em um servidor remoto, contendo a data em que sua validade é expirada. Para tornar recuperações frequentes mais eficientes a aplicação criou dois tipos de cache, sendo um em memória e outro na base de dados. Dessa forma, primeiro verifica-se se existe um arquivo válido em memória, em seguida no banco de dados e somente então, se não for encontrado, o arquivo é recuperado do servidor remoto.

Em uma implementação tradicional, toda essa lógica seria implementada na mesma classe, com diversos condicionais em sequência para verificar se uma determinada estratégia seria adequada para recuperação do arquivo. Utilizando o padrão *Chain of Responsibility*, cada possibilidade é implementada em uma classe diferente, as quais são encadeadas em um fluxo de execução. Cada uma delas verifica se o arquivo existe e é válido, retornando o mesmo em caso positivo e delegando a execução para a próxima instância da cadeia em caso negativo. A Figura 4.4 representa essa cadeia de execução.



Figura 4.4: Cadeia de Execução para Recuperação de Arquivo

Para implementarmos essa sequência de execução, o primeiro passo é definirmos uma superclasse que define um elemento da cadeia. A classe *RecuperadorArquivo*, apresentada na próxima listagem, possui um atributo do seu mesmo tipo chamado *proxima* que representa o próximo elemento. Essa classe também define um método abstrato chamado *recuperarArquivo()* que é um *hook method* que deve ser implementado pelas subclasses para a recuperação do arquivo de acordo com a estratégia desejada. Além disso, o método *chamarProximo()* é responsável por verificar se existe um próximo elemento e invocá-lo. Adicionalmente, o método *recuperar()* tenta recuperar o arquivo, retornando-o se ele for válido e chamando o próximo elemento da cadeia em caso negativo.

```
public abstract class RecuperadorArquivo {
```

```
private RecuperadorArquivo proximo;

public RecuperadorArquivo(RecuperadorArquivo proximo) {
    this.proximo = proximo;
}

public Arquivo recuperar(String nome){
    Arquivo a = recuperaArquivo(nome);
    if(a==null || !a.isValido())
        return chamarProximo(nome);
    else
        return a;
}

protected Arquivo chamarProximo(String nome) {
    if(proximo == null)
        throw new RuntimeException("Não foi possível " +
            "recuperar o arquivo");
    return proximo.recuperar(nome);
}

protected abstract Arquivo recuperaArquivo(String nome);
}
```

Cada subclasse de `RecuperadorArquivo` deve implementar o método `recuperaArquivo()` buscando o arquivo em uma fonte diferente. Pela descrição do problema, as implementações irão buscar o arquivo no banco de dados, de um cache em memória e de em um servidor remoto. A listagem a seguir apresenta a classe que faz a busca em um cache em memória para exemplificar como seria uma dessas implementações. O método `recuperaArquivo()` verifica se existe o arquivo no mapa e retorna nulo caso não exista. Para armazenar novos arquivos no mapa, o método `chamarProximo()` também é implementado para permitir que se tenha acesso ao arquivo retornado pelo próximo.

```
public class RecuperadorCacheMemoria extends RecuperadorArquivo {
    private Map<String, Arquivo> cache = new HashMap<>();

    public RecuperadorCacheMemoria(RecuperadorArquivo proximo) {
        super(proximo);
    }

    protected Arquivo recuperaArquivo(String nome) {
        if(cache.containsKey(nome))
            return cache.get(nome);
    }
}
```

```
        return null;
    }

    protected Arquivo chamarProximo(String nome) {
        Arquivo a = super.chamarProximo(nome);
        cache.put(nome, a);
        return a;
    }
}
```

### ALGUÉM VIU UM TEMPLATE METHOD?

Desafio o leitor a observar melhor o exemplo de código dessa seção e achar uma implementação do padrão Template Method! O método `recuperar()` da classe `RecuperadorArquivo` implementa uma lógica comum aos elementos da cadeia de execução e delega os pontos variáveis a *hook methods* que serão implementados pelas subclasses. Observe que o método `chamarProximo()`, também chamado pelo Template Method, apesar de não ser abstrato, também pode ser considerado um *hook method*. Pelo exemplo da classe `RecuperaCacheMemoria` pode-se perceber como esse método pode ser sobreescrito pela subclasse para a adição da funcionalidade.

Como já ressaltamos, é comum haver uma ligação forte entre diferentes padrões de projeto. Muitas vezes eles aparecem em conjunto e até é difícil diferenciá-los.

## Estrutura e Alternativas de Implementação

O padrão *Chain of Responsibility* coloca cada elemento que irá participar do processamento da lógica como o elemento de uma lista ligada, onde cada elemento tem acesso ao próximo na cadeia de execução. A Figura 4.5 mostra como a composição recursiva é utilizada na implementação da estrutura de classes. Nesse caso, uma *hook class* do mesmo tipo é definida na superclasse, significando que todo tipo de elemento da cadeia pode ser composto por um próximo elemento. Caso esse elemento seja nulo, ou possua um elemento que represente um `Null Object`, será marcado o final da cadeia de execução.

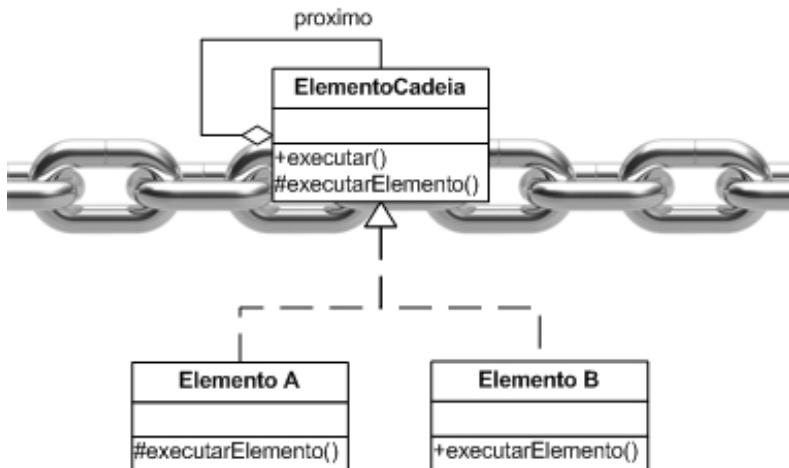


Figura 4.5: Estrutura do Chain of Responsibility

Nessa representação, o método `executar()` disponibiliza como API pública a execução de todos elementos da cadeia e o método `executarElemento()` representa a execução da lógica somente daquele elemento. Dessa forma, o método `executar()` invoca primeiro a lógica implementada na própria classe e em seguida invoca de forma recursiva o próprio método `executar()` no próximo elemento.

Como alternativa de implementação, o método `executarElemento()` pode receber como parâmetro o próximo elemento, sendo também responsável pela sua invocação. Dessa forma, cabe a implementação de cada classe decidir em que situações o próximo elemento deve ser invocado e em que momento essa invocação deve acontecer. Utilizando essa alternativa, cada implementação pode incluir lógica que será executada tanto antes quanto depois da execução do próximo elemento.

A partir da implementação do Chain of Responsibility adquire-se uma grande flexibilidade para a configuração da lógica que será executada. Por exemplo, facilmente um novo elemento da cadeia pode ser introduzido, ou um elemento existente pode ser excluído. A própria ordem que os elementos são organizados pode gerar uma grande diferença no final. Por exemplo, para a utilização do RecuperadorArquivo em dispositivos com memória RAM limitada, o elemento que faz o cache em memória poderia facilmente ser excluído da cadeia. Em um outro cenário em que o arquivo pudesse estar disponível em mais de um servidor, mais instâncias do elemento que busca o arquivo remotamente com endereços diferentes, poderiam ser incluídas.

## Filtros em Aplicações Web

Um exemplo do uso do `Chain of Responsibility` em APIs do Java EE são os filtros de aplicações web [?]. Um exemplo da estrutura básica de um filtro está representada na próxima listagem. A classe que representa o filtro deve implementar a interface `Filter` e, além de implementar os métodos `init()` e `destroy()` que servem respectivamente para a inicialização e finalização, ainda é preciso incluir o método `doFilter()` que é onde a lógica será efetivamente implementada.

```
public class FiltroExemplo implements Filter{  
  
    public void doFilter(ServletRequest req,  
                         ServletResponse resp,  
                         FilterChain chain){  
  
        //antes do próximo filtro  
        chain.doFilter(req,resp);  
        //depois do próximo filtro  
  
    }  
    public void destroy{}  
    public void init(FilterConfig config){}  
}
```

O método `doFilter()` recebe como um dos parâmetros uma instância da classe `FilterChain`, que representa justamente a cadeia de filtros que está sendo executada. Ao ser chamado o método `doFilter()` nessa instância, o controle da execução é passado para o próximo filtro ou, se não houver próximo, para o servlet que irá tratar a requisição. Observe que como o próximo filtro é invocado dentro do método `doFilter()`, pode ser adicionado funcionalidade tanto antes quanto depois da sua execução. Inclusive o próximo filtro nem precisa ser invocado, como no caso da requisição ser direcionada a uma página de erro. Outra coisa que pode ser feita é a modificação dos parâmetros que serão passados para os próximos filtros, inclusive encapsulando-os com a utilização de `Proxies` (um padrão que será visto no próximo capítulo).

Um dos usos mais clássicos para filtros em aplicações web é para verificar se o usuário está logado. Nesse caso, o filtro verifica se existe um usuário válido na sessão e redireciona para a página de login em caso negativo. Porém existem vários outros usos para filtros, como: realização de controle de acesso; fazer uma busca por

parâmetros maliciosos; iniciar e finalizar uma transação com a base de dados; e modificar a saída gerada pelo servlet. Em geral os filtros são utilizados para funcionalidades que precisam ser executadas em diversas requisições diferentes. A utilização do **Chain of Responsibility** em sua implementação permite que a ordem dos filtros seja modificada e que filtros possam ser facilmente introduzidos ou removidos.

Essa modelagem utilizada para os filtros é mesma dos *interceptors* do framework Struts 2 [?]. Um *interceptor* tem uma função bem parecida com um filtro, que é de realizar algum processamento antes e/ou depois do tratamento de uma requisição web. Grande parte das funcionalidades do framework, como a injeção dos parâmetros do *request* ou a introdução de objetos da sessão do usuário, é implementada através dos *interceptors*. Os interceptors que atuam para cada requisição são configurados através de um arquivo XML. Nesse caso, a utilização do padrão **Chain of Responsibility** permite que esses componentes do framework possam ser facilmente removidos ou que novos componentes específicos da aplicação possam ser adicionados.

## 4.4 REFATORANDO PARA PERMITIR A EXECUÇÃO DE MÚLTIPAS CLASSES

Uma evolução comum que ocorre em aplicações é a necessidade de que mais uma ação seja realizada como resposta a um evento. Por exemplo, imagine que ao receber uma informação a aplicação precise persisti-la em uma base de dados. Com o tempo, novos requisitos vão surgindo que exigem que novas ações sejam tomadas naquele cenário, como a chamada de um serviço remoto ou o registro em uma trilha de auditoria. O objetivo dessa refatoração é permitir que na chamada de um método, diversas classes possam ser executadas. Isso deve ser realizado de forma que não haja um acoplamento entre essas classes e que isso seja transparente os que invocam.

Para que essa refatoração possa ser realizada, é importante que a execução dessa funcionalidade já esteja isolada utilizando a composição simples. Ou seja, que a partir de uma *hook class* configurada seja possível trocar a lógica que será executada, mas não incluir múltiplas classes para a execução. Diferentemente do padrão **Observer**, onde a classe observada precisa gerenciar a existência de múltiplos observadores, nesse caso é desejável que a existência de múltiplas execuções seja transparente para a classe.

Para mostrar a refatoração para os dois padrões apresentados nesse capítulo, o exemplo da classe `GeradorArquivo` será retomado. O objetivo será permitir que diversos pós-processadores possam atuar em cima do arquivo gerado de forma trans-

parente para a classe principal. Após a implementação do padrão Bridge, existe uma interface chamada PosProcessador que deve ser implementada por qualquer implementação que irá compor a classe GeradorArquivos. No exemplo, os dois pós-processadores existentes respectivamente criptografam e compactam o arquivo após a sua geração. Observe que se ambos forem utilizados, a ordem de processamento irá alterar o resultado final, pois um arquivo criptografado e compactado, é diferente de um arquivo compactado e criptografado.

As seções a seguir descrevem como cada um dos padrões apresentados nessa seção poderiam ser implementados para resolver o problema com os requisitos descritos.

## Introduzindo um Composite

Para implementar o padrão Composite, o que precisa ser feito é a implementação de uma classe que possua a mesma interface de um pós-processador e seja capaz de coordenar a execução de diversos pós-processadores. A listagem a seguir mostra a classe PosProcessadorComposto que desempenha esse papel. Observe que ela é composta por um array de pós-processadores que são recebidos no construtor. Adicionalmente, o método processar() executa o mesmo método de forma recursiva nas instâncias de PosProcessador que compõe a classe.

```
public class PosProcessadorComposto implements PosProcessador{  
  
    private PosProcessador[] processadores;  
  
    public PosProcessadorComposto(PosProcessador... p){  
        processadores = p;  
    }  
    public byte[] processar(byte[] bytes){  
        for(PosProcessador p : processadores){  
            bytes = p.processar();  
        }  
        return bytes;  
    }  
}
```

A grande vantagem do uso do Composite nesse caso é que ele não exige nenhuma mudança nas classes existentes. A classe PosProcessadorComposto pode ser adicionada facilmente na classe GeradorArquivo e todos os outros pós-processadores po-

dem facilmente ser incluídos nessa classe. A dificuldade de usar o **Composite** irá aparecer em cenários onde a integração entre as classes envolvidas não acontece sempre da mesma forma. Isso pode exigir a criação de diversas classes que desempenham esse papel, mas possuem uma lógica de integração diferente.

## Refatorando para Chain of Responsibility

Uma outra abordagem para a refatoração seria implementar o padrão **Chain of Responsibility** para permitir que os pós-processadores possam ser organizados em uma cadeia de processamento. Isso vai exigir uma mudança na abstração que representa um pós-processador para incluir a composição recursiva, ou seja, para a adição de uma *hook class* que represente o próximo processador da cadeia. A interface **PosProcessador** precisaria ser representada como uma classe abstrata, cuja listagem está apresentada a seguir.

```
public abstract class PosProcessador{  
  
    private PosProcessador proximo;  
  
    public PosProcessador(PosProcessador prox){  
        proximo = prox;  
    }  
    public PosProcessador(){  
        proximo = nem PosProcessadorNulo();  
    }  
    public byte[] processarCadeia(byte[] bytes){  
        bytes = processar(bytes);  
        return proximo.processarCadeia(bytes);  
    }  
    protected abstract byte[] processar(byte[] bytes);  
}
```

A classe **PosProcessador** possui o atributo **proximo** de seu próprio tipo, que deve ser passado no construtor. Caso seja o final da cadeia, um construtor vazio pode ser invocado, o qual irá configurar um **Null Object** no atributo **proximo**. O método abstrato **processar()** representa a lógica de processamento específica daquela implementação que será invocada durante a execução da cadeia, e, por ele não poder ser invocado de fora, sua visibilidade é **protected**. O método **processarCadeia()** é o responsável pela execução da cadeia e invoca não somente o método **processar()** da própria classe, quanto o método **processarCadeia()** do atributo **proximo**.

Diferentemente da implementação do Composite, essa refatoração tem um pequeno impacto nas outras classes envolvidas. Por exemplo, as classes que invocam pós-processadores precisarão alterar o método que está sendo chamado de `processar()` para `processarCadeia()`. Esse método poderia ser mantido, porém as custas de uma mudança no nome do *hook method* que precisa ser criado nos pós-processadores. Ou seja, como o método disponível publicamente não seria mais o ponto de extensão, o nome de algum deles precisa ser modificado. Outras mudanças devem ser feitas nas subclasses, que precisam disponibilizar o novo construtor que recebe um `PosProcessador` como parâmetro e alterar de `implements` para `extends` a relação com `PosProcessador`, que não é mais uma interface.

## 4.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Esse capítulo apresentou conceitos a respeito da composição recursiva, onde objetos são compostos por outros que possuem uma de suas abstrações. Com a utilização do polimorfismo, é possível combinar diversas implementações de uma mesma abstração formando uma estrutura flexível e complexa. É possível estender o comportamento dessa estrutura a partir da criação de novas classes e sua inclusão em um de seus pontos. A própria organização das classes não deixa de ser um ponto onde o comportamento pode ser adaptado e configurado.

O padrão Composite utiliza esse conceito para permitir a representação de objetos compostos, em outras palavras, objetos que combinam a funcionalidade de outros do mesmo tipo. Com esse padrão, uma das subclasses é quem possui a composição recursiva. Nesse padrão as classes se organizam em forma de árvore, sendo as classes compostas os nós internos e as outras classes as folhas.

Já o Chain of Responsibility cria uma estrutura parecida com uma lista ligada, onde cada elemento possui uma referência para o próximo. A composição recursiva normalmente é implementada na superclasse, pois todo tipo de elemento deve poder ter um próximo. A partir desse encadeamento de classes, é possível configurar quais elementos devem ser incluídos no processamento.

A utilização desse tipo de estrutura normalmente é ligada a necessidade de flexibilidade e configurabilidade da lógica da aplicação. A recursividade da solução permite o desacoplamento entre os elementos e a criação de diversas possibilidades de configuração.

## CAPÍTULO 5

# Envolvendo Objetos

*“Não existe colher. Você verá que não é a colher que se dobra, apenas você mesmo.”*

– Garoto com a colher, Matrix

O encapsulamento é uma característica que faz com que os clientes das classes precisem conhecer apenas a sua interface, abstraindo sua implementação interna. O polimorfismo permite que a instância de uma classe possa ser atribuída a uma variável, desde que obedeça a abstração utilizada como tipo. Essas duas características fazem com que exista uma ignorância, no sentido correto da palavra, da classe cliente em relação a classe que ela realmente está lidando.

A partir desses conceitos, é possível que uma classe envolva uma instância sem que a classe cliente saiba que está lidando com outro objeto. Dessa forma, essa classe que envolve irá servir como intermediária entre a classe cliente e a classe alvo. Com isso, ela obtém o controle da execução antes e depois de cada invocação de método. Isso permite que, por exemplo, funcionalidades sejam adicionadas e que validações sejam realizadas. Pode-se inclusive interceptar a execução de um método, retornando um valor ou lançando um erro mesmo sem invocar o método original.

Esse capítulo apresenta os padrões e conceitos relacionados a esse encapsulamento de objetos. Este mesmo conceito pode ser utilizado para proteger um objeto, adicionar funcionalidades e até mesmo adaptar interfaces. Será mostrado que um dos grandes poderes dessa técnica é fazer o código cliente pensar que ainda está lidando com o objeto original e fazer que esse objeto não saiba que está sendo envolvido.

## 5.1 PROXIES E DECORATORS

*“O olhar é, antes de mais nada, um intermediário que remete de mim a mim mesmo.”*  
– Sartre

Diferentemente dos capítulos anteriores, vou começar já apresentando dois padrões ao mesmo tempo. O motivo para isso é que o padrão Proxy possui uma estrutura muito parecida com o padrão Decorator, sendo que a diferença é basicamente a motivação e o contexto no qual os padrões são aplicados.

A ideia básica desses padrões é criar uma classe que envolve uma outra classe do mesmo tipo. Dessa forma, ela pode ser passada de forma transparente como se fosse a classe original para quem a irá utilizar. A partir dessa estrutura, a classe que envolve serve como intermediária, tendo como interferir antes e depois da execução dos métodos. A Figura 5.1 representa o funcionamento desses padrões.

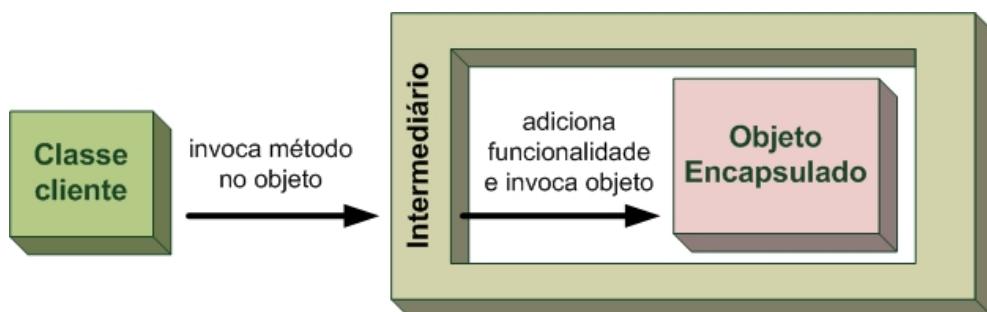


Figura 5.1: Envolvendo um objeto

A utilização desse tipo de técnica traz uma nova perspectiva para a modelagem de classes, pois ao invés de concentrar todos os aspectos de um método em apenas uma classe, é possível tratar diferentes questões em diferentes camadas. Assim

a funcionalidade da classe que está encapsulando pode ser utilizada para várias implementações. Para ficar mais concreto, uma validação de parâmetros que normalmente é feita dentro dos métodos, poderia ser colocada em uma classe que envolve a classe principal. Dessa forma, essa validação poderia ser reutilizada para diversas implementações.

## A Estrutura dos Padrões

A estrutura do Proxy e do Decorator utiliza a composição recursiva. Isso significa que ambos são compostos por uma classe que possui a mesma abstração que eles. A estrutura desses padrões está apresentada na Figura 5.2. Observe que são muito similares e permitem que a instância de uma classe encapsule um objeto e possa assumir o seu lugar. Dessa forma, os clientes dessa classe não terão conhecimento se estão lidando com a classe original ou com uma que a está encapsulando.

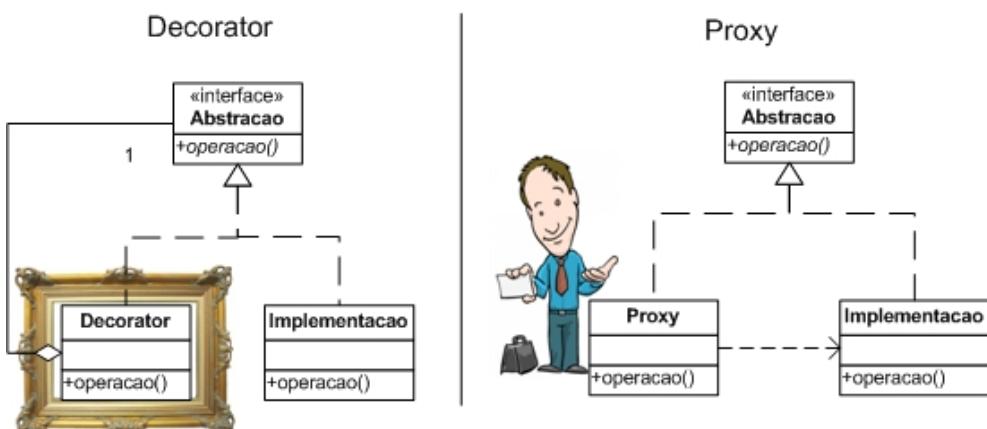


Figura 5.2: Estrutura dos padrões Proxy e Decorator

O padrão Decorator recebeu esse nome relacionado ao fato de “decorar” uma classe existente adicionando uma nova funcionalidade. Imagine como se a classe principal fosse um quadro e o Decorator fosse a moldura, que está acrescentando elementos no visual do quadro. O principal objetivo desse padrão é acrescentar funcionalidades a classes existentes de uma forma transparente a quem as utiliza. Isso também pode ser utilizado para uma melhor distribuição de responsabilidades entre as classes, permitindo que a classe principal se foque na regra de negócio central e que outras classes que a encapsulam cuidem de outras funcionalidades.

O padrão Proxy está mais ligado a prover um objeto para servir como intermediário na comunicação com um outro principal. Um exemplo comum de utilização desse padrão é para representar localmente objetos que estão em servidores remotos. Dessa forma, o proxy encapsula a lógica de acesso remoto fazendo parecer que o objeto está sendo acessado localmente. Um outro exemplo é para a criação de objetos “caros”, ou seja, que possuem um alto custo computacional de consumo de recursos para criação. O Proxy pode servir como uma representação para esse objeto, permitindo que o mesmo seja criado somente quando for necessário.

Uma coisa interessante da estrutura de ambos os padrões é sua composição recursiva que permite que seja feito um encadeamento de classes, de forma similar ao *Chain of Responsibility*. Sendo assim, um Proxy ou Decorator pode não sómente encapsular a classe original, mas também encapsular uma classe que já esteja encapsulada com um outro Proxy ou Decorator. Dessa forma, a classe fica com diversas camadas, cada uma com uma responsabilidade diferente, encapsulando a regra de negócio principal.

A principal diferença entre os dois padrões está no objetivo de cada um. Enquanto o Decorator visa adicionar novas funcionalidades, o Proxy serve mais como uma proteção ao objeto principal. Em termos de estrutura, o Decorator costuma permitir que o objeto encapsulado seja passado no construtor, ou configurado de alguma outra forma. Dessa forma, a composição é feita utilizando a abstração, permitindo que qualquer implementação seja encapsulada. Já o Proxy, muitas vezes, protege um objeto específico e a criação do objeto encapsulado ocorre dentro dele. No caso de um Proxy que faça acesso remoto, ele nem compõe a classe propriamente dita, mas utiliza as classes de acesso a rede para delegar as chamadas.

Porém essa diferença estrutural é muito sutil e pode até mesmo ser considerada como um detalhe de implementação. Em relação aos objetivos, a proteção ao acesso feita pelo Proxy não deixa de ser uma funcionalidade como no Decorator. Por esse motivo, daqui para frente nesse livro o padrão Proxy será citado referenciando ambos os padrões.

## **PARA QUE EU PRECISO DE UMA INTERFACE SE NUNCA VOU MUDAR ESSA IMPLEMENTAÇÃO?**

Muita vezes vejo desenvolvedores pensando duas vezes antes de criar uma abstração, no formato de uma interface, de algum componente importante do sistema. Por exemplo, vamos imaginar uma classe que retorne propriedades de um arquivo de configuração do sistema. Por mais que não exista a possibilidade de se criar uma nova implementação para essa classe, pode ser interessante que essa classe possua uma abstração.

Um dos motivos para isso é a possibilidade de se criar um Proxy que irá envolver essa implementação. Nesse exemplo da leitura do arquivo, poderiam haver Proxies para funcionalidades como o armazenamento dos valores em memória e para a verificação da consistência desses valores. Sendo assim, antes de pensar só vai haver uma classe, pense que a abstração também pode ser utilizada na criação de Proxies que podem encapsular a classe principal.

Se você por acaso não criou uma interface para uma classe que precisa de um Proxy, lembre-se que não é difícil refatorá-la com as refatorações automatizadas disponíveis nos IDEs. No Eclipse, por exemplo, ao extrair a interface, existe a opção para utilizá-la ao invés do tipo da classe onde for possível. Sendo assim, mais importante que criar a interface é obedecer o encapsulamento, evitando depender de detalhes internos da classe.

## **Cenários para Aplicação de Proxies**

A utilização de Proxies é um recurso tão poderoso que me arriscaria a dizer que é o padrão que mais utilizo. Antes de entrar em exemplos mais detalhados, acho importante mostrar alguns cenários onde a utilização dessa técnica pode ser feita. Acredito que isso ajuda a enxergar o potencial desses padrões.

Muitas vezes é necessária a criação de código de proteção para evitar que seja feita uma invocação de método com parâmetros inadequados ou em um contexto errado. Exemplos dessas situações seriam parâmetros que precisam estar em um formato específico ou métodos que só podem ser invocados em determinados esta-

dos da classe. Outro cenário onde é importante proteger a classe é para requisitos de segurança, evitando que uma funcionalidade seja acessada por um usuário não-autorizado ou bloqueando parâmetros maliciosos que tentam executar ataques de injeção (ver quadro). Para todos esses casos, para evitar que esse código de proteção fique misturado com o código do negócio, muitas vezes é interessante criar um proxy exclusivamente para bloquear esses acessos indevidos.

### ATAQUES DE INJEÇÃO

Um ataque de injeção é aquele que se aproveita de aplicações que se utilizam da concatenação de strings para a formação de um comando que será executado. Dessa forma, o atacante procura enviar um parâmetro malicioso para a aplicação visando a formação de um comando diferente durante a concatenação. Um dos exemplos mais famosos é o SQL Injection, onde procura-se injetar partes de comandos SQL em uma string que formará uma consulta que será executada no banco de dados. Outro exemplo de ataque é o *Cross-site Scripting*, que faz o uso de injeção de javascript em parâmetros que são utilizados para a construção de páginas web.

Um outro cenário onde o uso de Proxies é bastante comum é para fazer cache da execução de métodos. Quando uma funcionalidade demorada precisa ser executada de forma repetida em uma aplicação, é interessante armazenar o resultado obtido na primeira execução e reutilizá-lo em novas invocações. Colocar isso junto com a própria funcionalidade pode tornar o código confuso, misturando duas responsabilidades no mesmo local. Dessa forma, um Proxy pode ser utilizado para encapsular a classe, armazenando o resultado da primeira execução e retornando-o sem executar a classe novamente nas próximas invocações.

Outra utilização clássica desse tipo de técnica é para tornar transparente o acesso a um sistema remoto. A um certo tempo atrás, a comunicação entre aplicações remotas precisava ser implementada através do acesso a bibliotecas de rede, sendo uma atividade difícil e trabalhosa. Hoje, grande parte das tecnologias para comunicação remota, como RMI, EJB ou mesmo classes clientes de webservices, encapsula todo esse processo dentro de um Proxy, o qual é responsável por acessar remotamente um componente disponibilizado pelo servidor. Esse Proxy implementa a mesma

interface do objeto original, permitindo uma transparência desse processo para a aplicação, que pode acessar de forma indiferente o objeto local ou o remoto.

Em geral, esses padrões são utilizados para funcionalidades que são ortogonais a estrutura do software. Isso significa que elas cortam a estrutura do software estando presentes em diversas classes, não estando ligadas diretamente a funcionalidade da classe. Por exemplo, uma funcionalidade de registro de auditoria ou de controle de acesso costuma ser necessária em diversas classes de um mesmo tipo. A utilização de um Proxy permitiria, não somente a reutilização dessa funcionalidade em diversas implementações de uma abstração, como também permitiria que essa característica fosse modularizada, evitando seu espalhamento em diversas classes.

## Proxies na API do Java

As APIs da linguagem Java estão repletas de exemplos de implementação dos padrões Proxy e Decorator. Como todo bom encapsulamento, se você já utilizou algum deles, provavelmente não percebeu a sua presença. Essa seção trás alguns exemplos de uso desses padrões, que podem servir como uma referência para o seu uso.

O primeiro exemplo está na API de coleções. Essa API possui algumas classes que servem para “decorar” as coleções existentes adicionando funcionalidades a elas. A classe `Collections` possui métodos estáticos como `synchronizedList(List<T> list)` que retorna a lista recebida com sincronização e `unmodifiableList(List<? extends T> list)` para criar uma lista não-modificável. O que na verdade esses métodos fazem é encapsular a lista passada em um Proxy que adiciona essa característica ao objeto retornado.

Outro uso importante do padrão Proxy acontece no RMI (Remote Method Invocation), que é a tecnologia utilizada em Java para a invocação remota de métodos. Nesse modelo, após a criação da classe que disponibilizará seus métodos remotamente, são gerados os chamados *stubs* e *skeletons*. Os *skeletons* são classes que recebem no servidor as requisições e invocam a classe remota. Já os *stubs* são *Proxies* que ficam no cliente e representam a classe remota, possuindo a mesma interface que ela.

A API padrão para o mapeamento objeto-relacional no Java EE, o JPA (Java Persistence API), possui uma funcionalidade muito interessante que permite um carregamento preguiçoso de listas associadas a classe principal. Isso significa que se eu tenho uma classe `Pessoa` que possui uma lista de instâncias de `Telefone` que são persistentes em uma base relacional, é possível que ao carregar uma `Pessoa`, sua lista de

Telefone seja carregada somente a primeira vez que ela é acessada. Isso é feito a partir de um Proxy que acessa a base de dados e popula a lista somente em seu primeiro acesso.

### PROXIES DINÂMICOS

Quando criamos uma classe para ser um Proxy, ela precisa implementar a mesma abstração da classe que está encapsulando para que possa ser passada no lugar dela. Em Java é possível criar os chamados **Proxies Dinâmicos**. Eles tem a capacidade de assumir dinamicamente uma interface, permitindo que seja utilizado para diversas interfaces. Esse é um recurso avançado da API de reflexão que foge ao escopo desse livro, porém acho importante citar para que o leitor saiba da existência desse tipo de recurso.

## 5.2 EXEMPLOS DE PROXIES

*“Falar é fácil. Mostre-me o código.”*

– Linus Torvalds

Nesse capítulo, um longo caminho foi percorrido sem nem um exemplo de código. Para não decepcionar os leitores, introduzo esta seção dedicada a apresentar alguns exemplos de Proxy. A ideia é mostrar exemplos realistas que possam ser adaptados para outras aplicações. Cada subseção irá apresentar um contexto, a interface utilizada pelas classes e em seguida o código do Proxy.

### Invocações Assíncronas

Muitas vezes em alguns sistemas temos funcionalidades que demoram a ser executadas. Nesses casos, é ruim deixar o usuário esperando, sendo que em alguns casos ele pode inclusive achar que o sistema travou. O que costuma acontecer nesses casos são essas funcionalidades serem executadas em uma thread diferente. Abaixo segue a listagem da interface que abstrai os tipos de transação do sistema. Para as implementações, as informações da transação são armazenadas em atributos e normalmente passadas no construtor. Para mais detalhes sobre essa construção, ver o Capítulo ?? sobre o padrão Command.

```
public interface Transacao {  
    public void executar();  
}
```

Talvez o primeiro impulso fosse adicionar a criação de novas threads nas próprias implementações de `Transacao`. Porém isso iria gerar uma duplicação de código entre as classes que precisariam disso. Isso também causaria problemas se uma mesma transação precisasse ser executada na mesma thread ou em uma thread diferente dependendo do contexto. Nesse caso, a criação da thread na chamada da transação poderia resolver esse problema, mas não resolveria o problema da duplicação.

Uma forma de abordar esse problema seria a criação de um `Proxy` que encapsulasse uma transação e a invocasse de forma assíncrona. A classe `ProxyAssincrono` apresentada a seguir implementa essa funcionalidade. Ele pode ser utilizado em qualquer implementação de `Transacao`, evitando a duplicação, e inclusive a mesma classe pode ser utilizada com ou sem o `Proxy`.

```
public class ProxyAssincrono implements Transacao{  
  
    private Transacao t;  
  
    public ProxyAssincrono(Transacao t) {  
        this.t = t;  
    }  
    public void executar() {  
        Runnable r = new Runnable(){  
            public void run() {  
                t.executar();  
            }  
        };  
        Thread t = new Thread(r);  
        t.start();  
    }  
}
```

## Realizando o Cache de um DAO

Data Access Object, também conhecido como DAO, é um padrão com o objetivo de isolar o acesso a dados de uma aplicação [?]. A partir desse padrão, define-se uma interface para o acesso a dados e uma implementação para encapsular todas as responsabilidades referentes a interface com um banco de dados. Apesar de ser

um padrão fora do escopo desse livro, ele descreve uma prática bastante popular em aplicações que lidam com dados.

Esse exemplo se contextualiza em um software que possui um DAO que realiza o acesso a um banco relacional. Imagine que a interface dessa camada seja a apresentada na listagem a seguir. Normalmente um DAO costuma possuir mais métodos, porém nesse exemplo vamos considerar apenas três métodos respectivamente para recuperação, exclusão e gravação na base de dados. Imagine que a aplicação esteja tendo problemas de desempenho e tentando diminuir a quantidade de acessos a banco. Uma forma de fazer isso é guardando alguns objetos em memória e reaproveitá-los em leituras subsequentes. Como fazer isso com o menor impacto possível?

```
public interface DAO {  
    public Identificavel recuperar(int id);  
    public void excluir(int id);  
    public void salvar(Identificavel obj);  
}
```

Para não afetar nem os clientes nem os próprios DAOs, a solução seria colocar um Proxy encapsulando o DAO original para adicionar a funcionalidade de cache. A seguir é apresentada a implementação da classe CacheDAO. Ele possui um mapa que é utilizado para armazenar os objetos persistidos a partir do seu identificador. Os métodos `salvar()` e `excluir()` respectivamente adicionam e removem os objetos do mapa, mas apenas após delegar a chamada a classe que está encapsulando.

```
public class CacheDAO implements DAO {  
  
    private DAO dao;  
    private Map<Integer, Identificavel> cache;  
  
    public CacheDAO(DAO dao) {  
        this.dao = dao;  
        this.cache = new HashMap<>();  
    }  
    public Identificavel recuperar(int id) {  
        if(cache.containsKey(id))  
            return cache.get(id);  
        Identificavel obj = dao.recuperar(id);  
        cache.put(id, obj);  
        return obj;  
    }  
}
```

```
    }
    public void excluir(int id) {
        dao.excluir(id);
        cache.remove(id);
    }
    public void salvar(Identificavel obj) {
        dao.salvar(obj);
        cache.put(obj.getId(), obj);
    }
}
```

Dessa forma, o método `recuperar()` primeiro verifica se o objeto está no cache e em caso positivo retorna ele sem invocar o DAO encapsulado. Caso o objeto não esteja no cache, o DAO é então invocado e o objeto encapsulado é armazenado no cache antes de ser retornado. Observe que nesse exemplo, o Proxy pode em alguns casos retornar para o cliente sem nem invocar o método do objeto encapsulado.

Vale ressaltar que o exemplo de cache apresentando nesse exemplo é bem simples e não está preparado para lidar com uma série de questões importantes. Para exemplificar podem ser citadas a sincronização para diversos acessos paralelos, a expiração devido a possibilidade de atualização dos dados por outras origens e a distribuição no caso de aplicações onde os dados são acessados de várias máquinas. Caso em sua aplicação seja necessário um cache mais elaborado, sugere-se a utilização de padrões destinados a criação de cache para acesso a dados [?]. Porém, quanto mais complexo precisar ser o cache, mais adequado é a utilização de um Proxy para isolar essa complexidade do resto da aplicação. Frameworks como o Hibernate já trabalham dessa forma e até possibilitam a escolha da sua biblioteca de cache, como o EhCache e o Infinispan.

## Tentativas Repetidas de Acesso

Outra situação que não é incomum de acontecer é ser necessário adicionar uma funcionalidade em uma classe a qual não podemos modificar. Isso normalmente acontece quando essa classe faz parte de um framework ou biblioteca de classes que está sendo utilizada. Considere a interface representada a seguir como a implementada por uma classe disponibilizada por uma empresa para buscar um arquivo em seus servidores remotos. Porém, devido a instabilidade da rede, existem muitas falhas ao tentar recuperar o arquivo.

```
public interface AcessoRemoto {
```

```
    public byte[] buscarArquivo(String url) throws IOException;
}
```

Para diminuir o número de falhas, uma alternativa é realizar tentativas repetidas de acesso ao arquivo. Como a classe não pode ser modificada e não é desejável modificar as classes que buscam o arquivo, uma alternativa é a criação de um Proxy que tente buscar o arquivo novamente no caso de uma falha. Na listagem a seguir é apresentada a classe TentativasRepetidas que encapsula uma instância do tipo AcessoRemoto. O método buscarArquivo() invoca o mesmo método no objeto que está sendo encapsulado, porém dentro de um bloco try/catch onde um possível erro é capturado incrementando o número de tentativas. Sendo assim, o método é invocado novamente até que nenhum erro seja retornado ou que o número limite de tentativas seja atingido.

```
public class TentativasRepetidas implements AcessoRemoto{

    private AcessoRemoto ar;
    private int maximoTentativas;

    public TentativasRepetidas(AcessoRemoto ar, int maximoTentativas) {
        this.ar = ar;
        this.maximoTentativas = maximoTentativas;
    }

    public byte[] buscarArquivo(String url) throws IOException{
        int tentativas = 0;
        IOException ultimoErro = null;
        while(tentativas < maximoTentativas){
            try{
                return ar.buscarArquivo(url);
            }catch(IOException ex){
                tentativas++;
                ultimoErro = ex;
            }
        }
        throw ultimoErro;
    }
}
```

### 5.3 EXTRAINDO UM PROXY

Normalmente os padrões Proxy e Decorator são aplicados quando uma funcionalidade precisa ser adicionada e deseja-se causar um impacto mínimo no código que já foi implementado. Nesse caso não é necessário refatorar a aplicação para a implementação do padrão. A necessidade de refatoração na direção de um Proxy pode ocorrer quando uma classe está inchada com muitas funcionalidades e deseja-se dividir essas responsabilidades. A refatoração para esse cenário seria extrair um Proxy com uma funcionalidade não ligada diretamente ao objetivo principal da classe. Esse procedimento poderia ser realizado diversas vezes para diferentes funcionalidades.

Um outro caso onde essa refatoração seria necessária é quando existe duplicação entre uma parte do código entre implementações de uma mesma abstração. Isso também pode ser motivado quando se perceber que em uma nova implementação será necessária parte da funcionalidade já presente em uma outra classe. Como exemplo, outro dia criei uma classe em que havia um código interno que validava se a ordem de invocação dos métodos estava correta, porém quando fui criar uma nova implementação percebi que a mesma validação seria necessária. Dessa forma, extraí um Proxy com a parte de validação da primeira classe e passei a utilizá-lo com todas as implementações.

A Figura 5.3 apresenta os passos da refatoração de extração de um Proxy de uma classe existente. O primeiro passo é criar a classe que representará o Proxy, delegando todas as chamadas dos métodos para a instância que está sendo encapsulada. Nesse ponto é importante fazer com que os testes não atuem isoladamente na classe, mas nela encapsulada pelo Proxy recém criado. Inicialmente os testes deverão passar sem problemas, pois nenhum código foi adicionado na classe intermediária.

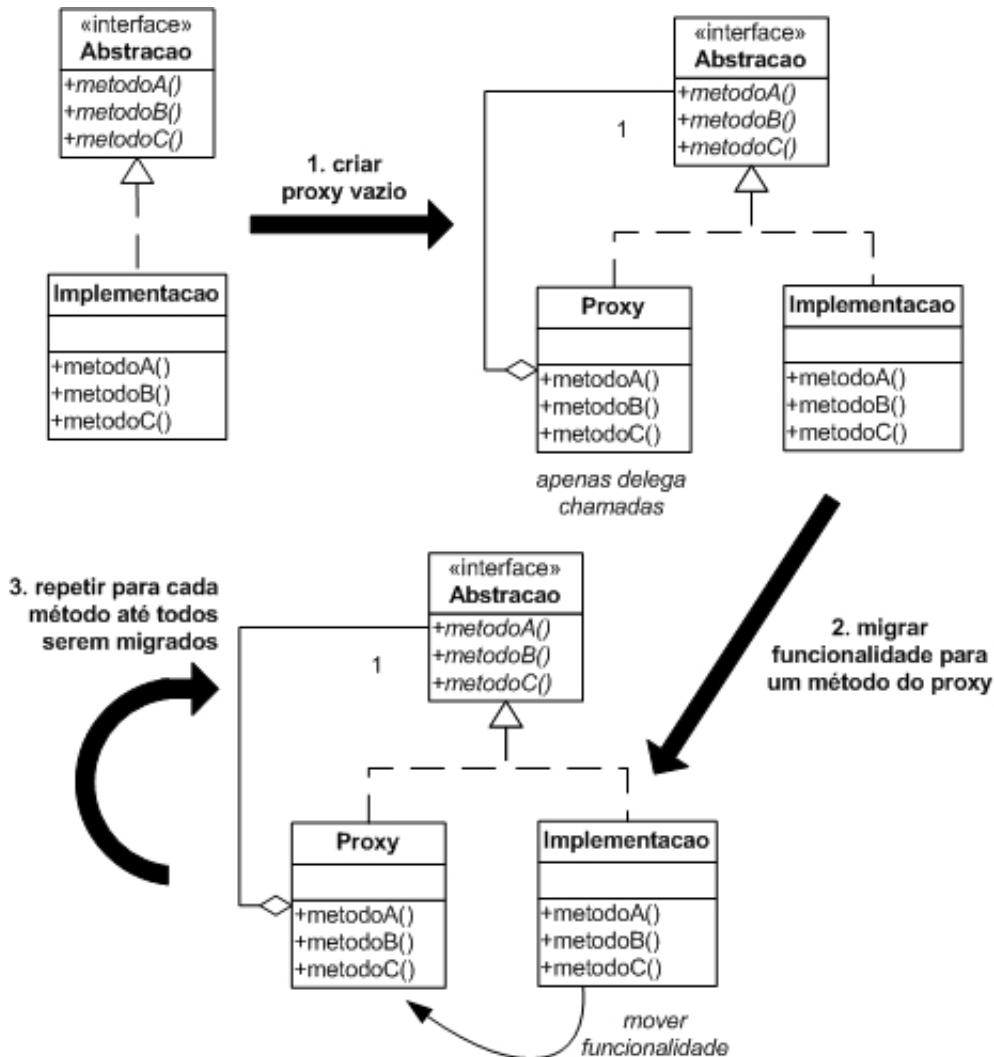


Figura 5.3: Refatoração de Extração de Proxy

Depois de possuir a suíte de testes atuando em cima da classe em conjunto com o Proxy, é hora de começar a migrar funcionalidade da classe encapsulada. A principal regra é que o comportamento conjunto deve permanecer inalterado. Dessa forma, a parte relativa a cada um dos métodos pode ir sendo movida para o intermediário, até que toda funcionalidade desejada esteja no Proxy. Depois disso, se o desenvolvedor achar necessário, ele pode separar os testes da classe dos testes relativos a

funcionalidade do Proxy, que devem funcionar com qualquer outra implementação encapsulada.

### CRIANDO UM PROXY NO ECLIPSE

A criação da versão inicial do Proxy, que simplesmente delega os métodos para classe encapsulada, pode ser um trabalho bem trabalhoso e braçal se a interface alvo possuir um número grande de métodos. Felizmente, o Eclipse possui uma funcionalidade de geração de código que cria automaticamente os métodos que delegam a chamada para os métodos de um atributo.

Para utilizar essa funcionalidade, primeiramente deve-se criar uma classe e incluir um atributo do tipo que será encapsulado. Não caia na tentação de adicionar a interface juntamente com seus métodos. Em seguida, clique com o botão direito sobre o código e escolha *Source > Generate Delegate Methods*. Nesse momento irá abrir uma janela apresentando quais métodos dos atributos você deseja criar e delegar para ele a chamada. É só escolher todos os métodos e em seguida declarar que a classe implementa a interface alvo. A Figura 5.4 mostra a caixa de diálogo aberta durante esse processo.

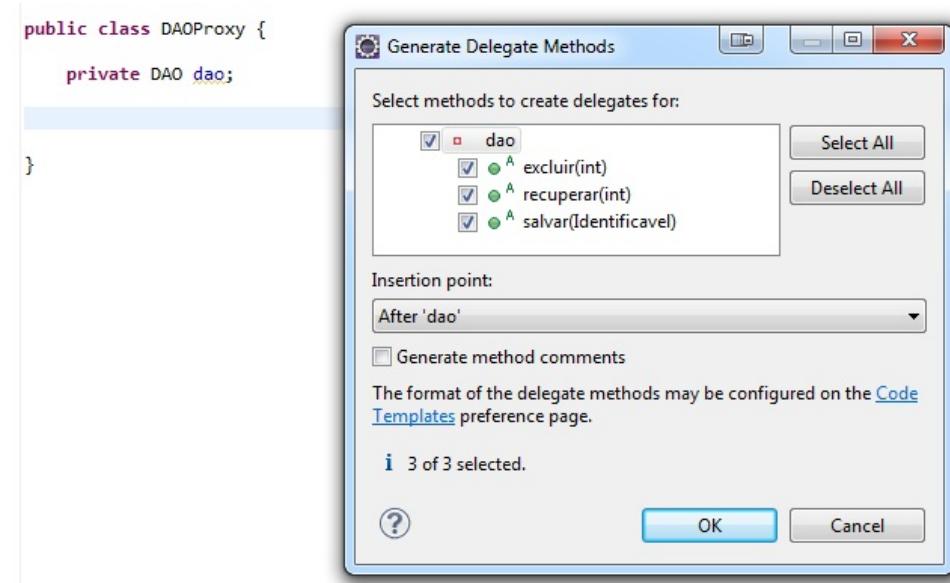


Figura 5.4: Automatizando a Delegação de Métodos no Eclipse

## 5.4 ADAPTANDO INTERFACES

*“A incapacidade de se adaptar traz a destruição.”*

– Bruce Lee

Um outro cenário onde existe a necessidade de criar uma classe que envolva outra é quando possuímos uma classe que implementa uma interface, mas precisamos que ela implemente outra. Um exemplo do mundo físico ocorre quando compramos um notebook nos Estados Unidos e precisamos ligar ele aqui nas tomadas do Brasil. No caso, o notebook americano possui uma fonte que possui a interface das tomadas americanas, porém precisamos que ele possua a interface que permita que ele seja ligado aqui no Brasil. Todos sabem que esse problema não é difícil de resolver, bastando um adaptador que encaixe em uma entrada no parão brasileiro e possua uma entrada no padrão americano. A Figura 5.5 mostra de forma visual a questão do adaptador.

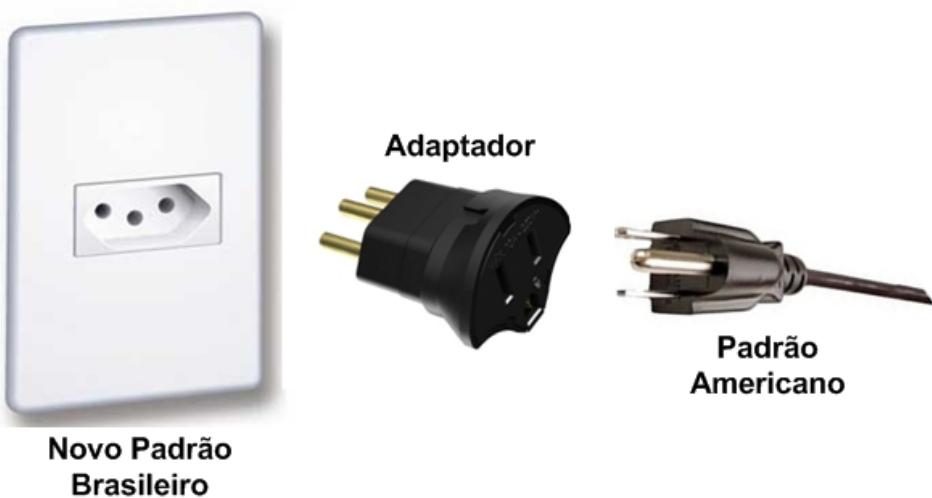


Figura 5.5: Representação do adaptador de tomadas

A ideia do padrão Adapter não é muito diferente de um adaptador de tomadas. Sua solução cria uma classe que implementa a interface requerida e que é composta por uma classe da interface da classe que possui a funcionalidade. Dessa forma, o Adapter traduz as chamadas da interface que ele implementa para a classe que ele encapsula. Isso nem sempre é trivial, pois normalmente não é somente o nome do método que muda, mas outras questões muitas vezes inesperadas.

### Estrutura do padrão Adapter

A estrutura do padrão Adapter é similar a dos padrões Proxy e Decorator, sendo que a principal diferença é que a classe que está sendo encapsulada não possui a mesma interface da que a está envolvendo. É justamente esse fato que permite a adaptação entre as interfaces. A Figura 5.6 mostra em um diagrama os principais participantes desse padrão.

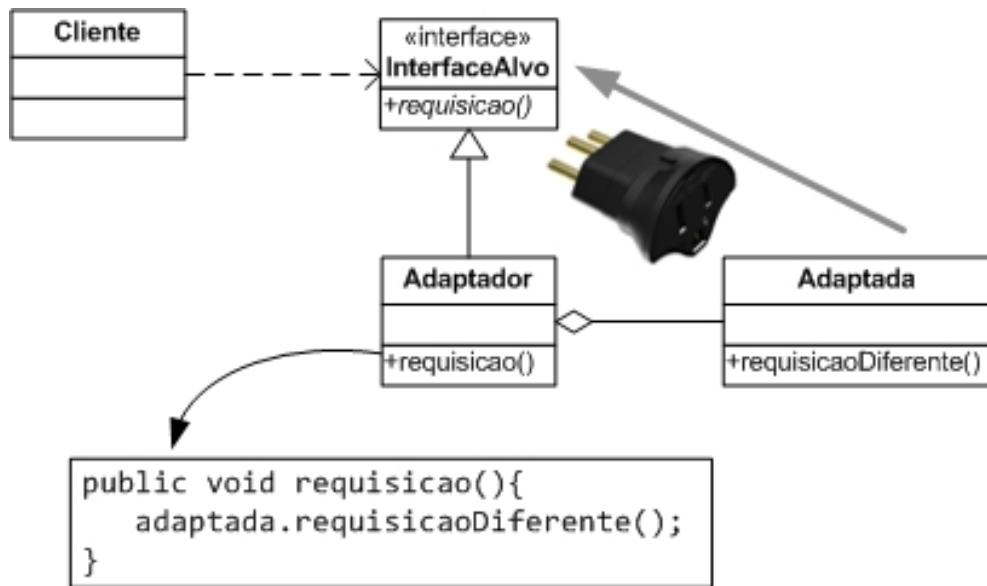


Figura 5.6: Estrutura do padrão Adapter

Nesse diagrama temos uma classe **Cliente** que depende de alguma forma de uma **InterfaceAlvo**. Sendo assim, para tornar possível que ela possa acessar uma classe **Adaptada** que não implementa essa abstração, ela precisa utilizar a classe **Adaptador**. Essa classe irá receber chamadas de método conforme a **InterfaceAlvo** e irá chamar os métodos correspondentes na classe **Adaptada**.

É importante ficar claro que nem sempre a adaptação é simples de ser feita. Questões comuns que precisam ser traduzidas incluem a nomenclatura de classes e métodos, além da conversão de parâmetros e retornos. Questões mais complexas podem envolver diferentes formas de interagir com as classes. Por exemplo, o que é um parâmetro de método em uma pode ser um atributo passado no construtor na outra, ou ainda, o que é o retorno em uma pode ser um atributo populado em um parâmetro passado para o método na outra. Porém, a maior dificuldade é quando existem diferenças semânticas entre as diferentes APIs. Nesse caso, normalmente é necessária a ajuda de especialistas do domínio para compreender as diferenças e possibilitar a tradução.

## Exemplo de Adaptador

Imagine uma aplicação que interaja com o serviço de SMS de operadoras de ce-

lular. Considere que a versão inicial da aplicação possua uma interface para interagir com o serviço de apenas uma operadora. A listagem a seguir apresenta a interface SMSender que é utilizada pela aplicação para o envio de mensagens. Essa classe possui o método sendSMS() que recebe como parâmetro uma instância da classe SMS, também apresentada na listagem, e retorna um valor booleano dizendo se a mensagem foi enviada com sucesso ou não.

```
public interface SMSender {  
    public boolean sendSMS(SMS sms);  
}  
  
public class SMS {  
    private String destino;  
    private String origem;  
    private String texto;  
    //getters e setters omitidos  
}
```

Ao evoluir a aplicação, foi necessário incorporar o serviço de envio de SMS de uma outra operadora. Não foi uma surpresa muito grande quando foi constatado que a API para o acesso a funcionalidade era completamente diferente. A listagem com a interface dessa nova API, EnviadorSMS, está representada a seguir. A primeira diferença está nos parâmetros, que ao invés de serem encapsulados em uma classe, no caso a SMS, são passado diretamente para o método. Outra diferença é que o texto da mensagem não é mais uma string, mas um array de strings. A primeira API recebia um texto longo e dividia em várias mensagens se necessário, porém nessa outra API a mensagem precisam ser divida em trechos de 160 caracteres antes da chamada do método. Finalmente, a nova API lança uma exceção para indicar uma falha, e não retorna um valor booleano para indicar o sucesso ou não.

```
public interface EnviadorSMS {  
    public void enviarSMS(String destino, String origem, String[] msgs)  
        throws SMSException;  
}
```

Para evitar que as classes cliente precisem ser alteradas e se acoparem a uma nova API, decidiu-se criar um Adapter para traduzir as chamadas de uma interface para outra. Para criar esse adaptador é preciso lidar com todas as questões relativas as diferenças entre as interfaces. A implementação do adaptador está representada na listagem a seguir.

```
public class SMSAdapter implements SMSSender {  
  
    private EnviadorSMS env;  
  
    public SMSAdapter(EnviadorSMS env) {  
        this.env = env;  
    }  
    public boolean sendSMS(SMS sms) {  
        String dest = sms.getDestino();  
        String orig = sms.getOrigem();  
        String[] txts = quebrarMsgs(sms.getTexto());  
        try {  
            env.enviarSMS(dest, orig, txts);  
        } catch (SMSException e) {  
            return false;  
        }  
        return true;  
    }  
    private String[] quebrarMsgs(String text){  
        int size = text.length();  
        int qtd = (size%160==0)? size/160: size/160+1;  
        String[] msgs = new String[qtd];  
        for(int i=0; i<qtd; i++){  
            int min = i*160;  
            int max = (i==qtd-1)? size: (i+1)*160;  
            msgs[i] = text.substring(min,max);  
        }  
        return msgs;  
    }  
}
```

O método `sendSMS()` inicialmente recupera as informações do objeto `SMS` e as atribui a variáveis locais. O método `quebrarMsgs()` é chamado para quebrar o texto da mensagem que está em apenas uma string em um array strings com tamanho máximo de 160 caracteres. Para lidar com a diferença do retorno, a chamada ao método `enviarSMS()` da classe `EnviadorSMS` é encapsulada em um bloco `try/catch`. Dessa forma, caso a exceção for capturada o método retorna `false`, e caso o método siga seu fluxo normal, é retornado `true`.

Através desse exemplo, é possível observar como existem vários fatores que devem ser levados em consideração na hora da adaptação. Muitas vezes pode-se perder

funcionalidade e informações. Em um caso oposto, podem ser fornecidas informações incompletas ou haverem funcionalidades da interface que não podem ser executadas. Na classe `SMSAdapter`, por exemplo, para transformar a exceção recebida em um retorno booleano, perdeu-se a mensagem com a causa do erro e outras informações associadas a exceção.

### ADAPTANDO EXCEÇÕES

Quando a exceção lançada por um método precisa ser transformada na exceção lançada pela interface da classe adaptadora, a solução mais comum é realizar a chamada do método encapsulada em um bloco `try`, e lançar a nova exceção dentro do bloco `catch`. Caso a nova exceção seja criada sem relação com a anterior, o erro lançado possuirá em seu *stacktrace* o método do `Adapter` como origem, omitindo a real causa do erro. O segredo nesse caso, é não esquecer de passar a exceção original no construtor da nova exceção, dessa forma o *stacktrace* anterior será incluído como causa do erro.

## Adaptadores para Migração de APIs

Um outro cenário onde o padrão `Adapter` costuma ser bastante utilizado é para dar suporte a classes que utilizaram uma versão antiga de uma API em uma versão nova do software. Imagine, por exemplo, que em uma aplicação fossem criadas classes que implementassem a interface de um framework para serem utilizadas por ele. Caso o framework modifique essa interface em sua próxima versão, a aplicação não poderá utilizar as classes já criadas. Isso é um grande problema, visto que a aplicação pode nem compilar com a nova versão do framework.

Como uma forma de contornar esse problema, o framework pode prover um adaptador que adapte as classes desenvolvidas na versão anterior da interface para a versão nova. Dessa forma, as classes da aplicação podem utilizar esse adaptador para continuar utilizando as classes já desenvolvidas.

Como um exemplo disso, a JDK 1.0 possuía a interface `Enumeration`, com os métodos `hasMoreElements()` e `nextElement()`, e era implementada por classes que desejavam permitir a iteração de seus elementos. Na JDK 1.2 foi introduzida a interface `Iterator`, com o mesmo objetivo, porém com nomes de métodos menores, `hasNext()` e `next()`, e com um método opcional `remove()`.

Para permitir que códigos que trabalhem com uma interface possam trabalhar com classes que sabem trabalhar com a outra, a biblioteca de classes *Apache Commons Collections* provê a classe `IteratorUtils`. Esse classe possui os métodos estáticos `asIterator()`, que adapta um `Enumeration` para um `Iterator`, e `asEnumeration()`, que faz a adaptação inversa. Dessa forma, se um código que utiliza `Enumeration` precisa utilizar uma classe que implementa `Iterator`, ele pode utilizar o `IteratorUtils` para fazer essa adaptação.

## 5.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Este capítulo abordou padrões que definem classes que envolvem objetos, servindo como intermediárias entre a classe que fornece a funcionalidade e a classe cliente. Com o uso do encapsulamento e do polimorfismo, é possível tornar essa classe intermediária transparente, permitindo facilmente a adição de uma nova camada entre as classes. Esse recurso poderoso permite a utilização de diversos artifícios escondidos sob a máscara da interface da qual a classe cliente depende.

Na primeira parte do capítulo, foram abordados os padrões `Proxy` e `Decorator`, onde essa classe intermediária possui a mesma abstração da classe que a está envolvendo. Dessa forma, ela pode adicionar novas funcionalidades e se passar pela própria classe. Já na segunda parte, foi abordado o padrão `Adapter` onde a classe intermediária é utilizada para encapsular uma classe com uma interface diferente da sua. Nesse caso, a ideia é justamente permitir que uma classe com uma determinada interface possa ser utilizada por uma classe que sabe interagir com classes com uma interface diferente.

## CAPÍTULO 6

# Estratégias de Criação de Objetos

*“A verdadeira felicidade vem da alegria de atos bem feitos, o entusiasmo de criar coisas novas.”*

– Antoine de Saint-Exupery

Nos capítulos anteriores, foram apresentados diversos padrões que montam estruturas utilizadas para fornecer flexibilidade e extensibilidade nas classes da aplicação. Quando utilizamos herança, devemos escolher entre diferentes implementações de uma abstração para que ela tenha o comportamento adequado. Com composição, outros objetos devem ser instanciados e inseridos na classe principal. Com a composição recursiva, diversas instâncias de implementações de uma abstração são combinadas em uma estrutura que fornece o comportamento desejada. Finalmente, para a utilização de Proxies, é preciso envolver o objeto e colocá-lo no lugar do encapsulado.

Para os padrões funcionarem é preciso que essas estruturas estejam montadas, porém saber como criá-las de forma adequada não é tão simples. Por exemplo, de nada adianta a classe utilizar as abstrações para se desacoplar das implementações,

se na hora de criar os objetos ela referencia diretamente as classes. Os objetos devem estar desacoplados também no momento da criação! Não somente a utilização de uma estrutura de herança ou composição deve ser transparente a classe cliente, mas também a criação dessa estrutura. Para que realmente exista flexibilidade, deve ser possível criar as diferentes implementações ou combinações de classe para que o comportamento possa ser alterado.

Esse capítulo irá abordar os padrões para a criação de objetos. Esses padrões mostram técnicas para o desacoplamento da lógica de criação de objetos, principalmente quando precisa-se utilizar um dos padrões apresentados nos capítulos anteriores. Também serão apresentadas soluções para encapsular uma lógica complexa de criação e para vincular a criação de objetos relacionados. A próxima seção começa o capítulo mostrando porquê os construtores, a forma tradicional de criar objetos, podem não ser suficientes para encapsular essa lógica de criação.

## 6.1 LIMITAÇÕES DOS CONSTRUTORES

Se você perguntar para qualquer desenvolvedor Java como se cria um objeto, ele irá prontamente responder que é através de construtores. Apesar de realmente não ser possível criar uma nova instância de uma classe sem a invocação de um construtor, sua aplicação não precisa interagir diretamente com eles para criar os objetos de que precisa. Principalmente quando se deseja desacoplar a classe cliente da classe que está sendo criada. Se o construtor for diretamente invocado, o código da classe cliente precisará ser mudado para substituí-la por uma nova implementação.

Esta seção mostra algumas limitações dos construtores que dificultam sua utilização para a criação de objetos. Algumas dessas questões são abordadas no famoso livro *Effective Java* de Joshua Bloch [?].

### Dois Construtores Não Podem Ter Parâmetros de Mesmo Tipo

Uma característica dos construtores é que eles possuem o mesmo nome da classe que criam. Isso não é um problema se a classe possui um processo de criação simples, fornecendo um ou dois construtores para sua criação. Porém, quando existem diversas formas de criar objetos de uma mesma classe, isso pode ser um problema. Para começar, toda a expressividade que é possível através da utilização de nomes descritivos para métodos não é possível ser utilizada nos construtores, visto que eles precisam ter o mesmo nome da classe. Isso dificulta bastante quando uma classe possui diversos construtores e não se sabe exatamente qual o comportamento que

será fornecido por cada um deles.

Essa questão fica ainda mais crítica quando é necessária a criação de construtores que possuem parâmetros de mesmo tipo. Como isso não é possível na linguagem Java, é preciso utilizar artifícios para diferenciar os construtores. Por exemplo, adicionar um parâmetro diferente a mais para diferenciá-los, ou ainda criar apenas um construtor com um parâmetro dizendo qual o tipo de criação que se deseja.

Para ficar mais concreta essa situação, imagine que uma classe chamada `CoordenadaGeografica` possa receber strings com representações textuais das coordenadas em seu construtor. Como existem diversos formatos para coordenada, como o **Geodésico** (*Lat 021°30.4423' S ; Long 055°09.6734' W*) e o **Geodésico Decimal** (*Lat -21.5070334899 ; Long -55.4119080998*), a intenção seria ter construtores separados para cada tipo de formato. Infelizmente, como ambos construtores teriam o mesmo nome e receberiam uma string como parâmetro, essa solução não seria possível de ser implementada.

## Um Construtor Sempre Cria um Novo Objeto

Uma outra característica de um construtor é que todo vez que ele é chamado, ele retorna um novo objeto da classe. Certamente é isso mesmo que um construtor deveria fazer, porém muitas vezes deseja-se utilizar um objeto que já existe e não gerar um novo objeto. Se esse for o caso, fica inviabilizada a criação desses objetos através de construtores.

Objetos que não contenham estado, ou seja, contenham basicamente métodos com lógica de negócio, são bons candidatos para terem um número limitado de instâncias. Outras vezes, deseja-se manter uma única entidade para representar uma determinada entidade em todo sistema. Imagine que em um sistema de locadora de carros deseja-se ter uma única instância para representar a instância de `TipoCarro` que representa um *Corsa*. Dessa forma, quando for necessário representar um *Corsa* ao invés de se criar uma nova instância, a instância existente deve ser retornada. Nada disso pode ser feito utilizando construtores diretamente.

## Um Construtor Só Pode Retornar Objetos da Mesma Classe

Talvez o principal problema dos construtores seja retornar objetos apenas da classe que definiu aquele construtor. Não é possível, por exemplo, retornar uma instância de um subclasse ou envolver o objeto em um *Proxy*. Dessa forma, a classe que invoca o construtor de uma outra se acopla a ela fortemente. Por mais que ela utilize

a abstração em outras partes do código, será necessário uma alteração no código de criação caso uma outra classe precise ser utilizada.

Para entender como isso pode ser um problema, imagine uma classe que é utilizada em vários pontos do sistema. Suponha agora que ela possuísse um grande método que fosse refatorado para a utilização do padrão *Template Method*, onde subclasses implementariam partes do algoritmo. Uma situação similar seria se fosse criado um *Decorator* para adicionar alguma nova funcionalidade a essa classe. Se ela fosse utilizada em vários pontos diferentes do sistema, o código de criação deveria ser modificado em todos esses pontos para que a subclasse adequada fosse criada ou que o *Decorator* fosse criado para encapsular a instância.

Sendo assim, é possível perceber que a utilização direta de construtores pode dificultar bastante a utilização dos padrões que foram vistos até o momento nesse livro. A lógica de criação pode se complicar principalmente quando alguns padrões são combinados e surgem questões como: escolher a subclasse correta para instanciar; escolher as implementações para compor a instância; e adicionar funcionalidades envolvendo o objeto em *Proxies*. As próximas seções apresentam alguns padrões que podem ser utilizados para eliminar essas limitações e encapsular uma lógica complexa de criação de objetos.

## 6.2 CRIANDO OBJETOS COM MÉTODOS ESTÁTICOS

*“Se permaneces estático na derrota, mova-se rumo à vitória.”*

– Edilson Sanches Pontes

A solução mais simples para eliminar a necessidade da utilização de construtores diretamente é a criação de métodos estáticos para criarem os objetos. Essa prática é conhecida como *Static Factory Method*. Apesar de ser considerado um padrão, essa prática não foi devidamente documentada como um padrão e é abordada com mais detalhes no *Effective Java* [?].

Nesse padrão, as classes clientes delegam para os métodos estáticos a lógica de criação das instâncias. Esses métodos recebem os parâmetros necessários e estes retornam a instância adequada. Como esses métodos de criação não possuem as restrições dos construtores, eles podem resolver as limitações descritas na seção anterior.

Um dos primeiros benefícios da utilização do *Static Factory Method* está na expressividade, pois é possível nomear o método de acordo com a lógica de criação envolvida. Isso acaba resolvendo o problema dos construto-

res com tipos de parâmetro repetidos, pois é só criar nome diferentes para eles. No exemplo da classe `CoordenadaGeografica` citado anteriormente, seria possível possuir métodos de criação chamados `criarCoordenadaGeodesico()` e `criarCoordenadaGeodesicoDecimal()`. Dessa forma, o código cliente ficaria inclusive mais claro em relação a estratégia de criação que está sendo utilizada.

Outra possibilidade interessante é a de retornar um objeto já existente. Esse objeto normalmente é guardado na primeira vez que ele é criado e depois é aproveitado para as chamadas seguintes. Ele é armazenado em uma variável estática, ou em alguma coleção também em uma variável estáticas. Um exemplo clássico disso é a obtenção de conexões para um banco de dados. A classe `DriverManager`, por exemplo, disponibiliza o método estático `getConnection()` onde uma conexão é retornada. Algumas implementações criam um *pool* de conexões, onde as conexões são reaproveitada depois de serem utilizadas.

Um outro ponto interessante do `Static Factory Method` é que ele pode retornar qualquer implementação. Dessa forma, é possível introduzir novas subclasses e retorná-las nesse método de forma transparente a classe cliente. Assim a classe cliente dependente realmente apenas do tipo retornado pela fábrica e não das implementações retornadas. Isso simplifica muito refatorações que criam novas classes de uma abstração, como a criação de `Template Method` e de um `Proxy`, pois ao alterar o `Static Factory Method`, todas as classes que o utilizam para a criação poderão receber uma nova implementação de forma transparente.

### STATIC FACTORY METHOD X FACTORY METHOD

Antes de ir fundo nas características do `Static Factory Method` é importante deixar bem claro que ele é diferente do padrão `Factory Method` apresentado no Capítulo 2. O padrão que está sendo descrito nessa seção consiste em um método estático utilizado para encapsular a criação e/ou obtenção de instâncias por parte da classe cliente. Já o `Factory Method` é um *hook method* definido por uma classe para delegar a criação de uma instância para suas subclasses.

## Exemplos de Static Factory Method

Para exemplificar a utilização de métodos fábrica, vamos retomar o exemplo do gerador de arquivo apresentado nos capítulos anteriores (a versão que utiliza o pa-

drão Composite). A listagem a seguir apresenta o código de uma classe que disponibiliza Static Factory Methods para a criação de geradores de arquivo. Nessa implementação optou-se por disponibilizar métodos diferentes para a criação dos geradores de arquivos em XML e de propriedades. Outra alternativa seria a passagem de um parâmetro que identificaria o formato do arquivo.

```
public abstract class FabricaGerador {  
  
    public static final String ZIP = "ZIP";  
    public static final String CRYPTO = "CRYPTO";  
  
    public static GeradorArquivo criarGeradorXML  
        (String... processadores) {  
        GeradorArquivo g = new GeradorXML();  
        g.setProcessador(criarProcessador(processadores));  
        return g;  
    }  
    public static GeradorArquivo criarGeradorPropriedades  
        (String... processadores) {  
        GeradorArquivo g = new GeradorPropriedades();  
        g.setProcessador(criarProcessador(processadores));  
        return g;  
    }  
    private static PosProcessador criarProcessador  
        (String... processadores) {  
        if(processadores.length > 1) {  
            PosProcessadorComposto pp = new PosProcessadorComposto();  
            for(String proc : processadores){  
                pp.add(criarProcessador(proc));  
            }  
            return pp;  
        } else if(processadores[0].equals(ZIP)){  
            return new Compactador();  
        } else if(processadores[0].equals(CRYPTO)){  
            return new Criptografador();  
        }  
    }  
}
```

Para os processadores, é passado um array de strings que identificam os pós-processadores em sua respectiva ordem. Ambos os métodos fábrica delegam a cria-

ção deles para o método `criarProcessador()`. Observe que caso exista mais de um, a classe `PosProcessadorComposto` que implementa o padrão `Composite` é utilizada.

O código a seguir mostra como a classe `FabricaGerador` seria utilizada na criação de um `GeradorArquivo`. Um dos métodos estáticos, como `criarGeradorXML()` ou `criarGeradorPropriedades()`, deve ser invocado passando como parâmetro de forma opcional as strings que representam os pós-processadores. Note que a classe tem contato apenas com a `FabricaGerador` e a abstração `GeradorArquivo`, sendo que o uso de todas as subclasses e pós-processadores ficam encapsulados dentro da fábrica.

```
GeradorArquivo ga = FabricaGerador.criarGeradorXML(  
    FabricaGerador.ZIP,FabricaGerador.CRYPTO);
```

A própria API do Java utiliza esse padrão em diversos pontos para encapsular a criação de objetos. Um exemplo comum são os métodos `parseInt()` e `valueOf()` da classe `Integer`. Além de serem métodos estáticos que retornam instâncias de `Integer`, eles fazem cache para que a mesma instância seja retornada caso o mesmo número seja passado. O mesmo ocorre com métodos similares em outras classes que encapsulam tipos primitivos.

## Impedindo de Invocar o Construtor

Disponibilizar um `Static Factory Method` não impede as classes clientes de invocarem o construtor da classe e ignorar toda a lógica de criação encapsulada. Para resolver essa questão, o segredo é fazer com que o construtor tenha uma visibilidade que dê acesso ao método fábrica e não dê acesso as classes clientes. Se o método estiver na mesma classe que está sendo criada, o construtor pode ser declarado como privado. Se o método fábrica estiver em uma classe diferente, então uma solução é declarar ele como protegido e colocar essa classe fábrica no mesmo pacote que as classes que são criadas por ele.

## 6.3 UM ÚNICO OBJETO DA CLASSE COM SINGLETON

“Só pode haver um!”

– Connor MacLeod, Highlander

Um caso especial da utilização do `Static Factory Method` é quando se deseja que a aplicação possua apenas uma instância de uma determinada classe. Isso normalmente é motivado por questões de negócio, onde faz sentido apenas um objeto

daquele tipo. Por exemplo, imagine um software que represente um jogo de xadrez entre duas pessoas. Nesse contexto provavelmente fará sentido apenas um tabuleiro de jogo. A estrutura para permitir esse tipo de construção é o padrão Singleton.

A listagem a seguir apresenta um exemplo de implementação do padrão Singleton. No exemplo, o Singleton é utilizado em uma classe que representa e armazena configurações do sistema. Esse é um exemplo em que o uso desse padrão é adequada, pois só existe uma única configuração para todo o sistema e dessa forma ela pode ser facilmente obtida a partir de qualquer classe.

```
public class Configuracao {  
  
    private static Configuracao instancia;  
  
    public static Configuracao getInstancia() {  
        if(instancia == null)  
            instancia = new Configuracao();  
        return instancia;  
    }  
  
    // Construtor privado!  
    private Configuracao() {  
        //lê as configurações  
    }  
  
    // ...  
}
```

Nesse caso, o método fábrica é normalmente definido na própria classe. O artifício de definir um construtor privado é frequentemente utilizado para impedir a criação de outras instâncias da classe. Um atributo estático é definido para armazenar essa instância e um método estático é disponibilizado de forma pública para sua recuperação. Observe que na listagem de exemplo o objeto é criado no primeiro acesso ao método. Uma implementação alternativa, seria incluir essa criação em um bloco estático para que ela fosse realizada quando a classe fosse carregada pela máquina virtual.

A listagem a seguir mostra o exemplo de como uma classe pode obter a instância da classe que implementa o padrão Singleton. Observe que o processo não é muito diferente da invocação de um Static Factory method. Em qualquer local da aplicação em que o método `getInstancia()` for invocado, o mesmo objeto será

retornado.

```
Configuracao c = Configuracao.getInstancia();
```

Uma das vantagens do Singleton é a facilidade de acesso dessa instância por qualquer objeto na aplicação. De qualquer classe é possível chamar o método `getInstancia()` e obter essa instância. Isso evita, por exemplo, que essa instância única precise ser passada como parâmetro para diversos lugares. No exemplo do tabuleiro de xadrez, essa provavelmente seria uma classe que precisaria estar acessível de diversos locais de acordo com a lógica do jogo.

Uma solução utilizando um Singleton oferece uma flexibilidade muito maior do que uma solução que utiliza métodos estáticos para a execução das regras de negócio. Por ser um objeto, a instância única pode ser especializada e encapsulada por um Proxy, o que não é possível fazer com métodos estáticos. Essa questão prejudica bastante a testabilidade de classes que acessam métodos estáticos, pois não é possível substituí-los por um objeto falso com propósitos de teste, conhecido como Mock Object [?]. Uma solução que concilia a praticidade dos métodos estáticos com a flexibilidade do Singleton, seria os métodos estáticos delegarem a lógica de sua execução para os métodos do Singleton.

## O Lado Negro do Singleton

O padrão Singleton deve ser utilizado com muito cuidado e nas situações em que realmente fizer sentido ter apenas uma instância de uma classe. Muitos acham que por ser um padrão, que ele pode ser utilizado em qualquer parte do sistema. O resultado é que o Singleton acaba sendo utilizado como uma variável global da orientação a objetos, o que pode reduzir a flexibilidade da aplicação deixando sua modelagem deficiente. Por ele ser utilizado mais em situações inadequadas, muitos acabam considerando o Singleton como uma má prática. Sendo assim, toda vez que for utilizar um Singleton refleta bastante se sua utilização é mesmo necessária. Também veremos, em um capítulo posterior, que inversão de controle e injeção de dependências podem ajudar bastante nesse caso.

## 6.4 ENCAPSULANDO LÓGICA COMPLEXA DE CRIAÇÃO COM BUILDER

*“Podemos construir? Sim! Podemos sim!”*

– Bob, o Construtor

Pelo que já foi dito sobre a criação de objetos até o momento nesse capítulo, acredito que tenha sido possível perceber como a criação de determinados tipos de objeto pode ser complexa. Nesses casos, a lógica da criação, que muitas vezes precisa validar parâmetros ou buscar informações em arquivos, acaba se misturando com a lógica da própria classe. Sendo assim, mantê-las na mesma classe pode deixá-la grande e confusa, tanto com a utilização de construtores quanto com a utilização de métodos estáticos de criação.

O padrão **Builder** fornece uma solução para essa questão da criação de objetos complexos, com a definição de uma classe responsável pelo processo de criação. A partir de um **Builder** é possível seguir o mesmo processo de criação para criação de diferentes estruturas e diferentes representações. Dessa forma, a classe que invoca os métodos de um **Builder** para a criação do objeto que precisa fica desacoplado dessa lógica de criação complexa e da classe concreta que está sendo criada.

Além disso, pode ser criada uma abstração para um **Builder** de forma que diferentes implementações possam ser criadas. Dessa forma, os clientes podem guiar a criação de diferentes objetos através do **Builder**, porém sem saber a implementação que está sendo utilizada e, consequentemente, qual a classe do objeto que está sendo criado. Esse tipo de prática é muito comum em APIs que disponibilizam interfaces que são implementadas por diferentes fornecedores. Por exemplo, imagine uma API que lida com certificados digitais. O padrão **Builder** pode ser utilizado para criar o objeto que assina um arquivo digitalmente, porém cada fornecedor possuiará uma implementação diferente do **Builder** de acordo com suas classes.

A Figura 6.2 apresenta a estrutura do padrão **Builder**. No diagrama, a classe **Cliente** precisa criar uma instância da abstração **Produto** e utiliza um **Builder** para essa tarefa. Observe que pela estrutura é possível ter várias implementações do **Builder**, possibilitando que diferentes estratégias de implementação sejam utilizadas. Porém é importante ressaltar que essa abstração do **Builder** só deve ser utilizada quando realmente for necessária.

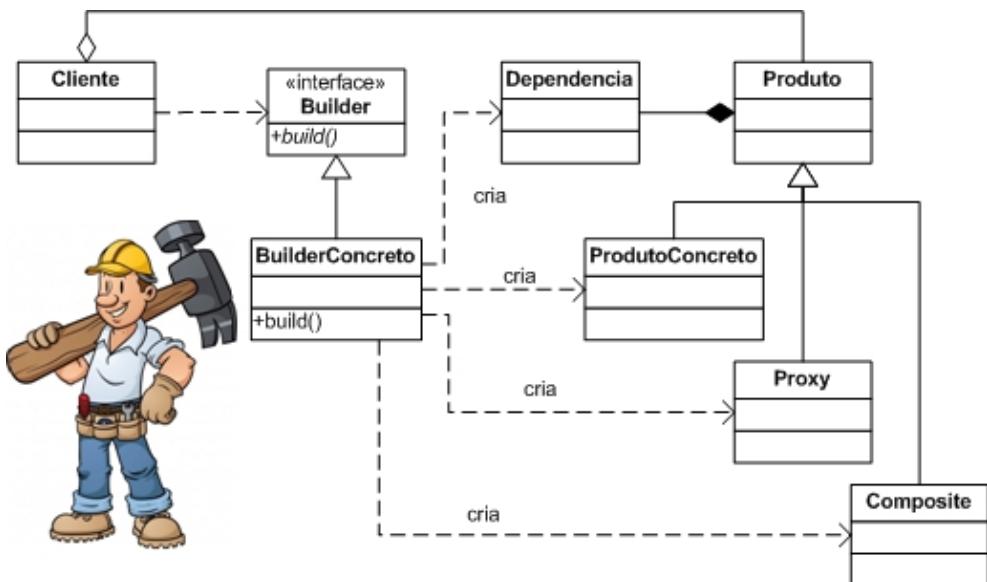


Figura 6.1: Representação do padrão Builder

Na figura alguns outros padrões estão representados para ilustrar a complexidade que o processo de criação pode possuir. Veja que a classe representada como Produto pode possuir subclasses, pode ser composta por outras, pode ser envolvida por um Proxy e pode ter suas implementações combinadas por um Composite, isso entre outros padrões que podem ser utilizados. Nenhum desses outros padrões citados são obrigatórios para a utilização de um Builder, porém a presença deles aumenta a complexidade do processo de criação favorecendo a utilização de um padrão de criação de objetos.

## Criando o SessionFactory do Hibernate com um Builder

Para melhorar a compreensão do padrão Builder essa seção apresenta um exemplo do seu uso no framework Hibernate para a criação do objeto da classe SessionFactory. O Hibernate é um framework cuja principal funcionalidade é realizar o mapeamento objeto-relacional, ou seja, entre classes e tabelas do banco de dados. Para que isso possa ser feito, é preciso que diversas configurações sejam lidas de arquivos e de anotações em classes, tornando a criação de um SessionFactory, a classe que gera as sessões para o acesso a base de dados, um processo bem complexo.

A listagem a seguir mostra a utilização da classe AnnotationConfiguration

para a criação de um `SessionFactory`. Nesse contexto, a classe `AnnotationConfiguration` é a implementação do padrão `Builder`. Observe como diversos método são chamados para adicionar configurações em relação a como essa classe deve ser criada. O encadeamento na chamada dos método se deve ao fato de cada um deles retornar a própria instância de `AnnotationConfiguration`, com exceção do método `buildSessionFactory()` que retorna a instância de `SessionFactory`.

```
SessionFactory sessionFactory = new AnnotationConfiguration()
    .addPackage("com.lojavirtual")
    .addAnnotatedClass(CarrinhoCompras.class)
    .addAnnotatedClass(Cliente.class)
    .addAnnotatedClass(Produtos.class)
    .addResource("orm.xml")
    .configure()
    .buildSessionFactory();
```

Esse é um bom exemplo de uso do padrão `Builder` pois envolve realmente um processo complexo de criação. A partir da leitura de arquivos XML e das anotações das classes mapeadas são extraídas as informações necessárias para o `SessionFactory`. É importante ressaltar que essa lógica extrapola as responsabilidades da classe `SessionFactory`, fazendo todo sentido a separação da responsabilidade da sua criação em uma classe separada.

## Builder com Interface Fluente

O termo **interface fluente** foi cunhado por Martin Fowler e Erick Evans [?], como uma forma de descrever um estilo de construção de interfaces. A ideia é dar o nome dos métodos da classe de forma que o código pareça uma frase em linguagem natural. A listagem a seguir mostra o exemplo de uma interface tradicional, utilizando métodos começados com `get` e `set`, e a mesma classe utilizando uma interface fluente. Com uma interface fluente é como se você estivesse definindo uma nova linguagem dentro da linguagem de programação a partir dos nomes dos métodos, prática que também é chamada de DSL (Linguagem Específica de Domínio) interna [?].

```
//Estilo comum
Pessoa p = new Pessoa();
p.setNome("João");
```

```
p.setDataNascimento("30/03/1985");
p.setCargo("gerente");

//Interface fluente
Pessoa p = new Pessoa()
    .chamada("João")
    .nascidaEm("30/03/1985")
    .comCargo("gerente");
```

A parte mais importante de uma interface fluente está na nomenclatura utilizada para os métodos. Eles devem soar como se fossem parte de uma frase, de forma que “*uma nova pessoa chamada João*” vira `new Pessoa().chamada("João")`. Outra questão é que os métodos devem retornar a instância da própria classe, ou seja, `this`, para as chamadas de método possam ser encadeadas. No exemplo apresentado, as chamadas dos métodos `chamada()`, `nascidaEm()` e `comCargo()` retornam a própria instância de `Pessoa`.

Muitas vezes, as aplicações precisam de utilizar as interfaces no padrão Java Beans (utilizando métodos com `get` e `set`), por requisito dos frameworks utilizados. Isso acaba inviabilizando a utilização da interface fluente nas classes de domínio. Porém, isso não impede que elas sejam utilizadas nos Builders, que criam essas classes. É importante mais uma vez ressaltar que para que a utilização do Builder ser adequada, é preciso que o processo de criação do objeto em questão seja complexo e exija a configuração de diversos parâmetros.

Um dos frameworks pioneiros na utilização de interfaces fluentes para a criação de objetos foi o framework JMock [?]. Este framework é utilizado para a criação de mock objects, que são objetos falsos utilizados para substituir as dependências de uma classe em testes de unidade. A listagem a seguir apresenta alguns trechos de código do JMock para definir o comportamento do objeto falso e suas expectativas em relação as chamadas da classe testada. Observe que apesar das limitações relativas a sintaxe da linguagem Java, é possível ler o código como se fosse uma frase.

```
//uma chamada ao método log() com uma string contendo 'erro'
one (logger).log(with(stringContaining("error")));

//exatamente 3 chamadas de count()
//retornando consecutivamente 10, 20 e 30
exactly(3).of(counter).count();
will(onConsecutiveCalls(returnValue(10),
    returnValue(20),returnValue(30)));
```

```
//permitir chamada ao método sqrt com valor menor que zero
//irá lançar a exceção IllegalArgumentException
allowing (calculator).sqrt(with(lessThan(0)));
will(throwException(new IllegalArgumentException()));
```

## Exemplos de Utilização do Builder

Para exemplificar os conceitos do padrão Builder e de interface fluente, será utilizado como contexto o mesmo exemplo para a criação da classe GeradorArquivo mostrado com o Static Factory Method. A listagem a seguir mostra a implementação da classe BuilderGerador. Essa classe possui um atributo `instancia` que armazena o objeto que está sendo construído. Ao chamar um dos dois métodos `gerandoEmXML()` ou `gerandoEmPropriedades()` a subclasse correta é recuperada e atribuída ao atributo. Já os métodos `comCriptografia()` e `comCompactacao()` adicionam os processadores na instância sendo construída, utilizando o Composite quando houver mais de um. No final de todo processo, o método `construir()` deve ser chamado para retornar o objeto criado.

```
public class BuilderGerador {

    private GeradorArquivo instancia;

    public BuilderGerador gerandoEmXML() {
        instancia = new GeradorXML();
        return this;
    }

    public BuilderGerador gerandoEmPropriedades() {
        instancia = new GeradorPropriedades();
        return this;
    }

    public BuilderGerador comCriptografia() {
        adicionaProcessador(new Criptografador());
        return this;
    }

    public BuilderGerador comCompactacao() {
        adicionaProcessador(new Compactador());
        return this;
    }

    private void adicionaProcessador(PosProcessador p) {
```

```
    PosProcessador atual = instancia.getProcessador();
    if(atual instanceof NullProcessador) {
        instancia.setProcessador(p);
    }else{
        PosProcessadorComposto pc = new PosProcessadorComposto();
        pc.add(atual);
        pc.add(p);
        instancia.setProcessador(pc);
    }
}
public GeradorArquivo construir() {
    return instancia;
}
}
```

Para deixar o exemplo mais interessante, vamos imaginar que existam ainda Proxys que possam ser utilizados para encapsular a classe `GeradorArquivo`. Dentro desse exemplo, imagine que a classe `ProxyAssincrono` invoque o método de geração de arquivos em uma thread diferente e que a classe `LoggerProxy` grave as chamadas realizadas em um arquivo de log para fins de auditoria. A listagem a seguir mostra como essas classes poderiam facilmente ser incorporadas no `Builder`. Ao ser chamado o método, o proxy encapsula a instância armazenada e o resultado é atribuído a ela mesmo.

```
public class BuilderGerador {
    //...
    public BuilderGerador assincrono() {
        instancia = new ProxyAssincrono(instancia);
        return this;
    }
    public BuilderGerador registrandoAuditoria() {
        instancia = new LoggerProxy(instancia);
        return this;
    }
}
```

Para ver como fica a construção do objeto utilizando interface fluente, a listagem a seguir mostra um exemplo de código. Observe como a construção do objeto flui de forma fácil entre as chamadas de método para sua configuração.

```
GeradorArquivo ga = new BuilderGerador().gerandoEmXML()
```

```
.comCriptografia().comCompactacao().assincrono()  
.registrandoAuditoria().construir();
```

## 6.5 RELACIONANDO FAMÍLIAS DE OBJETOS COM ABSTRACT FACTORY

*“Família que \${algumVerbo} unida, permanece unida.”*

– Dito popular

Nos padrões anteriores, foram abordados problemas referentes a complexidade de criação dos objetos, além de seu desacoplamento da classe cliente e da própria classe que está sendo criada. Um outro problema surge quando mais de um objeto relacionado precisa ser criado. Muitas vezes existem famílias de objetos nos quais existem classes relacionadas em hierarquias paralelas. Um exemplo é quando uma mesma API é implementada por diversos fornecedores. Nessa caso, cada um deles fornece implementações diferentes para as mesmas abstrações. Não faz sentido utilizar uma classe de um fornecedor junto com outra classe de outro, pois apesar de obedecerem as mesmas abstrações elas não podem ser misturadas.

Um padrão que existe que foca na solução desse problema é o **Abstract Factory**. Nesse padrão, ao invés de termos uma fábrica para a criação de um objeto, ela é destinada a criação de uma família de objetos relacionados. Dessa forma, se todos os objetos relacionados forem obtidos a partir da mesma fábrica, não haverá inconsistência entre eles.

Um exemplo que temos na plataforma Java é a API JDBC. A classe `Connection` representa uma conexão com o banco de dados e também é utilizada para criar outros objetos utilizados para interagir com o banco de dados. Se misturarmos os objetos obtidos de conexões diferentes (o `ResultSet` do MySQL com o `Statement` do PostGreSQL, por exemplo) obteremos um erro, porém o fato de existir uma instância como um ponto central de onde todos podem ser obtidos evita que esse tipo de erro aconteça. Nesse caso, a classe `Connection` faz o papel de **Abstract Factory**, pois só vai fabricar objetos relativos ao banco de dados ao qual ela sabe se comunicar.

### Estrutura do Abstract Factory

A estrutura básica do padrão **Abstract Factory** está apresentada no diagrama a seguir. As classes `ProdutoA` e `ProdutoB` representam as classes pertencentes a uma família. Dessa forma, a família “X” é representada pelas implementações `ProdutoAX`

e ProdutoBX e o similar ocorre com a família “Y”. O exemplo apresenta apenas dois tipos de produto e duas famílias, porém em uma implementação desse padrão pode haver mais de cada um deles.

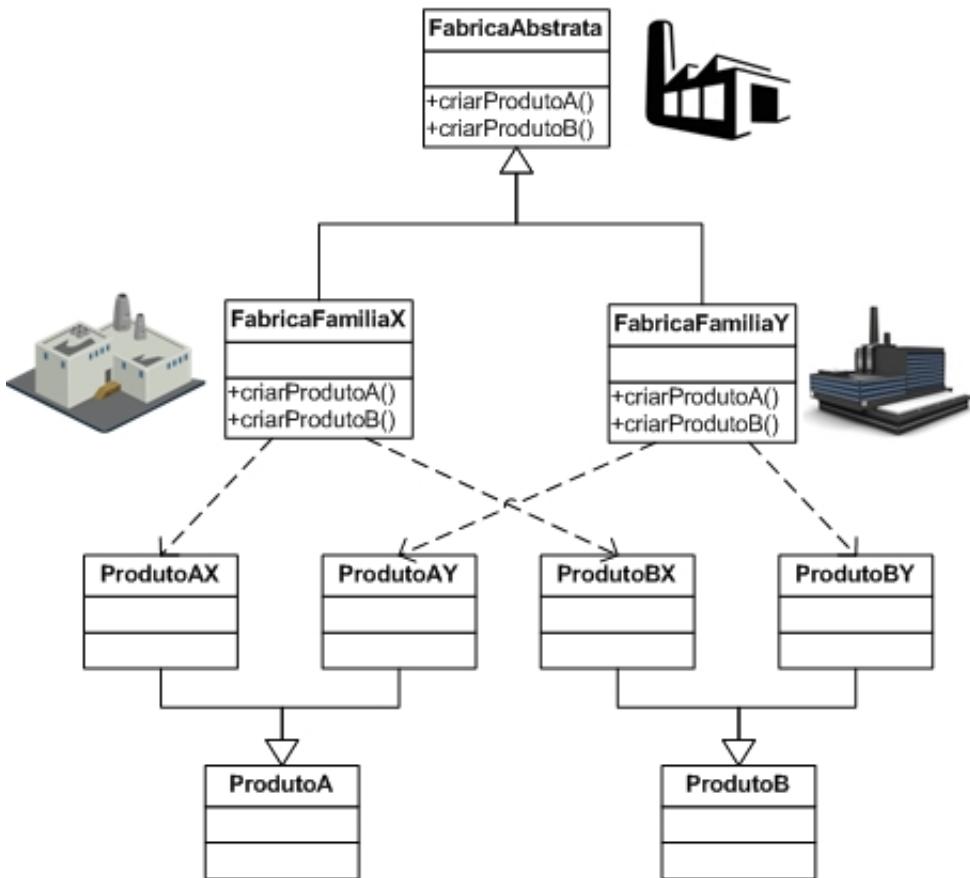


Figura 6.2: Estrutura Básica do Abstract Factory

A classe **FabricaAbstrata** é a que abstrai a criação de uma família de objetos. Cada implementação é responsável pela criação de uma das famílias de objeto e possui métodos para a criação de cada um dos tipos de objeto da família. Como pode ser visto no diagrama, cada família possui uma implementação da **Abstract Factory**. Por exemplo, a **FabricaFamiliaX** cria instâncias das classes **ProdutoAX** e **ProdutoBX**.

Existe uma troca importante que se faz quando se utiliza o **Abstract Factory**. Utilizando esse padrão, é muito fácil criar e introduzir na aplicação uma nova família

de objetos. Para isso, basta criar as implementações para as abstrações dos objetos que serão criados e uma nova `Abstract Factory` que os retorne. Por outro lado, esse padrão dificulta a criação de um novo tipo de objeto pertencente a essa família. Isso exigiria a adição de um método de criação na `Abstract Factory`, exigindo modificações em todas as suas implementações. Em resumo, é fácil adicionar uma nova família de objetos, porém é difícil adicionar um novo tipo de objeto na família.

## Criando Objetos para o Envio de SMS

Vamos imaginar que uma aplicação precise interagir com o sistema de SMS de diversas operadoras de telefonia móvel. Também vamos supor que existam diversos objetos que precisem ser criados para interagir com esse serviço. Esses objetos são:

- `ObservadorSMS`: Recebe eventos relativos a chegada de mensagens destinadas a aplicação.
- `FiltroSMS`: Configura um filtro que restringe as mensagens que devem ser recebidas ou tratadas por um determinado objeto.
- `EnviadorSMS`: Responsável por enviar SMS para dispositivos móveis.
- `RespondedorAutomatico`: Configura respostas automáticas para determinados tipos de mensagem.

Dependendo da operadora, alguns serviços, como a resposta automática ou o filtro para o recebimento de mensagens, pode ser implementada diretamente no servidor ou pela API cliente. Por esse motivo, não é possível, por exemplo, utilizar um filtro de uma operadora para filtrar as mensagens recebidas por um observador de outra operadora. O mesmo vale para a classe `RespondedorAutomatico` que também pode ser criada a partir de um filtro.

Dessa forma, o padrão `Abstract Factory` pode ser utilizado para concentrar a criação desses objetos em uma única classe. Esse objeto utilizado para a criação não precisa ser necessariamente apenas destinado a isso, podendo representar algum conceito do sistema. Nesse caso, a classe que irá criar os objetos pode ser a representação de uma conexão do sistema com os servidores da operadora. A seguir está representada a listagem que representa a abstração dessa classe.

```
public interface ConexaoOperadora{  
    public FiltroSMS criarFiltro(String expressao);
```

```
public EnviadorSMS criarEnviador();
public ObservadorSMS criarObservador();
public ObservadorSMS criarObservadorComFiltro(FiltroSMS f);
public RespondendorAutomatico criarRespostaAutomatica(FiltroSMS f);
//outros métodos referentes a conexão
}
```

Pode-se observar pela listagem que existem métodos de criação que recebem uma instância de `FiltroSMS` como parâmetro. Caso um objeto criado a partir de uma operadora diferente fosse passado como parâmetro, um erro seria lançado, por mais que a abstração fosse aceita pelo tipo do parâmetro. A concentração desses métodos de criação na mesma classe ajuda a evitar que esse tipo de mistura de objetos ocorra.

O fato da `Abstract Factory` ser definida a partir de uma abstração permite que os clientes possam interagir de forma transparente com diferentes operadoras. Essa abstração também permite que cada classe tenha suas particularidades, como por exemplo, os parâmetros que são necessários para que a conexão possa ser estabelecida.

## 6.6 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Esse capítulo focou em padrões para a criação de objetos. O padrão `Static Factory Method` encapsula a criação de objetos permitindo que sejam retornadas instâncias de subclasses, instâncias já existentes e se tenha uma expressividade maior a partir de nomes mais significativos. Foi visto também o padrão `Singleton` como um caso especial do `Static Factory Method` para o cenário em que deve haver apenas uma instância de uma determinada classe.

Para casos mais complexos, onde diversos parâmetros podem ser utilizados para construção do objeto, o `Builder` é um padrão adequado. Também foi mostrado como esse padrão pode ser útil quando outros padrões que deixam a estrutura dos objetos mais complicada, como `Proxy`, `Template Method` e `Composite`, são empregados nas classes que precisam ser criadas. Finalmente, foi abordado o `Abstract Factory`, que foca em cenários onde famílias de objetos relacionados precisam ser criadas.



## CAPÍTULO 7

# Modularidade

*“Nosso objetivo definitivo é a programação extensível. Por isso queremos dizer a construção de uma hierarquia de módulos, cada um adicionando nova funcionalidade ao sistema.”*

– Niklaus Wirth

No capítulo anterior foram apresentados diversos padrões para a criação de objetos. Esses padrões, de uma forma geral, desacoplam a classe cliente da sua criação. Isso permite que essa classe dependa apenas da sua abstração, possibilitando que novas implementações sejam criadas e incorporadas a classe responsável pela criação. Apesar disso, a classe cliente acaba dependendo indiretamente da classe que está sendo criada. Por exemplo, a classe cliente pode depender de um *Builder*, que por sua vez depende das implementações. O mesmo vale para os outros padrões!

Essa dependência, mesmo que indireta, impede que a classe cliente e as implementações sejam divididas em módulos independentes. No exemplo citado do *Builder*, a classe cliente precisa ter acesso a ele, o qual precisa ter acesso as implementações. Com essa estrutura não é possível, por exemplo, inserir dinamicamente

novas implementações, permitindo que elas sejam utilizadas pela classe cliente, pois para isso a classe responsável pela criação, como o `Builder` também precisaria de ser modificada.

A modularidade é cada vez mais um requisito não-funcional importante nas aplicações. Porém modularidade não é apenas dividir o software em módulos que formam um único bloco, mas permitir novos módulos possam ser criados e incorporados no software sem a necessidade de sua modificação. Um exemplo de modularidade é o IDE Eclipse, onde novos plugins podem ser facilmente e dinamicamente incorporados.

Para que esse tipo de modularidade possa ser atingida, é necessário não somente a utilização de interfaces e abstrações para a interação entre os objetos, mas também um desacoplamento no momento de sua criação. Dessa forma, se uma classe cliente utilizar um `Builder` ou qualquer outro intermediário para instanciar seus objetos, essa classe não pode depender diretamente das implementações. Isso irá permitir que novas implementações sejam incorporadas sem modificar a aplicação.

## 7.1 FÁBRICA DINÂMICA DE OBJETOS

*“Vidas fortes são motivadas por fins dinâmicos; as menores por desejos e inclinações.”*  
– Kenneth Hildebrand

A primeira pergunta a ser respondida, é como obter um objeto sem depender direta ou indiretamente de sua classe? Em Java, a API de reflexão permite que uma classe seja instanciada a partir de uma string com o seu nome. Isso permite que as classes que serão instanciadas possam ser definidas, por exemplo, em um arquivo de configuração. Dessa forma, ao ler esse arquivo, é possível instanciar a classe sem depender diretamente dela. Isso permite que novas classes possam ser simplesmente adicionadas ao *classpath* da aplicação e configuradas no arquivo.

O padrão `Dynamic Factory` [?] propõe essa estrutura para a criação de objetos. Ela é aplicável quando se deseja criar um objeto de uma abstração conhecida, ou seja, que possui uma determinada interface ou superclasse, porém que a implementação não pode ser determinada em tempo de compilação. Isso normalmente ocorre quando deseja-se permitir que novas implementações possam ser incorporadas ao software sem a sua modificação, tornando-o mais extensível e flexível. Um exemplo seria um software que precisa ser implantado em diversos clientes e em alguns deles é necessário desenvolver classes específicas para aquele contexto.

## Estrutura do Padrão

A estrutura do padrão Dynamic factory não é muito diferente da estrutura de outros padrões de criação. Dessa forma, a classe cliente irá depender apenas da abstração compartilhada pelas implementações e irá acessar uma classe que implementa uma fábrica dinâmica, responsável pela criação dos objetos. Essa fábrica por sua vez acessa uma outra classe responsável por ler as informações a respeito de que implementação deve ser instanciada para uma determinada interface ou superclasse. A partir dessa informação, a API de reflexão é utilizada para instanciar essa classe.

A Figura 7.1 mostra a estrutura do padrão. A classe representada como `LeitorMetadados` é responsável por ler as informações a respeito de quais implementações devem ser instanciadas. Essas informações podem estar armazenadas em um arquivo de configuração, em um banco de dados ou até mesmo nas anotações de alguma classe. Independente da forma como essas configurações são definidas, é importante que elas contenham qual a implementação que deve ser instanciada para uma determinada abstração.

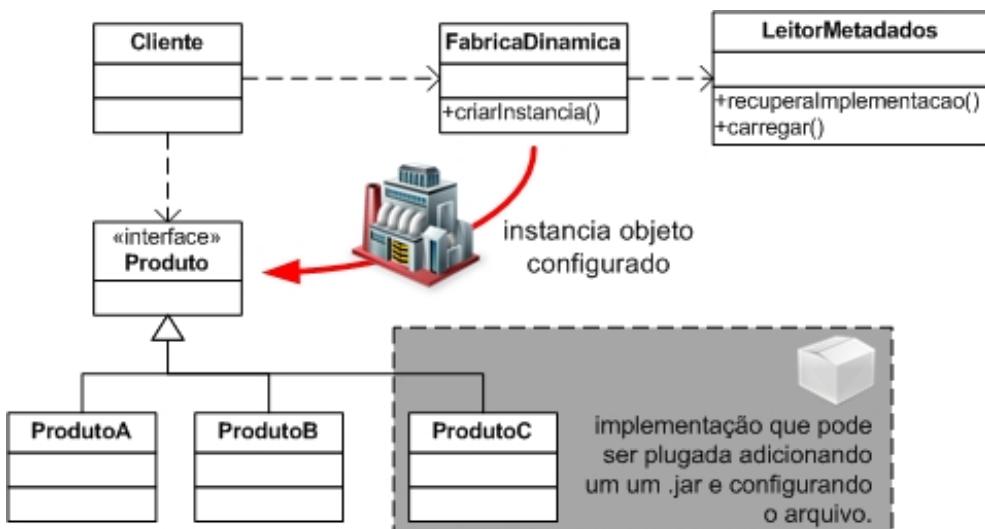


Figura 7.1: Estrutura do Dynamic Factory

A partir dessa estrutura é muito fácil introduzir no sistema novas classes. Para isso basta configurar o nome dessa classe no local onde o `LeitorMetadados` obtém as informações e adicionar a classe em um arquivo `.jar` no classpath da aplicação. Sendo

assim, não apenas a classe `Cliente` irá depender apenas da abstração de `Produto`, mas o processo de criação dentro da classe `FabricaDinamica` também.

### O QUE SÃO METADADOS?

A palavra “metadado” é sobrecarregada de significado dependendo da área da computação em que estamos. De uma forma geral, metadado significa “dado sobre o dado”, que no contexto de uma linguagem orientada a objetos seriam as informações a respeito das classes de um sistema. Nesse caso, os dados são as informações da aplicação e as classes contém informações sobre os seus tipos, sua visibilidade e como são manipulados. Dessa forma, no contexto de uma classe, os metadados seriam seus atributos, seus métodos, suas interfaces, sua superclasse, entre outros.

Na estrutura do padrão `Dynamic Factory`, a classe `LeitorMetadados` é responsável por ler uma informação a respeito de uma abstração, que é a classe que deve ser instanciada naquele contexto. Dessa forma, como essa é uma informação referente a uma classe do software, pode-se dizer que ele está lendo um novo metadado para essa classe.

O padrão `Dynamic Factory` pode ser implementado em conjunto com um `Builder` ou com um `Static Factory Method` de acordo com as necessidades da aplicação. Se necessário, mais de uma classe pode ser configurada para uma mesma abstração, assim como classes que serão utilizadas para composição, ou mesmo para envolver a classe principal, como um `Proxy`. O que diferencia esse padrão dos vistos no capítulo anterior é o desacoplamento entre a fábrica e as implementações que estão sendo criadas.

## Dynamic Factory na Prática

Para exemplificar a criação de uma fábrica dinâmica com o uso de reflexão, a listagem a seguir mostra o exemplo de uma classe que cria os objetos de acordo com o nome da classe configurado em um arquivo de propriedades. A classe `FabricaDinamica` possui um construtor que recebe o nome do arquivo com as configurações de criação. A classe `Properties` da própria API padrão do Java é utilizada para fazer essa leitura, fazendo o papel de leitora de metadados.

```
public class FabricaDinamica {  
  
    private Properties props;  
  
    public FabricaDinamica(String arquivo)  
        throws FileNotFoundException, IOException{  
        props = new Properties();  
        props.load(new FileInputStream(arquivo));  
    }  
  
    public <E> E criaImplementacao(Class<E> interface) {  
        String nomeClasse = props.getProperty(interface.getName());  
  
        if(nomeClasse == null) {  
            throw new IllegalArgumentException(  
                "Interface não configurada");  
        }  
  
        try {  
            Class clazz = Class.forName(nomeClasse);  
            if (interface.isAssignableFrom(clazz)) {  
                return (E) clazz.newInstance();  
            } else {  
                throw new IllegalArgumentException(  
                    "Classe configurada não implementa a interface");  
            }  
        } catch (ClassNotFoundException e) {  
            throw new IllegalArgumentException(  
                "Classe configurada não existe",e);  
        } catch (Exception e) {  
            throw new IllegalArgumentException(  
                "Não foi possível criar a implementação",e);  
        }  
    }  
}
```

Muitas vezes, classes já existentes no sistema ou em algum framework utilizado pela aplicação podem ser utilizados para ser um participante de um padrão. No exemplo apresentado da Dynamic factory, foi utilizada a classe Properties para realização da leitura do arquivo com os metadados. Em outros padrões, como o

Observer, por exemplo, o papel de observador ou observado pode ser desempenhado por alguma classe já existente. Caso seja requerida uma interface diferente para a classe existente, o padrão Adapter pode ser utilizado para fazer a adaptação.

A classe do exemplo espera que no arquivo sejam definidas propriedades com o nome da interface e valores com o nome da classe que deve ser instanciada. Dessa forma, quando o método `criaImplementacao()` é invocado, o nome da interface é utilizado para recuperar a respectiva implementação que deve ser instanciada. Essa instanciação é feita, primeiramente obtendo-se a instância de `Class` referente àquela implementação através do método estático `Class.forName()`, e em seguida invocando o método `newInstance()`. Observe que antes de efetivamente criar o objeto é feita uma verificação se a classe realmente implementa a interface. Nesse caso, com a chamada do método `newInstance()`, o construtor sem parâmetros da classe será invocado. Sendo assim, para que essa fábrica seja utilizada, é preciso que as implementações possuam esse construtor.

### REFLEXÃO EM JAVA

Reflexão é a capacidade de um sistema de executar computações a respeito de si mesmo. De uma forma mais prática, isso envolve o sistema poder obter informações, acessar e até mesmo modificar suas próprias classes. A API Reflection da linguagem Java provê funcionalidades apenas para obter informações e acessar as classes do sistema. Funcionalidades de modificação de classes são mais comuns em linguagens dinâmicas.

A classe básica da API Reflection é a classe `Class` que representa uma classe do sistema. A partir dela é possível obter as informações de uma classe, como seu nome, seus métodos, seus atributos, sua superclasse e suas interfaces. Além de obter informações, também é possível instanciar essa classe através do método `newInstance()`, como visto no exemplo. Outras classes dessa API representam outros elementos de código, como `Method`, `Field` e `Constructor`. Está fora do escopo desse livro a explicação do funcionamento dessa API, porém é importante saber esses fundamentos para a compreensão de como o padrão Dynamic Factory funciona.

Para exemplificar a utilização da classe `FabricaDinamica`, será retomado o exemplo do gerador de arquivos. Para simplificar o exemplo, vamos supor que as

implementações possuem um construtor sem parâmetros e que o pós-processador é configurado através de um método setter. A versão apresentada da classe `FabricaDinamica` não suporta o carregamento de mais de uma implementação da mesma interface, porém isso poderia ser contornado com uma pequena modificação. Para manter o exemplo simples, será configurada apenas uma instância da interface `PosProcessador` para ser inserida no `GeradorArquivo`.

A seguir são apresentadas listagens que mostram um exemplo de como o arquivo de propriedades seria definido e como a criação das classes seria realizada. Observe que o código que instancia o `GeradorArquivo` com sua dependência não referencia em nenhum momento a implementação. Adicionalmente, a implementação também não é referenciada pela classe `FabricaDinamica`, que lê o nome da classe de um arquivo externo. Esse tipo de prática permite que novas classes sejam criadas e plugadas no sistema, sem a necessidade da recompilação das classes já existentes.

```
br.com.casadocodigo.GeradorArquivo=br.com.casadocodigo.GeradorXML  
br.com.casadocodigo.PosProcessador=br.com.casadocodigo.Compactador  
  
FabricaDinamica f = new FabricaDinamica("configuracoes.prop");  
GeradorArquivo gerador = f.criaImplementacao(GeradorArquivo.class);  
gerador.setPosProcessador(f.criaImplementacao(PosProcessador.class));
```

## Uma Base Para Outros Padrões

Mesmo sendo um padrão importante, a `Dynamic Factory` é muitas vezes utilizada por baixo dos panos. Apesar de ser válida sua utilização direta para a criação de alguns objetos do sistema, pode ser complicado utilizar esse padrão como estratégia de criação geral dentro de uma arquitetura. As próximas seções apresentam padrões para a criação desacoplada de objetos que podem ser utilizados no lugar da `Dynamic Factory`. Observem que para que eles sejam possíveis de ser implementados para se obter modularidade, ainda é necessário que uma `Dynamic Factory` atue no carregamento dinâmico dessas classes em algum momento.

## 7.2 INJEÇÃO DE DEPENDÊNCIAS

*“A verdadeira felicidade é... aproveitar o presente, sem a ansiedade da dependência do futuro.”*

– Lucius Annaeus Seneca

Quando um objeto precisa da colaboração de outro para cumprir suas respon-

sabilidades, é um processo natural de que o próprio objeto crie ou busque essas instâncias. Essa responsabilidade pela criação das dependências, quando está no próprio objeto, pode acabar criando um acoplamento e prejudicando a modularidade, mesmo quando interfaces são utilizadas para interação entre as classes.

Uma forma de evitar que a própria classe seja responsável por criar suas dependências é criá-las de forma externa e inseri-las no objeto no momento ou depois de sua criação. Essa é a ideia principal do padrão Dependency Injection [?]. Apesar de se utilizar muito o nome desse padrão em português, Injeção de Dependências, para padronizar a utilização dos nomes dos padrões em inglês nesse livro, ele será referenciado como Dependency Injection.

### INVERSÃO DE CONTROLE?

Devido ao fato desse padrão inverter a responsabilidade de criação das dependências de uma classe, ele também é conhecido como Inversão de Controle. Porém o nome “inversão de controle” também é utilizado para denominar práticas para o desenvolvimento de frameworks, nas quais as classes do framework invocam classes da aplicação, invertendo assim o controle do fluxo de execução. Dessa forma, o nome Dependency Injection acabou prevalecendo.

Apesar de sua simplicidade, o Dependency Injection é um padrão muito poderoso, pois desacopla a classe de suas dependências, permitindo que elas possam ser reutilizadas em diferentes contextos. Consequentemente, a testabilidade dessa classe acaba também sendo aumentada, pois objetos voltados para o teste podem ser injetados no lugar das dependências.

## Funcionamento da Injeção

O foco do padrão Dependency Injection é na criação e configuração de uma classe que possui uma outra como dependência. Essa dependência, por sua vez, é do tipo de uma abstração, normalmente representada por uma interface, a qual possui normalmente mais de uma implementação. O problema no caso é como criar e conectar essas instâncias sem que exista uma dependência de uma para outra. Esse padrão propõe como solução a existência de uma outra classe que irá “montar” a aplicação, criando as implementações corretas de cada componente e conectando-as de forma a se obter o comportamento desejado.

A Figura 7.2 apresenta uma representação do padrão Dependence Injection. A classe Montador é quem na verdade coordena todo o processo, tanto de criar as instâncias corretas, quanto de injeta-las na classe Produto. Essa, por sua vez, deve apenas prover uma forma das dependências poderem ser injetadas. A próxima seção explora mais detalhes a respeito desse processo.

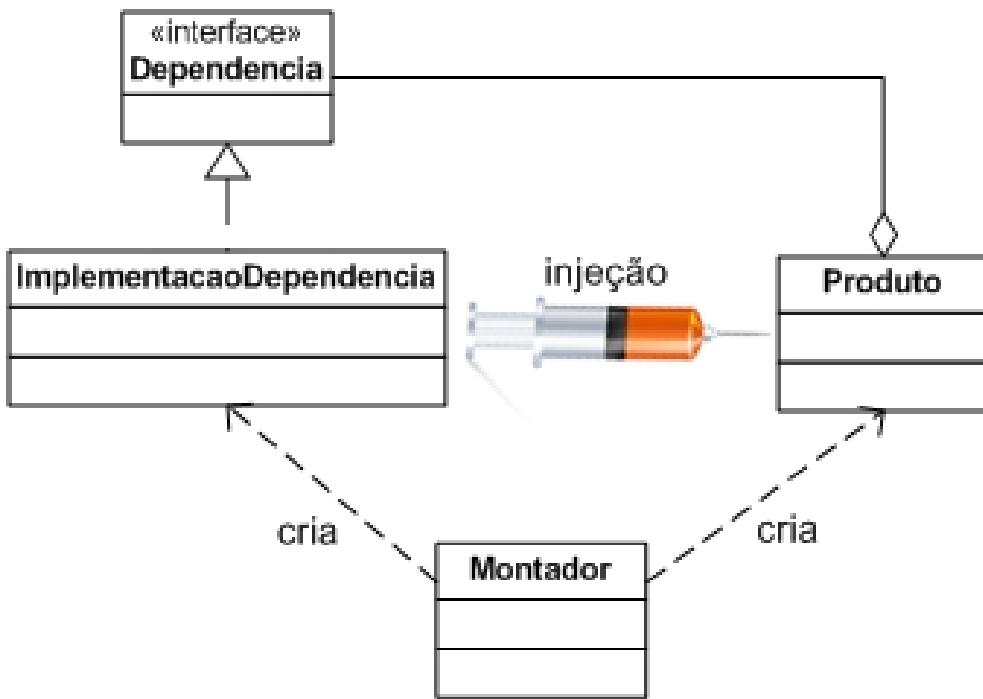


Figura 7.2: Injeção de Dependências

O processo de criação e reutilização de instâncias é controlado pelo Montador, que pode definir diferentes estratégias para isso. Por exemplo, caso a dependência seja um objeto que possa ser compartilhado, a mesma instância pode ser injetada diversas vezes. Uma outra estratégia, muito comum para conexões a serviços ou servidores remotos, é a criação de um **pool** de objetos, de onde eles são retirados para a injeção em uma classe e retornados ao final de sua utilização. O fato desse processo não ser controlado pela classe permite que ela possa ser reutilizada em diversos contextos.

Imagine uma classe que realize acesso a uma base de dados e receba através de uma Dependency Injection a conexão que deve ser utilizada por ela. Essa classe

poderia ser utilizada por uma aplicação desktop onde apenas uma conexão é criada para toda aplicação, mas também poderia ser reusada em uma aplicação Java EE que executa em um servidor de aplicação no qual a conexão é obtida de um **pool**. O simples fato dessa conexão ser criada de forma externa a classe e injetada em sua estrutura é que a torna reutilizável nas diferentes arquiteturas citadas.

Como pode-se perceber, grande parte do trabalho acaba ficando com a classe **Montador**, responsável pela criação e conexão entre as classes envolvidas. Ela pode ser simples e criada especificamente para uma aplicação, ou ser genérica e carregar dinamicamente as classes que precisam ser instanciadas através de uma **Dynamic Factory**. Felizmente, existem frameworks e APIs, como o Spring e o CDI do Java EE, que implementam montadores que criam as instâncias a partir de configurações em arquivos XML e anotações. Sendo assim, raramente é necessário criar uma classe para fazer o papel de montador. Mais à frente será mostrado como a injeção de dependências funciona no Spring.

## Formas de Injetar Dependências

A injeção de dependências pode ser habilitada de diversas formas em uma classe. Essa seção irá explorar as alternativas existentes, assim como a motivação na utilização de cada uma delas. De alguma forma, deve ser possível que uma entidade externa reconheça a dependência e possa inseri-la na classe, mesmo que em apenas certos momentos de seu ciclo de vida.

O tipo mais comum de injeção de dependências é através do método **setter**. Através desse método, a classe permite que a dependência seja configurada com um método **setter**, que a recebe como parâmetro. A listagem a seguir mostra como a classe **AcessoDados** disponibiliza um método **setter** para que a conexão com a base de dados possa ser configurada. Utilizando essa estratégia, a dependência pode ser atribuída e alterada a qualquer momento.

```
public class AcessoDados {  
    private Connection connection;  
    public void setConnection(Connection c) {  
        connection = c;  
    }  
    //...  
}
```

Uma outra abordagem para a injeção de dependências é o construtor. Nessa estratégia, a dependência deve ser injetada no objeto no momento de sua criação.

Esse caso acaba sendo mais indicado para cenários onde a dependência é obrigatória e essencial para o funcionamento da classe. A listagem a seguir mostra o mesmo exemplo da classe `AcessoDados`, porém utilizando a `Dependency Injection` através do construtor.

```
public class AcessoDados {  
    private Connection connection;  
    public AcessoDados(Connection c) {  
        connection = c;  
    }  
    //...  
}
```

Como foi apresentado no capítulo anterior, a utilização de construtores traz diversas restrições e a classe que utiliza injeção por construtores acaba sofrendo os mesmos problemas. Por outro lado, o construtor é uma forma de garantir que uma instância da classe não será criada sem receber aquele parâmetro. Uma dependência bidirecional, por exemplo, seria impossível de ser injetada por construtores nas duas classes... Na prática, a injeção por métodos `setter` acaba sendo mais comum, sendo feita uma configuração cuidadosa para que o montador não deixe de configurar as dependências importantes.

Uma outra abordagem existente para a `Dependency Injection` é a utilização de interfaces. Nesse caso, é definida uma interface que declara métodos que devem ser utilizados para a injeção da dependência. Dessa forma, a classe que precisa da dependência deve implementar essa interface para poder recebê-la. Apesar de ser uma abordagem mais “burocrática” que as outras duas, ela permite que a classe que injeta a dependência reconheça facilmente o objeto que precisa dela para poder proceder com a injeção.

A estratégia de utilização de interfaces para injetar dependências é muito utilizada quando o montador precisa gerenciar diversas instâncias e passar certos objetos apenas para as classes que precisam deles. A interface acaba funcionando como um marcador que indica quando o objeto precisa da dependência.

Para exemplificar essa situação, considere um sistema que possui uma abstração chamada `Processo` que representam processos de negócio que podem ser executados. Das classes que implementam esses processos, algumas delas precisam saber qual é o usuário que as está executando. Para não tornar essas classes acopladas a como as instâncias da classe `Usuario` são armazenadas e acessadas, tomou-se a decisão de utilizar `Dependency Injection` para que o usuário seja inserido nessas ins-

tâncias. A listagem a seguir mostra a definição da interface que as classes que desejam receber a instância de **Usuario** devem implementar.

```
public interface CienteUsuario{  
    public void recebeUsuario(Usuario u);  
}
```

Como forma de injetar a dependência, foi criado um Proxy que encapsula classes da interface Executor, que é responsável por executar as classes do tipo Processo passadas a elas. A classe ProxyInjecaoUsuario, apresentada na próxima listagem, verifica com o operador instanceof se o objeto implementa a interface e injeta a dependência da classe Usuario em caso positivo.

```
public class ProxyInjecaoUsuario implements Executor {  
    private Executor executor;  
    private Usuario usuario;  
    public ProxyInjecaoUsuario(Executor e, Usuario u) {  
        executor = e;  
        usuario = u;  
    }  
    public void executar(Processo p) {  
        if(p instanceof CienteUsuario) {  
            ((CienteUsuario)p).recebeUsuario(usuario);  
        }  
        executor.executar(p);  
    }  
}
```

Pelas técnicas anteriores de injeção de dependências, pode ser observado que a classe deve sinalizar de alguma forma onde e qual a dependência que precisa ser injetada. Uma outra forma de se sinalizar isso, bastante utilizada em frameworks e APIs mais recentes, é através de anotações. A anotação é normalmente incluída em um atributo ou no seu respectivo método **setter** onde a dependência deve ser injetada. A listagem a seguir mostra um exemplo de um Servlet que deve receber uma injeção de dependência de um EJB do servidor de aplicações onde está implantado. Note que nesse caso é feita utilização de reflexão para que a dependência possa ser inserida diretamente em um atributo, mesmo ele sendo privado.

```
public class ServletLogin extends HttpServlet {  
    @EJB
```

```
private ServicoAutenticacao servico;  
//lógica do servlet omitida  
}
```

## Os Efeitos Colaterais da Injeção

Se por um lado a Dependency Injection tem o potencial de tornar a classe mais reutilizável, a passagem de responsabilidade da montagem do relacionamento das classes para um terceiro também pode gerar erros em tempo de execução. Como a classe não tem mais controle sobre suas dependências, ela fica sujeita a erros referentes a uma falha nesse processo que ocorre de forma externa.

Um erro bastante comum quando esse padrão é utilizado ocorre quando uma dependência deixa de ser configurada. Nesse caso, é lançada a conhecida e indesejável `NullPointerException` quando a classe tenta invocar algum método nessa dependência. Por uma questão de segurança de código, isso pode levar muitos desenvolvedores a incluirem diversos condicionais no código para verificar se a instância não é nula, o que gera uma certa poluição e verbosidade no código.

Apesar da injeção de dependências simplificar os testes de unidade, devido a chance de erro com as montagens, quando esse padrão é utilizado, é muito importante que se façam os testes de integração da aplicação. Esses testes servem para garantir, não somente que as classes estão funcionando de forma adequada juntas, mas que a aplicação está criando e injetando as instâncias de forma adequada. Seria como se estivéssemos testando se o montador está fazendo a injeção de forma adequada.

A montagem dinâmica da aplicação também pode dificultar a compreensão do contexto global de como os objetos interagem para formar a funcionalidade da aplicação. A flexibilidade de uma classe em poder interagir com diversas implementações, também dificulta saber com que classes ela está interagindo em um contexto concreto. Por exemplo, ao se deparar com um erro, o desenvolvedor não tem como saber qual foi a classe invocada por ela. Para isso ele deve procurar no montador ou nas configurações da aplicação para saber qual era a estrutura naquele caso. Essa indireção acaba sendo um efeito colateral ao desacoplamento conquistado.

## Injeção de Dependências no Spring

Normalmente, quando a injeção de dependências é utilizada em uma aplicação, é utilizado algum framework para realizar a montagem das classes. Um dos frameworks que primeiro se destacou por esse tipo de funcionalidade e muito utilizado até hoje é o Spring. Utilizando-o, as instâncias com suas respectivas dependências

são definidas em um arquivo XML e o framework se encarrega de criá-las.

A listagem a seguir exemplifica como o Spring seria utilizado no exemplo do gerador de arquivo. Cada instância é definida como um **bean** no arquivo de configuração, recebendo um nome que será utilizado para referências internas e para recuperá-lo de forma externa. Utilizando o nome dos **beans**, as instâncias podem ser injetadas utilizando diferentes estratégias, como pelo construtor ou através de propriedades.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans...>
    <bean id="compactador" class="br.com.casadocodigo.Compactador" />
    <bean id="criptografador" class="br.com.casadocodigo.Criptografador">
        <constructor-arg type="int" value="3"/>
    </bean>
    <bean id="composite"
          class="br.com.casadocodigo.PosProcessadorComposto">
        <constructor-arg>
            <ref bean="compactador"/>
            <ref bean="criptografador"/>
        </constructor-arg>
    </bean>
    <bean id="geradorArquivo" class="br.com.casadocodigo.GeradorXML">
        <property name="processador" ref="composite" />
    </bean>
</beans>
```

No exemplo, o **bean** criptografador recebe um parâmetro no construtor referente ao tamanho do deslocamento de caracteres utilizado no algoritmo. As referências dos **beans** criptografador e compactador são também passadas como parâmetro para o construtor do **bean** composite. Finalmente, o **bean** geradorArquivo recebe uma injeção do **bean** composite na propriedade processador. Através do arquivo XML apresentado, é possível definir qual classe será instanciada para cada **bean** e como uma deve ser utilizada na construção da outra.

Os usuários do framework Spring, para recuperarem um **bean** devem fazer uso da classe `ApplicationContext`. Essa classe é responsável por carregar o arquivo XML com a definição da composição dos objetos e retorna-los a aplicação. A listagem a seguir mostra como essa classe seria utilizada na criação de uma instância do `GeradorArquivo` como definido no arquivo XML. Em aplicações web, o Spring normalmente é integrado diretamente com o container web já criando os objetos e

injetando-os nas classes que tratam as requisições. Nesse caso, não há a necessidade de acesso direto a classe `ApplicationContext` pela aplicação, pois isso é feito pelo container web ou pelo framework que está sendo utilizado.

```
String xml = "applicationContext.xml";
ApplicationContext ac = new ClassPathXmlApplicationContext(xml);
GeradorArquivo ga = (GeradorArquivo) ac.getBean("geradorArquivo");
```

A partir do que foi mostrado, é possível perceber como seria simples a mudança das implementações envolvidas e da configuração da estrutura montada. Por serem definidos em um arquivo XML, os **beans** podem ser alterados, consequentemente mudando o comportamento da aplicação, sem a necessidade de sua recompilação. Para exemplificar esse fato, imagine que seja necessário introduzir um Proxy na classe `GeradorArquivo` para a invocação assíncrona de seus métodos. Isso poderia ser facilmente realizado colocando a classe do Proxy no **bean** `geradorArquivo`, fazendo-o encapsular o que estava definido anteriormente, conforme mostrado na listagem a seguir.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans...>
    ...
    <bean id="geradorArquivo"
          class="br.com.casadocodigo.GeradorProxyAssincrono">
        <constructor-arg>
            <ref bean="geradorEncapsulado"/>
        </constructor-arg>
    </bean>
    <bean id="geradorEncapsulado" class="br.com.casadocodigo.GeradorXML">
        <property name="processador" ref="composite" />
    </bean>
</beans>
```

## 7.3 SERVICE LOCATOR

*“Quando tentar localizar algo, procure primeiro em sua mente.”*  
– Craig Bruce

O padrão Service Locator [?] é uma alternativa ao padrão Dependency Injection para permitir a modularidade nas aplicações. Diferentemente da injeção, as próprias classes são responsáveis por buscar as suas dependências. Porém,

elas fazem isso através de uma outra classe responsável por localizar a implementação que deve ser utilizada para uma determinada interface.

No contexto desse padrão, a palavra “serviço” é utilizada para denotar uma colaboração de uma outra classe. Seria como se a classe externa estivesse prestando um serviço para a classe que a utiliza. Nesse caso, a classe principal declara o tipo de serviço que ela precisa utilizando uma abstração, como uma classe abstrata ou uma interface, e então utiliza um `Service Locator` para encontrar a classe que irá prestar aquele serviço para ela.

### SERVICE LOCATOR COMO UM J2EE PATTERN

O `Service Locator` foi documentado como um Core J2EE Pattern e foi muito utilizado para a localização de serviços remotos, principalmente enterprise beans, em aplicações J2EE [?]. Nesse contexto, o `Service Locator` se conectava a um repositório JNDI para adquirir uma referência a esses serviços remotos e fazia um cache dessa referência para evitar acessos desnecessários ao registro de nomes. Nas novas versões da plataforma enterprise Java, que perdeu o “2” e agora se chama simplesmente Java EE, a utilização desse padrão ficou obsoleta a própria plataforma já presta esse serviço, realizando inclusive a injeção dessas dependências nas classes.

Por esse motivo, quando se fala em `Service Locator` muita gente acha que é um padrão obsoleto pela experiência que teve com ele na plataforma J2EE. Nesse contexto, esse padrão tinha o objetivo principal de centralizar o mecanismo de busca de serviços, encapsulando as peculiaridades de cada registro e evitando buscas desnecessárias. Porém, fora desse contexto, esse padrão ainda é muito importante, principalmente para permitir a modularidade das aplicações. Sendo assim, se você ainda guarda algum trauma da utilização desse padrão para buscar EJBs, deixe isso de lado porque ele pode ser muito útil em outras situações.

## Estrutura para Localização de Serviços

No `Service Locator`, não existe a figura de uma classe que centraliza a responsabilidade de criação e ligação das classes que irão compor a aplicação. Nesse padrão

cada classe busca suas próprias dependências através de uma classe responsável por buscar a instância que deve ser adicionada como dependência.

A Figura 7.3 apresenta a estrutura do padrão Service Locator. No diagrama, a classe Cliente representa a classe que precisa criar uma instância da abstração Servico. Apesar de no diagrama haver uma relação de agregação entre essas classes, nesse padrão a relação poderia ser mais fraca, como a criação e utilização do serviço de forma encapsulada dentro de um método. Segundo o padrão, uma classe responsável por encontrar e retornar a implementação desse serviço, representada por LocalizadorServicos, é utilizada pela classe Cliente para fazer isso.

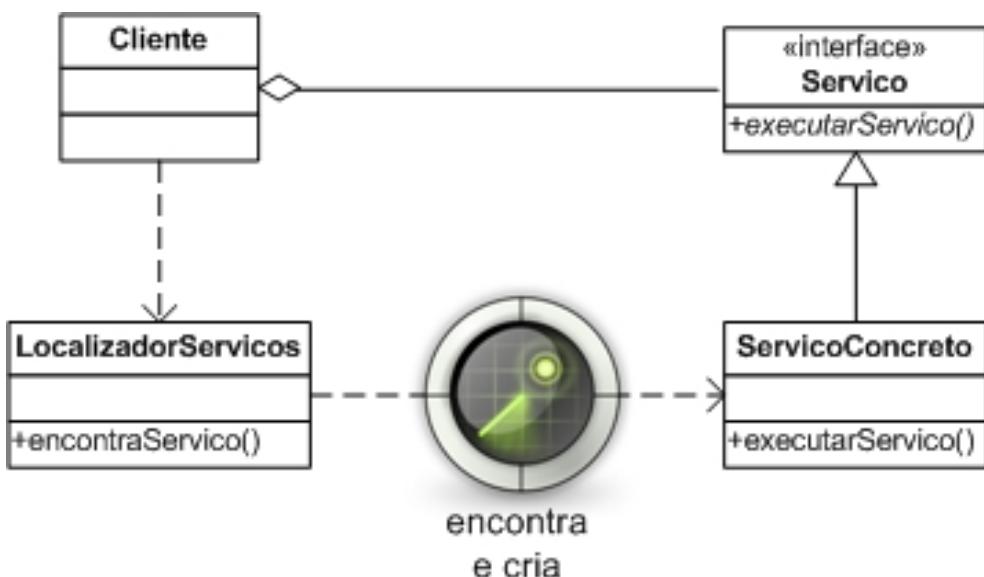


Figura 7.3: Estrutura do Service Locator

Uma classe que implemente um localizador de serviços pode possuir uma implementação fixa, retornando as instâncias desejadas de acordo com o parâmetro passado. Era esse tipo de Service Locator que costumava ser utilizado nas aplicações J2EE. Porém, para conseguir modularizar a aplicação, é preciso que ele faça uso de uma Dynamic Factory para que os serviços retornados possam ser mais facilmente alterados.

Quando o padrão Dependency Injection é utilizado, essa questão ainda precisa ser implementada, porém quem é o responsável por buscar os serviços não é alguém

que é invocado pelas classes, e sim o montador. Por isso, muitas vezes, o montador precisa de um `Service Locator` para poder localizar as implementações e injetá-las nas classes.

Um dos problemas desse padrão está no fato das classes ficarem acopladas a classe responsável pela busca de serviços. Isso impede essas classes de serem reutilizadas fora de um contexto em que o `Service Locator` esteja presente. De qualquer forma, o `Service Locator` costuma ser utilizado encapsulado na classe, não sendo uma dependência exposta na API externa da classe. Isso é bom pois não cria essa dependência nas classes cliente.

## Utilizando a classe `ServiceLoader`

A JDK possui uma classe chamada `ServiceLoader` que pode ser utilizada como um `Service Locator` que carrega dinamicamente as implementações de uma interface. Essas implementações podem estar empacotadas e configuradas dentro de diferentes arquivos JAR. Sendo assim, a implementação será carregada de acordo com os arquivos JAR que estiverem presentes no classpath da aplicação. Basta incluir e remover arquivos do JAR para alterar a implementação utilizada.

Para que a classe `ServiceLoader` possa ser utilizada, o primeiro passo seria ter uma interface que serviria de abstração para o serviço a ser criado. As implementações dessa interface poderiam então ser colocadas em arquivos JAR para serem localizadas. Além da implementação, ainda é necessário incluir no JAR um arquivo que registra a classe criada como uma implementação daquele serviço. Esse arquivo deve ser colocado dentro do diretório `META-INF/services` e deve ter o nome completo da interface, incluindo o pacote. Como conteúdo do arquivo, deve haver simplesmente uma linha com o nome da classe para cada implementação daquela abstração contida naquele JAR.

A Figura 7.4 ilustra como deve ser essa estrutura. Considere `Abstracao` como a interface para qual se deseje definir os serviços. No exemplo, o `arquivo1.jar` possui as implementações `ImplementacaoA` e `ImplementacaoB` e o `arquivo2.jar` possui a `ImplementacaoC`. Cada arquivo JAR possui um arquivo chamado `Abstracao`, o nome da interface, com os nome das classes. Dessa forma, de acordo com o arquivo JAR colocado no classpath da aplicação, classes diferentes seriam retornadas pelo `ServiceLoader`.

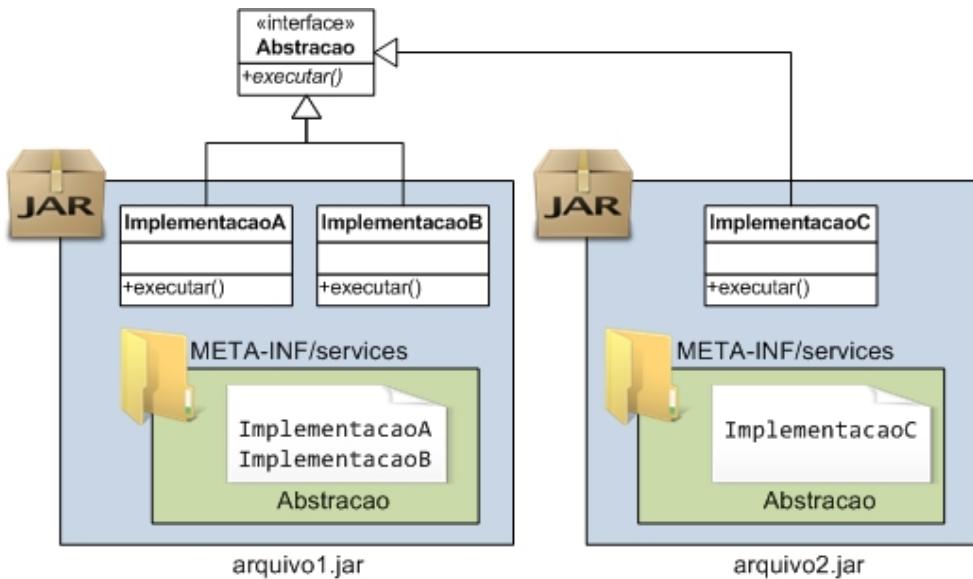


Figura 7.4: Arquivos JAR para utilização do ServiceLoader

A listagem a seguir ilustra, dentro do exemplo da Figura 7.4, como funciona a classe `ServiceLoader`. Esse trecho de código imprime no console todas as implementações que estão registradas para uma interface. O método estático `load()` da classe `ServiceLoader` é utilizado para carregar todas as implementações registradas da interface `Abstracao`. Em seguida, é utilizado um `Iterator` para percorrer as implementações recuperadas e imprimi-las no console.

```

ServiceLoader<Abstracao> sl = ServiceLoader.load(Abstracao.class);
Iterator<Abstracao> i = sl.iterator();
while(i.hasNext()) {
    System.out.println(i.next().getClass().getName());
}
  
```

Para mostrar um exemplo mais concreto do uso do `ServiceLoader`, a listagem a seguir mostra como ele poderia ser utilizado para a recuperação dos pós-processadores para a classe `GeradorArquivo`. Nesse caso, o `Service Locator` está sendo utilizado no construtor para popular o atributo `processador` da classe. Observe que caso exista apenas uma implementação, ela será atribuída diretamente ao atributo, mas caso exista mais de uma, a classe `PosProcessadorComposto` é utilizada. Dessa forma, caso um arquivo JAR configurado com uma nova implementa-

ção for adicionado ao classpath da aplicação, essa implementação será recuperada pelo ServiceLoader. No caso da classe PosProcessadorComposto, que é uma implementação que não deve ser retornada, basta não incluí-la no arquivo.

```
public abstract class GeradorArquivo {

    private PosProcessador processador = new NullProcessador();

    public GeradorArquivo() {
        ServiceLoader<PosProcessador> sl =
            ServiceLoader.load(PosProcessador.class);
        Iterator<PosProcessador> i = sl.iterator();
        List<PosProcessador> lista = new ArrayList<>();
        while(i.hasNext()) {
            lista.add(i);
        }
        if(lista.size() == 1) {
            processador = lista.get(0);
        } else if (lista.size > 1) {
            processador = new PosProcessadorComposto(lista);
        }
    }

    //demais métodos
}
```

## COMBINANDO DIVERSOS PADRÕES EM UMA SOLUÇÃO

No decorrer desse livro, vimos diversos padrões que aos poucos foram sendo incorporados as soluções apresentadas. A ideia é que aos poucos eles comecem a fazer parte da nossa forma de pensar e que acabamos utilizando-os mesmo sem perceber. Desafio você a dar mais uma olhada no código da última listagem e tentar identificar os padrões que estão ali presentes. Utilizamos o Bridge na abstração do GeradorArquivo e no uso de composição para os pós-processadores, o Null Object é utilizado para os casos onde não há pós-processador, o Composite é utilizado para quando há mais de um pós-processador, e, finalmente, o Service Locator é utilizado para o carregamento dinâmico dos pós-processadores.

Note que não estou dizendo para aplicar todos os padrões possíveis em uma mesma solução, pois isso é uma má prática. Porém, o que acaba acontecendo, é que na solução sucessiva dos problemas de design de uma classe, é natural que o resultado seja fruto da combinação de diversos padrões.

## 7.4 SERVICE LOCATOR VERSUS DEPENDENCY INJECTION

Depois de aprender sobre os padrões Dependency Injection e Service Locator, surge a dúvida de qual deles utilizar. Antes de começar a explorar as vantagens e desvantagens de cada um, é importante ressaltar que com a utilização de qualquer um dos dois é possível se obter modularidade, particionando a aplicação em partes efetivamente independentes. Dessa forma, fica mais simples a introdução de novas implementações sem a necessidade de recompilação do código das classes que as utilizam. Com isso, cada um dos módulos do sistema pode evoluir de forma independente, facilitando a manutenção e evolução do software.

Uma das grandes diferenças entre os dois padrões está em quem possui a responsabilidade de montar a configuração de objetos da aplicação. Quando o padrão Dependency Injection é utilizado, essa responsabilidade fica centralizada na classe responsável pela montagem. Nesse caso, essa classe precisa conhecer todas as dependências de todas as classes envolvidas, estaticamente ou dinamicamente a partir de alguma configuração. Diferentemente, quando se utiliza o Service Locator, cada

classe é responsável por buscar suas dependências a partir de uma classe responsável pela localização dos serviços. Isso permite que novas classes definam novos pontos onde outras abstrações possam ser inseridas, sem a necessidade de haver um ponto central saiba dessa nova dependência.

Em resumo, quando se injeta dependências, a responsabilidade de montar toda a aplicação fica centralizada e, quando cada classe busca seus serviços, essa responsabilidade fica distribuída. Sendo assim, se o contexto é uma aplicação onde se deseja ter modularidade para questões de manutenção e evolução, mas não existe a necessidade da adição de novas classes dinamicamente, então o Dependency Injection é uma boa solução. Porém, em cenários onde a arquitetura é baseada em plugins e não existe um ponto na aplicação onde se tem consciência de todas possíveis dependências, então o Service Locator se mostra mais adequado.

Uma consequência disso está em quão explícitas são as dependências de uma classe. Quando se usa o Dependency Injection, é fácil visualizar através dos construtores, interfaces e métodos de acesso quais são as dependências que podem ser injetadas. Por outro lado, a invocação de um Service Locator pode estar encapsulada dentro dos métodos da classe, não expondo externamente as dependências. Isso pode ter um efeito negativo, pois não deixa claro quais dependências precisam ser configuradas para uma determinada classe.

Apesar de não ser obrigatório, o Dependency Injection acaba sendo muito utilizado em ambientes onde os objetos da aplicação são gerenciados por um container. Um exemplo disso seriam aplicações web ou aplicações EJB, onde o ciclo de vida dos objetos é controlado pelo servidor. Dessa forma, como a criação dos objetos que tratarão as requisições fica a cargo do container, ele tem a oportunidade de injetar as dependências no momento certo. Frameworks como Spring são chamados de *lightweight container*, ou container leve, devido ao fato de controlarem o ciclo de vida dos objetos gerenciados por eles.

Um ponto onde o Dependency Injection certamente leva vantagem é na testabilidade. Como a dependência é injetada na classe, esse mesmo mecanismo pode ser utilizado para a injeção de um objeto falso, ou *mock object* [?], para isolar a classe durante o teste. Já com o Service Locator, isso pode ser um pouco mais complicado, pois o *mock object* precisaria ser configurado para ser retornado por ele. Dependendo de como essa configuração é feita, isso pode ser mais simples ou mais complicado, porém de qualquer forma pode ser um problema para casos em que a classe do *mock* é gerada dinamicamente por algum framework. Nesses casos, uma solução seria já pensar em uma implementação que simplifique a substituição da

implementação retornada para um serviço em código de teste.

## ALTERNATIVAS DE PROJETO

Os padrões de projeto documentam soluções recorrentes que são utilizadas em um determinado contexto. Por mais que eles documentem boas soluções, nem sempre um determinado padrão é a melhor solução para uma aplicação, e é por isso que podem existir mais de um padrão que pode ser aplicado na resolução de um mesmo problema, como mostrado nesse capítulo. Se você quer flexibilizar os passos de um algoritmo, você pode utilizar o *Template Method* para fazer isso com herança, ou o *Strategy* para usar composição. Se você quer combinar soluções de classes existentes de forma transparente, pode-se utilizar o *Composite* ou o *Chain of Responsibility*. Por mais que ambos os padrões nesses casos resolvam o problema, eles possuem características e consequências diferentes. Dessa forma, é importante não utilizar os padrões cegamente e avaliar dentre as alternativas existentes, quais as que possuem as consequências mais adequadas para o contexto da sua aplicação.

## 7.5 CONSIDERAÇÕES FINAIS DO CAPÍTULO

Nesse capítulo abordamos padrões que focam em soluções para que se consiga obter modularidade para aplicações. Essas soluções atuam em um momento crucial na vida dos objetos: a sua criação. Desde o primeiro capítulo desse livro, já foram apresentadas soluções que utilizam encapsulamento e polimorfismo para desacoplar os objetos, porém no momento da criação, onde a implementação efetivamente precisa ser instanciada, a referência direta a classe costuma interferir na modularidade.

O padrão *Dynamic Factory* foi apresentado como uma base que pode ser utilizada pelos outros padrões para o carregamento dinâmico das implementações que devem ser utilizadas para compor a classe principal. Em seguida, foram apresentados os padrões *Dependency Injection* e *Service Locator*, que apresentam alternativas a respeito de como uma classe pode delegar a criação de suas dependências. Se combinados com uma solução que utiliza um *Dynamic Factory* na criação das instâncias, é possível realmente modularizar a aplicação.



## CAPÍTULO 8

# Indo Além do Básico

*“Ao infinito... E além!”*

– Buzz Lightyear, Toy Story

Durante todo esse livro trilhamos um caminho que nos apresentou diversos padrões que podem ser utilizados para projetar um software orientado a objetos. Cada capítulo focou em uma técnica ou em tipo de problema, para os quais alguns padrões foram apresentados. Nem todos os padrões do GoF [?] foram apresentados, porém, com o conhecimento dos padrões e técnicas apresentados nesse livro, acredito que o leitor tenha uma boa base para a realização e evolução do projeto de uma aplicação. Além disso, com essa visão adquirida, fica mais simples a compreensão de outros padrões.

Esse último capítulo não presenta nenhum padrão novo, porém explora diversos tópicos mais avançados e recentes a respeito da utilização de padrões. Ele aborda, por exemplo, a utilização dos padrões para o desenvolvimento de frameworks e como os tipos genéricos podem ser utilizados de forma efetiva na implementação dos padrões. Também são discutidas questões a respeito da utilização de padrões com a

técnica de desenvolvimento Test-Driven Development e como os padrões apresentados se aplicam para problemas mais amplos relacionados a arquitetura. Finalmente, o livro e o capítulo finaliza falando um pouco sobre a comunidade nacional e internacional de padrões.

## 8.1 FRAMEWORKS

*“Um framework não é bom pelo que ele faz, mas quando pode ser utilizado para o que ele não faz.”*

– Eduardo Guerra

Nos dias de hoje, é muito difícil alguém desenvolver uma aplicação sem a utilização de um framework, principalmente na linguagem Java. Ele não apenas provê um reuso de seu código, como também o reuso de sua estrutura, de seu design. Dessa forma, além de possibilitar o reaproveitamento de funcionalidades acelerando o ritmo de desenvolvimento, ele também direciona a arquitetura da aplicação a utilização de boas práticas de código.

Um framework pode ser visto como um software incompleto que precisa ser preenchido com partes específicas de uma aplicação para poder ser executado [?]. Imagine, por exemplo, um framework que faça o agendamento de execuções. Sozinho, esse framework não faz nada, pois não tem o que ser agendado sem a existência de uma aplicação. Ao ser instanciado, esse framework é completado com classes da aplicação que serão invocadas por ele, e então faz sentido a sua execução.

Um framework apresenta uma estrutura abstrata referente a um determinado domínio. Esse domínio pode ser horizontal, referente a uma camada de aplicação, como apresentação ou persistência, ou vertical, referente ao domínio da aplicação, como seguros ou comércio eletrônico. De qualquer forma, o framework captura o conhecimento desse domínio, identificando os principais papéis que as classes desempenham e como elas se relacionam. A partir disso, o framework provê um estrutura reutilizável que pode ser utilizada para a criação de aplicações dentro daquele domínio.

Um framework possui pontos com funcionalidade fixa, chamados de *frozen spots*, e pontos com funcionalidade variável, chamados de *hotspots*. Os *hotspots* são pontos de extensão disponibilizados pelo framework, onde a aplicação pode inserir parte do seu código. Os *frozen spots* são os que efetivamente provêm a funcionalidade do framework e que coordenam a execução dos *hotspots*. Igualmente importantes, os *hotspots* e os *frozen spots* formam a arquitetura do framework em conjunto.

Um framework propõe uma forma de reuso muito diferente de uma biblioteca de funções ou de classes. Quando uma biblioteca é utilizada, a aplicação é responsável por invocar a funcionalidade das classes, coordenando sua execução e utilizando-a como um passo de seu processamento. Quando um framework é utilizado, normalmente é ele quem tem o controle sobre a execução e invoca a funcionalidade das classes da aplicação em pontos específicos. Isso permite que a aplicação reutilize o código e a estrutura do framework, especializando-o para suas necessidades.

Um framework também é diferente de padrões de projeto, pois representa efetivamente uma implementação, um software. Os padrões, como já foi dito durante o livro, já existem como conhecimento, como uma ideia. Porém, muitos padrões são utilizados para viabilizar a criação de *hot spots* na estrutura dos frameworks. Os *hook methods* e as *hook classes*, apresentados respectivamente nos capítulos 2 e 3, são exemplos de técnicas que podem ser utilizadas. Através da herança, a aplicação pode estender uma classe do framework e inserir funcionalidade em um método. Com composição, a aplicação pode implementar uma interface do framework e inserir uma classe para ser invocada em um determinado ponto da execução.

Os tipos de *hot spots* apresentados nessa seção não são os únicos que podem ser utilizados em frameworks. A utilização de reflexão e metadados, especialmente definidos com anotações, tem sido uma prática cada vez mais utilizada em frameworks mais recentes [?]. Porém essas questões já estão além do escopo desse livro.

## Ainda existem bibliotecas?

Biblioteca de funções é um conceito que veio da programação estruturada, onde o reúso permitido pelo paradigma de programação era mais restrito. Porém, ainda existem algumas bibliotecas de funções em Java, por exemplo, como as classes `Math` e `Collections`. Em aplicações, essa biblioteca de funções é frequentemente colocada em alguma classe terminada com “`Utils`”. Nessa classe, usualmente são incluídos métodos, normalmente estáticos, que são de uso geral da aplicação.

Um conceito de biblioteca inserido na programação orientada a objetos são as bibliotecas de classe. Agora, ao invés de termos apenas métodos reutilizáveis, temos classes! Um exemplo de biblioteca de classes muito popular em Java é a `Collection API`. Diferentemente dos frameworks, as bibliotecas de classes são simplesmente classes reutilizáveis, que não permitem nenhum tipo de extensão. O conceito de biblioteca de classes também é aplicável dentro da estrutura de um framework como um conjunto de classes que compartilha de uma mesma abstração e podem ser inseridas em um de seus hotspots. Por exemplo, em um framework de validação

de instâncias onde o algoritmo de validação de uma propriedade é um *hotspot*, faria sentido uma biblioteca de classes com os tipos mais comum de validação, como formato de string, limites numéricos, entre outros.

Muitas vezes, em aplicações, utilizamos classes como se fossem parte de uma biblioteca, porém se olharmos com calma sua documentação, veremos que ela possui diversos *hotspots* onde é possível estender o seu comportamento, se caracterizando mais como um framework. Isso tem a ver com um padrão chamado de Low Surface-to-volume Ratio [?], cuja tradução do nome seria “baixa proporção da superfície para o volume”. Esse padrão tem a ver com você tentar prover uma série de serviços a partir de uma interface externa compacta. Isso torna o framework mais simples de ser utilizado, muitas vezes fazendo parecer que quem está provendo aquele serviço é uma simples classe. O Facade é muitas vezes utilizado para se atingir esse objetivo.

Outro fator que faz com que muitas vezes os frameworks se pareçam com simples classes, vem do fato dos *hotspots* serem configurados por default com classes mais comuns de serem usadas. Outra alternativa são os *hotspots* serem configurados através de Builders que cria e injeta as classes de acordo com parâmetros e configurações. Esse tipo de prática propicia uma Gentle Learning Curve [?], ou curva de aprendizado suave, pois o desenvolvedor usuário do framework não precisa conhecer e configurar todos os *hotspots* para instanciar o framework. Porém, caso seja necessário, ele pode ir aos poucos aprendendo mais sobre a sua estrutura para alterar o comportamento quando for necessário.

## Outros Exemplos de Frameworks

Se você já desenvolveu alguma aplicação em Java, muito provavelmente você já usou algum framework. Muitas vezes os desenvolvedores acabam não percebendo que a classe que estão desenvolvendo faz parte de um contexto mais amplo e é invocada por um framework em um *hotspot*. Essa seção irá apresentar alguns exemplos de framework para que os conceitos apresentados possam ser vistos a partir de uma perspectiva mais concreta.

Vou começar com o exemplo de frameworks para aplicações web, pois além de ser um domínio familiar para muitos leitores, eles possuem um *hotspot* bem óbvio. O que muda de uma requisição para outra? A funcionalidade que é executada quando a requisição chega no servidor! Dessa forma, a classe responsável por tratar uma requisição, chamada de *controller* em um modelo arquitetural MVC [?], acabe sendo um *hotspot* nesse framework. Exemplos de classes desse tipo seriam Servlets na API

padrão Java EE, Actions no framework Struts e Managed Beans no JSF. Observe que antes da classe da aplicação receber o controle da requisição, diversas outras funcionalidades são executadas pelo framework, indo desde a tradução dos parâmetros até controle de acesso.

Um outro exemplo interessante de framework é o Quartz (<http://quartz-scheduler.org/>), cujo principal objetivo é realizar o agendamento de tarefas. É possível perceber que pelo domínio desse framework, sua execução não faz sentido fora do contexto de uma aplicação. Por mais que o framework possua classes implementadas para as mais variadas opções de agendamento, sem a aplicação ele não tem o que agendar. Dessa forma, o principal *hotspot* do framework são as tarefas com lógica da aplicação a serem executadas de acordo com o agendamento realizado. A listagem a seguir mostra um exemplo de uma classe para ser executada pelo framework.

```
public class JobExemplo implements Job {  
    public void execute(JobExecutionContext context)  
        throws JobExecutionException{  
            //lógica específica da aplicação  
    }  
}
```

Um exemplo de framework um pouco diferente é o Log4J, cujo objetivo é fazer o log de uma aplicação. Dentro desse domínio, o Log4J é bastante flexível, permitindo a gravação de diversos tipos de informações, em diferentes formatos e em diferentes locais, como no console, em arquivos, em bancos de dados e até remotamente. Diferentemente dos outros frameworks citados, o Log4J já possui uma biblioteca de classes que podem ser utilizadas em seus *hotspots*. Dessa forma, uma aplicação poderia utilizar o framework sem implementar nenhum de seus *hotspots*, porém se for necessário ela pode, por exemplo, criar classes para adição de propriedades específicas no log ou para que seja realizado o log em um local diferente. A partir das configurações do Log4J, apresentadas na listagem a seguir, é possível perceber que existem várias propriedades que recebem nomes de classe, que na verdade estão configurando implementações para serem usadas nos *hotspots*.

```
# Configura o nível de log para o logger principal como INFO.  
log4j.rootLogger=INFO, Console, LogFile
```

```
# Configuração do log do console  
log4j.appender.Console=org.apache.log4j.ConsoleAppender
```

```

log4j.appenders.Console.layout=org.apache.log4j.PatternLayout
log4j.appenders.Console.layout.ConversionPattern=%d{ISO8601}
                                         [%t] %-5p %C - %m%n

# Configuração do log em arquivo
log4j.appenders.LogFile=org.apache.log4j.DailyRollingFileAppender
log4j.appenders.LogFile.DatePattern='.'yyyy-MM
log4j.appenders.LogFile.file=/path/to/logfile.log
log4j.appenders.LogFile.layout=org.apache.log4j.PatternLayout
log4j.appenders.LogFile.layout.ConversionPattern=%d{ISO8601}
                                         [%t] %-5p %C - %m%

```

## Enxergando o GeradorArquivo como um Framework

O exemplo do gerador de arquivo utilizado ao longo desse livro pode ser considerado um framework. Isso porque a classe `GeradorArquivo` não apenas pode ser invocada pela aplicação, como também provê pontos onde essa funcionalidade pode ser estendida e especializada por aplicações. A Figura 8.1 apresenta quais seriam os *hotspots* e os *frozen spots* do gerador de arquivos. Exemplos de *frozen spots* estão no algoritmo de geração de arquivos e na combinação dos pós-processadores no Composite. Já os *hot spots* seriam o formato de geração dos arquivos, as formas de pós-processamento e a própria ordem de execução dos pós-processadores.

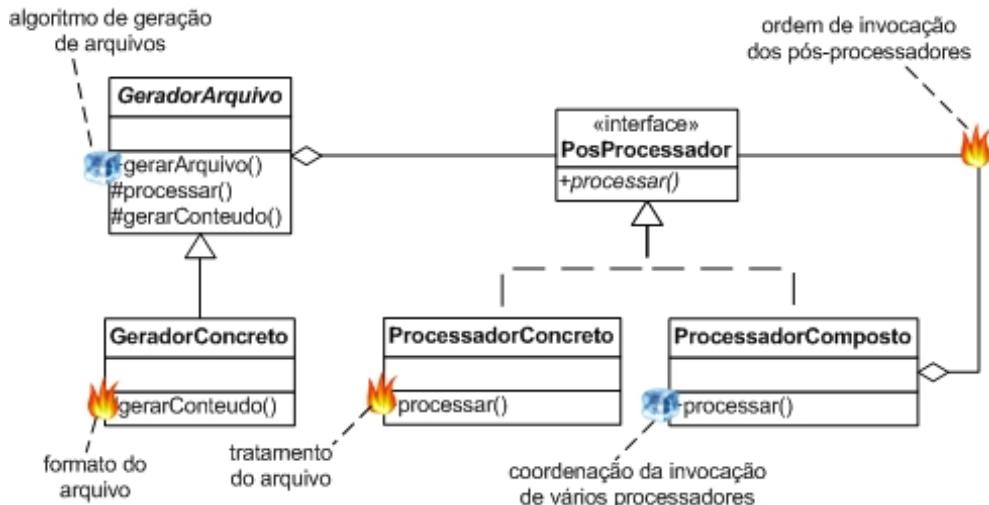


Figura 8.1: Hotspots e frozen spots do gerador de arquivo

Uma coisa importante de se ressaltar é que nem sempre precisa ser uma classe da aplicação a ser utilizada no lugar de um *hot spot*. Um framework pode possuir uma biblioteca de classes que representam implementações mais gerais que podem ser utilizadas pela aplicação em um ponto de extensão. Por exemplo, no gerador de arquivos, o *Compactador* e o *Criptografador* seriam classes mais gerais que poderiam fazer parte da biblioteca de classes do framework. Porém, nada impede que uma aplicação crie um implementação própria e utilize naquele ponto.

## 8.2 UTILIZANDO TIPOS GENÉRICOS COM OS PADRÕES

*“Eu apenas quero crescer criativamente e ser inspirado. Eu não quero fazer algo genérico ou burro.”*

– Sienns Miller

Os tipos genéricos são uma funcionalidade de linguagem introduzida no Java 1.5. A partir dela, é possível parametrizar tipos para que seja possível substituir partes da assinatura de seus métodos, como retornos e argumentos, pelo parâmetro passado para o tipo. Um cenário onde isso é muito aplicável é para coleções. Uma lista, por exemplo, ao ser instanciada pode receber como parâmetro o tipo que deve armazenar, de forma que métodos de recuperação retornem aquele tipo e métodos de inserção aceitem como parâmetro apenas aquele tipo.

O suporte de linguagem a tipos genéricos é uma funcionalidade que visa primeiramente a segurança de código. Antes de existirem tipos genéricos, as coleções no Java podiam receber qualquer objeto. O problema acontecia quando uma instância era recuperada e precisava ser convertida para um determinado tipo. Caso o objeto não fosse do tipo esperado, uma `ClassCastException` era lançada. Com tipos genéricos, além do código ficar mais limpo sem a necessidade de fazer o *cast*, erros como esse são pegos já em tempo de compilação.

Os tipos genéricos também podem ser utilizados na definição de interfaces. Dessa forma, a classe que implementar essa interface pode manter o tipo genérico para ser definido nas instâncias ou já definir o tipo utilizado no momento da implementação. A partir disso, uma mesma interface pode ser utilizada para implementação de diversas interfaces onde apenas o tipo em retornos e parâmetros são alterados. Baseado nesse modelo, outras classes podem utilizar o tipo da interface parametrizado para aceitarem apenas implementações que, por exemplo, retornarem aquele tipo.

Apesar dos padrões serem ideias mais gerais, os tipos genéricos podem ser muito úteis para a criação de implementações mais flexíveis e seguras. Por exemplo, em padrões onde existe a colaboração de duas classes, pode-se garantir em tempo de compilação, a partir do contrato entre as classes, que apenas instâncias com o mesmo tipo genérico podem se relacionar.

Para exemplificar, vamos utilizar o padrão `Observer`. Nesse padrão, um observador é aquela classe que possui o papel de receber notificações dos objetos que está observando. Como nessa definição, o objeto passado como parâmetro na notificação pode variar, este acaba sendo um excelente candidato para ser um parâmetro genérico da interface que define esse contrato. A listagem a seguir apresenta a interface `Observador` que define um parâmetro genérico relativo a classe do evento utilizada para notificação.

```
public interface Observador<E>{
    public void notificar(E evento);
}
```

Para que o software faça sentido, uma classe deve poder apenas ser observada por uma que consiga entender os eventos gerados por ela. Ou seja, alguém que gera eventos de uma classe, só poderia ser observado por alguém que recebe evento dessa classe, ou de alguma de suas abstrações. A listagem a seguir apresenta a interface `Observavel` que também define um tipo genérico. No caso, ela aceita no método `adicionaObservador()`, uma classe que implemente a interface `Observador` com o tipo genérico igual ou sendo uma abstração ao seu.

```
public interface Observavel<E>{
    public void adicionaObservador(Observador<? super E> o);
}
```

A listagem a seguir mostra como as interfaces poderiam ser implementadas e conectadas. Suponha uma classe chamada `ObservadorNumerico` que implemente a interface `Observer` com o tipo genérico `Number` e uma classe chmada `ObservavelInteiro` que implemente a interface `Observavel` com o tipo genérico `Integer`. Nesse caso, o observador poderia ser inserido na classe observável, pois uma classe que recebe eventos do tipo `Number` vai saber lidar com um evento do tipo `Integer`, por ser sua superclasse. Porém, uma classe que, por exemplo, implementasse `Observavel<String>` não pdoeria ser passada para a classe `ObservavelInteiro`.

```
public class ObservadorNumerico implements Observador<Number>{
    public void notificar(Number evento){
        ...
    }
}

public class ObservavelInteiro implements Observavel<Integer>{
    public void adicionaObservador(Observador<? super Integer> o) {
        ...
    }
}

ObservadorNumerico on = new ObservadorNumerico();
ObservavelInteiro oi = new ObservavelInteiro();
oi.adicionaObservador(on);
```

A utilização de tipos genéricos para implementação de padrões ajuda na criação de soluções que permitem uma maior segurança de código. No exemplo apresentado do padrão `Observer`, a própria estrutura montada impede que uma classe não compatível seja passada como parâmetro. Esse tipo de restrição não seria imposta se o método da interface `Observador` aceitasse um `Object` como parâmetro. Por outro lado, se eu quisesse restringir o tipo observado sem usar tipos genéricos, seria necessário a criação de uma interface para cada caso.

O uso do `Observer` foi apenas um exemplo para mostrar como a utilização de tipos genéricos pode ajudar a amarrar os tipos de uma solução com padrões para evitar inconsistências. Outros padrões também podem ser utilizados para se beneficiar desse recurso de linguagem [?]. O `Abstract Factory`, por exemplo, pode utilizar tipos genéricos para assegurar algum relacionamento entre os tipos retornados e o `Command` pode se utilizar desse recurso para permitir que as classes declarem no tipo o retorno resultante de sua execução.

## 8.3 PADRÕES COM TEST-DRIVEN DEVELOPMENT

*“TDD é uma ideia louca que funciona.”*

– Kent Beck

O Test-driven Development, TDD, é uma técnica de desenvolvimento e projeto de software onde os testes são criados antes do código de produção. É uma técnica

que vem se tornando cada vez mais popular no mercado e mais atenção da academia. Com essa técnica, as classes vão sendo desenvolvidas incrementalmente em pequenos ciclos, os quais alternam entre a criação de um teste, o desenvolvimento de código de produção para satisfazer o teste adicionado e a refatoração do código criado.

Durante a criação dos testes, o trabalho do desenvolvedor é focado na definição da API externa da classe e do seu comportamento esperado dentro de cada cenário definido. A partir de cada teste adicionado, a tarefa a seguir é implementar a solução mais simples possível na classe que está sendo desenvolvida, de forma ao novo teste e os anteriores serem todos satisfeitos. Como nesse momento o objetivo é fazer o teste passar, logo que isso é atingido, segue um momento para se refletir sobre o design e a clareza do código. Dessa forma, antes de seguir o desenvolvimento com a introdução de mais um teste, o código é refatorado e a manutenção do comportamento verificada com a execução dos testes.

Existem hoje muitos desenvolvedores que utilizam o TDD mais como uma técnica de desenvolvimento do que uma técnica de projeto. Em outras palavras, eles projetam como serão as classes e como elas irão se relacionar, e depois implementam a funcionalidade utilizando TDD. Quando o TDD é utilizado também como uma técnica de projeto, a modelagem da aplicação vai acontecendo de forma evolutiva, incluindo a definição das colaborações e dos contratos entre as classes. A seção a seguir fala sobre como o projeto ocorre com TDD e o papel dos padrões nesse processo.

## Projetando com TDD e Padrões

O projeto de uma classe com TDD acontece em grande parte na definição do teste. É nele que você vai expressar a API externa da classe e como ela é utilizada pelos seus clientes. O fato dos testes serem criados antes favorece certas características na classe que está sendo desenvolvida, que favorece a sua testabilidade. Uma dessas características é a coesão, pelo fato de que é mais simples testar uma classe com uma responsabilidade bem definida do que classes que acumulam diversas responsabilidades. Isso ajuda em um melhor particionamento das classes para uma melhor divisão de responsabilidades.

Outra questão que é trabalhada durante os testes é o desacoplamento da classe que está sendo desenvolvida das suas dependências. Apesar de não ser obrigatório no TDD, normalmente utiliza-se testes de unidade no desenvolvimento. Isso é feito para que o teste possa focar exclusivamente nas responsabilidades daquela classe. Para

isso, é necessário inserir objetos falsos (mock objects) na classe testada, e simular diferentes comportamentos das dependências para a criação dos cenários de teste.

Para possibilitar a inserção dos mock objects, é preciso que a classe possua um mecanismo no qual as dependências possam ser substituídas. Além disso, é preciso que seja definido um contrato, normalmente através de uma interface, que formalize a interação entre a classe desenvolvida e as dependências. Essas necessidades de teste resultam na utilização de um conjunto de boas práticas que acabam favorecendo o design da aplicação.

Mas será que apenas esses princípios de TDD são suficientes para modelar um software? Na verdade, os testes servem como um mecanismo para expressar o design desejado para uma classe, mas se o desenvolvedor não souber qual deve ser a solução para o problema, a utilização dessa técnica ficará restrita. O mesmo se aplica para a modelagem com o uso de UML, pois ela é apenas uma ferramenta para expressar o modelo da aplicação através de diagrama, e de nada adianta se o desenvolvedor não souber quais classes vai representar. Sendo assim, não importa qual a forma ou técnica na qual o design da aplicação está sendo definido, os padrões são importantes para que o desenvolvedor possa direcionar sua solução.

No caso do TDD, já no primeiro teste, o desenvolvedor precisa definir como a classe será criada. Nesse momento, já pode entrar em ação o conhecimento a respeito de padrões de criação. Será utilizado um construtor ou um `Static Factory Method`? É preciso utilizar um `Singleton` para a aplicação ter apenas uma instância dessa classe? Em um segundo momento, caso o processo de criação das classes for complexo o suficiente, pode-se partir para a criação de um `Builder`, o qual poderia ser desenvolvido em sua própria classe de teste.

O `Dependency Injection` é um padrão muito utilizado quando se utiliza TDD, principalmente quando realmente procura-se isolar a classe testada de suas dependências. Isso é feito para facilitar a substituição da dependência por mock objects pelo teste. Porém, como foi visto no Capítulo 7, o padrão `Service Locator` poderia também ser utilizado nessas situações sem prejuízo a modularidade.

Quando um padrão é utilizado, normalmente existem mais de uma classe envolvida, sendo cada uma com uma responsabilidade bem definida. Ao se optar por desenvolver uma solução baseada em um padrão a partir de TDD, deve-se a partir dos testes direcionar a interface da classe e de suas dependências para o uso desse padrão. Imagine um exemplo onde o padrão `Visitor` estivesse sendo utilizado. Nos testes de classes que implementam a interface visitante, que é passada como parâmetro, seria verificado se as chamadas aos métodos gerariam o efeito esperado. Nos

testes das classes visitadas, os testes verificariam se os métodos do objeto visitante passado como parâmetro seriam invocados corretamente. Esse é um exemplo no qual o padrão escolhido interfere na forma como os testes seriam criados. Mesmo assim, a interface que serve como contrato entre essas duas classes poderia ir sendo modelada de forma incremental através do TDD.

A refatoração também é uma fase do ciclo de TDD que é muito importante para o projeto do software que está sendo desenvolvido. Como no TDD o projeto ocorre de forma evolutiva e de acordo com as necessidades correntes, a reestruturação da solução que está sendo adotada é algo que é feita com uma certa frequência. Nesse contexto, como nem sempre a necessidade do uso de um padrão acontece na primeira versão da classe, é natural que eles sejam implementados a partir de refatorações. Principalmente quando uma classe começa a acumular responsabilidades ou aparece uma duplicação de código. Ter os padrões como um direcionamento para as refatorações, como foi mostrado no decorrer dos capítulos, é uma excelente forma de dirigir o design da aplicação para boas soluções de forma evolutiva.

Concluindo, por mais que o TDD favoreça a criação de classes mais coesas e desacopladas, isso não é o suficiente para o projeto de uma aplicação como um todo. Nesse contexto, a utilização de padrões, tanto para direcionar os testes, quanto para o alvo de refatorações, pode ter um impacto positivo nas soluções desenvolvidas.

## Exemplo de TDD usando o Padrão Observer

Para ilustrar a utilização de padrões com TDD, vamos imaginar o exemplo de uma aplicação de e-commerce. Nessa aplicação, ao se adicionar um produto no carrinho de compras, existem diversas funcionalidades que devem ser acionadas. Por exemplo, deve-se registrar essa informação para as estatísticas do produto, deve-se adicionar a categoria do produto nos interesses do cliente e deve-se criar uma reserva para uma unidade do produto no estoque. A inclusão de todas essas funcionalidades no carrinho de compras poderia sobrecarregar essa classe e torná-la acoplada a diversos subsistemas.

Um padrão adequado para esse tipo de problema é o `Observer`, pois o carrinho de compras poderia notificar as classes interessadas em seus eventos sem estar acoplado a elas. Nesse caso, a responsabilidade da classe que representa o carrinho de compras seria simplesmente notificar os observadores dos eventos ocorridos. Baseando-se nisso, os testes dessa classe deveriam apenas focar se as notificações estão sendo feitas na hora e com as informações corretas.

O primeiro passo seria a criação de um teste que definisse como seria a funcional-

lidade de notificação do carrinho de compras, conforme o código mostrado na próxima listagem. Observe que é instanciada uma classe, chamada de `MockObservador` para simular a existência de um observador durante o teste. Esse mock object é adicionado como um observador na classe testada e o teste verifica se, ao adicionar um produto no carrinho de compras, o observador é notificado com o produto adicionado.

```
@Test
public void testeNotificacao(){
    MockObservador mock = new MockObservador();
    CarrinhoCompras cc = new CarrinhoCompras();
    cc.addObservador(mock);

    Produto p = new Produto("Cabo HDMI", 30.0);
    cc.adicionar(p);

    assertTrue(mock.recebeuNotificacao());
    assertEquals(p, mock.produtoRecebido());
}
```

Em seguida, é preciso definir a classe `MockObservador` que está sendo utilizada no teste. Essa classe deve implementar a interface esperada pelo método `addObservador()` da classe testada, no caso `ObservadorCarrinho`. Observe que essa classe também deve possuir os métodos utilizados no teste para verificação, no caso `recebeuNotificacao()` e `produtoRecebido()`. Veja que a implementação dessa classe deve fazer sentido para o teste, ou seja, o método `notificaProduto()` exigido pela interface implementada, deve armazenar o resultado para poder ser verificado no teste.

```
public class MockObservador implements ObservadorCarrinho {

    private Produto p;

    @Override
    public void notificaProduto(Produto p){
        this.p = p;
    }
    public boolean recebeuNotificacao(){
        return p != null;
    }
}
```

```
public Produto produtoRecebido(){  
    return p;  
}  
}
```

O que é interessante notar nesse caso, é que a interface e os seus métodos são definidos no momento que está sendo criado o teste. O teste é um mecanismo que está sendo utilizado para a definição desse contrato, porém é a partir do padrão `Observer` que a solução foi construída.

Nesse caso, após a definição dos testes da classe observada, seriam criadas as classes observadoras, que implementariam a interface `ObservadorCarrinho`. Se o TDD continuasse a ser utilizado nesse desenvolvimento, seria criada uma nova bateria de testes para cada classe. Nesses testes, o método `notificaProduto()` seria invocado simulando uma notificação de uma classe que está sendo observada e verificando o comportamento esperado da classe em questão. Quando as classes estivessem desenvolvidas, um teste de integração, envolvendo o carrinho de compras e as classes observadoras, também poderia ser criado para verificar se as funcionalidades se integram de forma correta.

## 8.4 POSSO APlicar esses PADRões na ARQUITETURA?

*“Não existem regras na arquitetura para um castelo nas nuvens.”*

– G. K. Chesterton

Muitos dos padrões apresentados nesse livro também podem se aplicar em um contexto mais amplo de uma arquitetura. Nesse caso, ao invés de classes, os participantes dos padrões seriam componentes e até mesmo subsistemas de um software. Como os padrões são ideias e não estão ligados a implementações específicas, os problemas e as soluções dos padrões podem ser aplicados em uma escala diferente.

Pegando novamente como exemplo o padrão `Observer`, da mesma forma que uma classe pode precisar receber notificações de outra, um subsistema pode precisar receber atualizações de outros subsistemas remotos. Enquanto dentro de um software orientado a objetos os contratos são firmados a partir de interfaces e classes abstratas, entre sistemas remotos são utilizados protocolos de comunicação. A solução certamente precisa de uma adaptação, mas certamente a mesma ideia pode ser utilizada.

Porém, a utilização desses padrões deve ser feita com muito cuidado, pois apesar das soluções serem similares, as consequências na utilização deles podem ser bem

diferentes. Por exemplo, em uma arquitetura, quando lidamos com subsistemas que estão distribuídos em uma rede, sempre devemos considerar a possibilidade de um dos nós envolvidos podem falhar. Isso é algo que não precisamos considerar quando estamos lidando com classes implantadas em uma mesma máquina virtual. Essa é apenas uma das questões que devem ser consideradas; existem outras, como a tolerância a esse tipo de falha, questões de desempenho em relação a forma como é feita a comunicação, entre outras.

Para exemplificar essa questão, considere o padrão Proxy. Como foi visto, ele pode ser utilizado para proteger o acesso a um determinado objeto. Normalmente, quando isso é feito, é adicionada uma funcionalidade nesse acesso, como, por exemplo, alguma verificação de segurança ou validação dos dados. Dessa forma, o Proxy fica entre o cliente e a classe que está sendo acessada.

De forma similar, o Proxy também pode ser aplicado de forma mais abstrata a arquiteturas. Nessa caso, ele irá ficar como intermediário entre dois subsistemas. Ele irá disponibilizar os mesmos serviços e utilizar o mesmo protocolo que o subsistema original, para que seja transparente para o cliente a sua existência. Em outras palavras, o cliente deve acessar o Proxy como se estivesse acessando o subsistema original. A Figura 8.2 mostra a representação desse padrão implementado no contexto de uma arquitetura. Nesse caso, Proxy também poderia adicionar alguma funcionalidade nesse acesso, como funcionalidades de segurança ou registro de auditoria, da mesma forma que o padrão original.

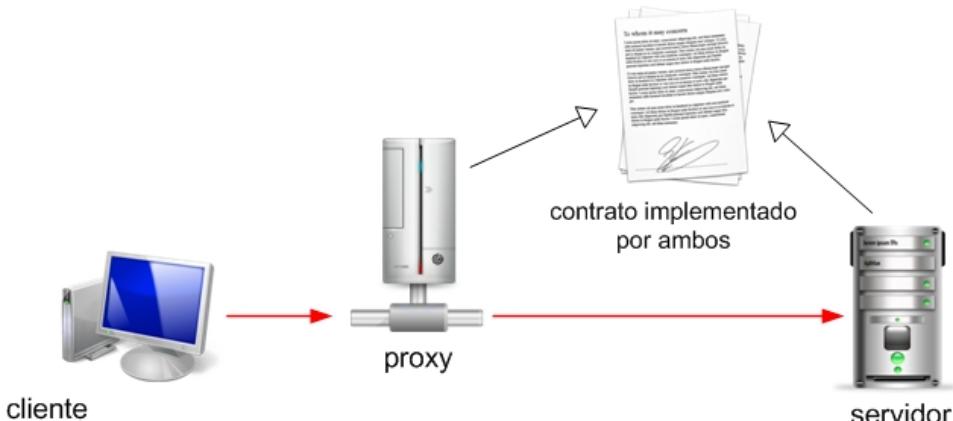


Figura 8.2: Estrutura de um Proxy para Criação de Arquitetura

Porém, como foi dito, o uso do Proxy em uma arquitetura pode ter consequências diferentes de seu uso com objetos. Por exemplo, se houver algum problema com a máquina onde estiver o Proxy, o cliente perderá acesso ao serviço, mesmo este estando funcionando. Isso adiciona um novo possível ponto de falha na arquitetura, o que é uma consequência negativa em relação a implementação com objetos. Por outro lado, o Proxy também pode ser responsável pelo roteamento das mensagens entre o cliente e o servidor, permitindo que o servidor possa mudar sua localização sem afetar os clientes, o que seria uma nova consequência positiva.

Muitas vezes, é complicado separar o que é uma solução de design do que é uma solução arquitetural. O importante é não utilizar as soluções dos padrões cegamente e sempre considerar as consequências antes da sua utilização. Lembre-se que existem soluções alternativas que podem resolver o problema de uma forma diferente, gerando um outro conjunto de consequências. Trabalhar com arquitetura é saber fazer trocas, por exemplo, sacrificando desempenho para aumentar a segurança, e escolher as soluções de forma a atender os requisitos não-funcionais do sistema.

## 8.5 COMUNIDADE DE PADRÕES

*“Fundamental a qualquer disciplina relacionada a ciência ou engenharia é uma vocabulário para expressar seus conceitos, e uma linguagem para os relacionar juntos.”*

– Brad Appleton

Após ler esse livro, você pode se perguntar: Como surgem novos padrões? Onde eu encontro novos padrões? Como eu faço para trabalhar com isso? O Hillside Group é uma organização internacional sem fins lucrativos cujo objetivo é fomentar a documentação de padrões para as mais diferentes áreas, principalmente as relacionadas ao desenvolvimento de software. Dessa forma, ela auxilia na organização de diversas conferências focadas em padrões ao redor do mundo e ajuda autores a amadurecerem seus padrões para sua publicação, seja em formato de artigos ou de livros.

As conferências de padrões, conhecidas como PLoP (Pattern Languages of Program), acontecem ao redor do mundo e recebem submissões de artigos com padrões, linguagens de padrões, ou relativos a utilização de padrões. A conferência principal acontece todo ano nos EUA e em 2013 irá completar 20 edições. Pessoalmente já tive o prazer de participar de algumas edições, e inclusive de ser o organizador do PLoP 2012, que aconteceu em Tucson, no Arizona. São conferências normalmente

para um número pequeno de pessoas (normalmente entre 40 e 60 pessoas) e que são muito diferentes de uma conferência científica tradicional.

Existem outras conferências PLoP ao redor do mundo como o EuroPLoP (Alemanha), AsianPLoP (Japão), KoalaPLoP (Austrália), VikingPLoP (países da Escandinávia), e, o estreante de 2013, GuruPLoP (Índia). Existem também conferências temáticas, onde pessoas interessadas se reunem para discutir padrões de um domínio específico, como o Meta PLoP (metaprogramação), o Scrum PLoP (padrões para documentar práticas do Scrum) e o ParaPLoP (programação paralela). No Brasil, a cada dois anos, ocorre o SugarLoafPLoP, o evento latino-americano de padrões, e nos anos intermediários um MiniPLoP, uma versão reduzida do evento.

### O QUE SÃO LINGUAGENS DE PADRÕES?

Uma linguagem de padrões é um conjunto de padrões que documentam soluções e boas práticas para um determinado domínio. Por exemplo, eu escrevi uma linguagem de padrões para a construção de frameworks que utilizam anotações e metadados. Mais do que simplesmente uma solução pontual, a linguagem de padrões possui soluções que sinergicamente podem ser combinadas e se complementam, sendo uma forma de documentar o conhecimento de modelagem sobre aquele domínio.

A submissão de um padrão para um PLoP possui um processo diferente do que a submissão de artigos para outras conferências. Ao sumeter um padrão, após uma filtragem inicial, ele é atribuído a um *shepherd* (pastor). O *shepherd* é uma especialista na área de padrões cujo papel é orientar e ajudar o autor a evoluir e amadurecer o trabalho submetido. Sendo assim, durante cerca de um mês e meio, o autor em conjunto com o *shepherd* irão interagir e cooperar para que o trabalho evolua durante esse período. Só depois disso que o artigo realmente será avaliado pelo comitê e decidido se ele será aceito ou não na conferência.

Na conferência, o autor precisa participar de uma seção chamada de *Writers Workshop*, que é bem diferente de uma seção tradicional onde o autor simplesmente apresenta o seu trabalho. Nessa seção, um grupo irá discutir o trabalho do autor levando em consideração o entendimento que obtiveram sobre o trabalho, apontando pontos de melhoria e dando sugestões de melhoria. O autor participa da discussão apenas no inicio, onde lê uma pequena parte do artigo, e no fim, onde pode tirar

dúvidas sobre algum comentário. Por mais angustiante que seja ver as pessoas discutindo sobre seu trabalho e não pode opinar ou explicar alguma coisa, esse é um processo onde se tem um grande feedback a respeito de como outras pessoas enxergam o seu trabalho.

Pessoalmente gosto muito do estilo da comunidade de padrões, que tem um espírito muito colaborativo, focado em ajudar aqueles que participam dela a documentar seus padrões da melhor forma possível. Nas conferências, além do estilo peculiar das apresentações, ainda existe toda uma cultura que valoriza muito a interação entre as pessoas. Existem, por exemplo, tradições como a de jogos para aproximar os participantes e troca de lembranças onde cada um trás normalmente algo típico de sua região ou país.

## 8.6 E AGORA?

*“Toda vez que você tem medo de fazer alguma coisa e você faz, ela te deixa mais forte. Mesmo que você falhe.”*

– Fred Bartlit

Durante esse livro foi trilhado um caminho que passou por diversos padrões que mostram e exemplificam técnicas para a criação do projeto de um software orientado a objetos. Diferentemente de quando se aprende a usar um novo framework ou uma nova API, quando se está adquirindo conhecimento, saber projetar um software é uma habilidade. E uma habilidade só se aprende de verdade com prática, sendo que o conhecimento é necessário, porém não suficiente.

Sendo assim, o próximo passo agora é tentar colocar em prática os padrões e técnicas apresentados nesse livro. Quando desenvolver um software procure olhar para ele com olhar mais crítico, tentando identificar quais padrões ele já implementa, muitas vezes de forma induzida pelos frameworks utilizados. Uma dica pode estar nos próprios nomes das classes, que com frequência “entregam” o padrão utilizado. Procure raciocinar em cima daquela solução e identificar quais foram os motivos para a utilização daquele padrão e quais benefícios ele está trazendo para o design. Aprendemos muito observando soluções prontas e abstraindo o que foi utilizado na sua concepção.

Em seguida, veja quais problemas você consegue identificar e como poderia ser feito para melhorar. Muitas vezes, um padrão deixou de ser aplicado em uma situação em que ele era adequado, porém também não é raro encontrar implementações de padrões incorretas, onde eles foram utilizados no contexto errado. A duplicação

de código, por exemplo, é um indício de que algo não está cheirando bem naquela parte do software. Veja então como algum padrão poderia te ajudar na resolução daquele problema, considerando as consequências e qual o peso de cada uma para o cenário em questão.

Adicionalmente, ao criar uma nova solução, pense em como algum padrão poderia te ajudar na resolução dos problemas envolvidos. Procure conter a ansiedade de aplicar diversos padrões em sequência sem ter alguma motivação nos requisitos. Também não deixe de combinar os padrões nas situações em que isso for adequado. Projetar um software consiste em saber equilibrar os requisitos através da combinação de soluções para os pequenos problemas, de forma a se obter um resultado final adequado. Teste suas soluções! Crie cenários baseados em histórias do cliente e veja se seu design se encaixa. Use boas práticas e mantenha o código limpo para que seja fácil de refatorá-lo frente a novas necessidades.

Finalmente, desenvolvimento de software é algo que se estuda durante toda sua carreira! Conheça novos padrões, estude como os padrões se encaixam nas novas tecnologias, e busque os padrões específicos para os domínios relacionados a sua arquitetura. Além disso, lembre-se que design de software é uma atividade criativa, então evite formulas prontas, tenha sempre uma visão crítica e utilize os padrões como uma ferramenta da sua criatividade.

Boa sorte!