

# Padrões de Projeto em Java

Henrique Duarte Moura

agosto de 2015  
versão 0.91

# H3dema

---



# Sumário

<b>1</b>	<b>Introdução</b>	<b>7</b>
1.1	Este livro . . . . .	7
1.2	O que você precisa para este livro . . . . .	8
1.3	Para quem é este livro . . . . .	8
1.4	Licença . . . . .	8
<b>2</b>	<b>Sobre os padrões de projeto</b>	<b>11</b>
2.1	Por que utilizar padrões de projeto . . . . .	12
2.2	Quer saber mais . . . . .	12
<b>3</b>	<b>Padrão: Singleton</b>	<b>13</b>
3.1	O padrão <b>Singleton</b> . . . . .	13
3.2	Quando usar o <b>Singleton</b> . . . . .	14
3.3	Uma implementação . . . . .	15
3.4	Conclusões . . . . .	16
<b>4</b>	<b>Padrão: Façade</b>	<b>19</b>
4.1	O padrão <b>Façade</b> . . . . .	19
4.2	Problemas resolvidos pelo padrão <b>Façade</b> . . . . .	20
4.3	Vantagens do padrão <b>Façade</b> . . . . .	20
4.4	Uma implementação . . . . .	21
4.5	Conclusões . . . . .	23
<b>5</b>	<b>Padrão: Iterator</b>	<b>25</b>
5.1	O padrão <b>Iterator</b> . . . . .	25
5.2	Vantagens do padrão <b>Iterator</b> . . . . .	26
5.3	Uma implementação . . . . .	26
<b>6</b>	<b>Padrão: Proxy</b>	<b>29</b>
6.1	O padrão de projeto <b>Proxy</b> . . . . .	29
6.2	Problemas resolvidos pelo padrão <b>Proxy</b> . . . . .	29
6.3	Vantagens e desvantagens do <b>Proxy</b> . . . . .	31
6.4	Uma implementação . . . . .	31
6.5	Conclusões . . . . .	34
<b>7</b>	<b>O padrão Mediator</b>	<b>35</b>
7.1	O padrão . . . . .	35
7.2	Considerações sobre o padrão . . . . .	35
7.3	Uma implementação . . . . .	35

<b>8 Padrão: Observer</b>	<b>51</b>
8.1 O padrão Observer . . . . .	51
8.2 Problemas resolvidos pelo padrão Observer . . . . .	52
8.3 Vantagens do padrão <b>Observer</b> . . . . .	52
8.4 Uma implementação . . . . .	53
8.5 Conclusões . . . . .	55
<b>9 Padrão: Decorator</b>	<b>57</b>
9.1 O padrão <b>Decorator</b> . . . . .	57
9.2 Quando usar o padrão <b>Decorator</b> . . . . .	59
9.3 Uma implementação . . . . .	59
9.4 Conclusões . . . . .	63
<b>10 Padrão: Factory</b>	<b>65</b>
10.1 Por que devemos usar este padrão ? . . . . .	65
10.2 Factory Method . . . . .	65
10.2.1 Vantagens de usar o padrão <b>Factory Method</b> . . . . .	66
10.3 Implementação com o padrão <b>Factory Method</b> . . . . .	66
10.4 Abstract Factory . . . . .	68
10.4.1 Vantagens de usar o padrão <b>Abstract Factory</b> . . . . .	69
10.5 Uma implementação usando o padrão <b>Abstract Factory</b> . . . . .	69
10.6 Abstract Factory ou Factory Method ? . . . . .	74
<b>11 Padrão: Composite</b>	<b>75</b>
11.1 O padrão <b>Composite</b> . . . . .	75
11.2 Quando usar o padrão <b>Composite</b> . . . . .	76
11.3 Uma implementação . . . . .	76
<b>12 Padrão: Command</b>	<b>79</b>
12.1 O padrão <b>Command</b> . . . . .	79
12.2 Problemas resolvidos com o padrão <b>Command</b> . . . . .	80
12.3 Vantagens e desvantagens do padrão <b>Command</b> . . . . .	80
12.4 Uma implementação . . . . .	81
12.5 Conclusões . . . . .	82
<b>13 Padrão: Strategy</b>	<b>85</b>
13.1 O padrão <b>Strategy</b> . . . . .	85
13.2 Quando usar o padrão <b>Strategy</b> . . . . .	86
13.3 Uma implementação . . . . .	86
13.4 Conclusões . . . . .	89
<b>14 Padrão: MVC</b>	<b>91</b>
14.1 Ideia principal do padrão <b>MVC</b> . . . . .	91
14.1.1 O Modelo . . . . .	91
14.1.2 A Visão . . . . .	92
14.1.3 O Controlador . . . . .	93
14.2 Vantagens do padrão <b>MVC</b> . . . . .	93
14.3 Uma implementação . . . . .	93
<b>15 Padrão: Producer-Consumer</b>	<b>103</b>
15.1 Vantagens do padrão <b>Producer-Consumer</b> . . . . .	103
15.2 Uma implementação . . . . .	104



<b>16 Padrão: Object Pool</b>	<b>107</b>
16.1 O padrão . . . . .	107
16.2 Uma implementação . . . . .	108
<b>17 Apendice A - Cripto</b>	<b>113</b>





# Capítulo 1

## Introdução

O Java é uma ótima linguagem de programação. Com Java conseguimos uma boa evolução na programação orientada a objetos. Java tem suporte a múltiplas plataformas. Ela tem todos os recursos orientados a objetos essenciais e pode ser usada para implementar *padrões de projeto*.

Um *padrão de projeto* é uma solução reutilizável e geral para um problema que ocorre comumente dentro de um determinado contexto de desenvolvimento. É uma solução de engenharia de software. Um programador enfrenta diariamente problemas que foram resolvidos muitas vezes no passado por outros programadores que eles evoluíram padrões comuns para resolvê-los. O *padrão de projeto* não é uma receita concreta para resolver um problema, como ocorre com um algoritmo. Um padrão é uma prática. É uma descrição de como resolver um problema que pode ser utilizado em diferentes situações e implementado em diferentes linguagens (como no Java). Estes padrões são “melhores práticas formalizadas” que um programador pode utilizar para resolver problemas comuns quando projeta uma aplicação ou sistema. Um padrão não é um algoritmo.

O *padrão de projeto* acelera o processo de desenvolvimento, ao proporcionar uma prática comprovada para resolver algum tipo de problema. Muitas vezes, é preferível utilizar um *padrão de projeto* do que utilizar uma solução não provada, uma vez que problemas invisíveis ocorrem frequentemente durante a implementação e a resolução de problemas imprevistos retarda o desenvolvimento de forma dramática.

Além disso, um *padrão de projeto* é uma ferramenta de comunicação entre os programadores, pois cria uma base comum de referência do trabalho realizado. É muito mais fácil dizer: “Usamos aqui o padrão **singleton**” em vez de descrever o que o código realmente faz, isto é, que a classe só permite criar um objeto.

Existe uma grande quantidade de padrões. O livro do “Gang of Four”, considerado o primeiro livro sobre o assunto, apresenta 23 padrões diferentes. Neste nosso livro apresentaremos alguns dos padrões mais comuns. A relação dos padrões é mostrada em uma tabela no capítulo 2. Se você quiser conhecer mais, recomendamos alguns livros na seção 2.2.

### 1.1 Este livro

O conteúdo mais atual deste livro pode ser encontrado online em <https://goo.gl/6F0aHq>. Neste endereço estão disponíveis também as classes utilizadas nos capítulos. Este livro é composto de 17 capítulos. O capítulo 1 apresenta informações sobre este livro. O próximo capítulo apresenta uma introdução aos *padrões de projeto*. A partir do capítulo 3, cada capítulo tratará de um tipo de *padrão de projeto*:

No capítulo 3 apresentamos um padrão que garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto, denominado **Singleton**. O *padrão de projeto* denominado **Facade** é um classe que disponibiliza uma interface simplificada para uma das funcionalidades de uma API. Este padrão é mostrado no capítulo 4. No capítulo

5 apresentamos o *padrão de projeto* denominado **Iterator** que é um objeto que nos permite examinar uma coleção de objetos, como por exemplo listas. Para controlar o acesso a um objeto, podemos utilizar um substituto denominado **Proxy**, apresentado no capítulo 6. No capítulo 7 apresentamos um padrão para aumentar re-uso e diminuir acoplamento denominado **Mediator**. No capítulo 8 apresentamos o *padrão de projeto* denominado **Observer**. Este padrão define uma dependência um-para-muitos entre objetos de modo que quando um objeto observado muda o estado, todos seus dependentes são notificados e atualizados automaticamente. Quando queremos adicionar um comportamento a um objeto já existente em tempo de execução utilizamos o padrão **Decorator** apresentado no capítulo 9. No capítulo 10 apresentamos dois padrões denominados **Factory Method** e **Abstract Factory**. Estes padrões permitem delegar a criação de classes para uma classe que funciona como uma “fábrica”. Quando desejamos que um objeto represente um conjunto composto por objetos semelhantes, utilizamos o padrão do capítulo 11 denominado **Composite**. No capítulo 12 apresentamos o *padrão de projeto* denominado **Command** que permite encapsular uma solicitação como um objeto. No capítulo 13 apresentamos o *padrão de projeto* denominado **Strategy**. Este padrão nos auxilia a representar uma operação a ser realizada sobre os elementos de uma estrutura de objetos. Apresentaremos ainda um padrão que não foi apresentado no GoF. Mostramos este padrão no capítulo 14. Ele é chamado **MVC** (Model-view-controller) facilitando a reusabilidade de código e a separação dos dados da sua apresentação. No capítulo 15, mostramos um padrão que trata de concorrência de processos, denominado **Producer-Consumer**. Apresentamos no capítulo 16 um padrão que reaproveita objetos, denominado **Object Pool**.

## 1.2 O que você precisa para este livro

Você vai necessitar de uma instalação Java 1.7 ou superior. Nos nossos exemplos são compilados e rodados utilizando o Java 8. Esta versão pode ser baixada do site do Java e instalada no Windows, Linux ou MAC. Você precisará da versão denominada JDK que possui o compilador `javac`. Para criar applets e aplicações Java, você precisa, no mínimo, das ferramentas de desenvolvimento fornecidas pelo JDK. O JDK inclui o Java Runtime Environment, o compilador Java e as APIs Java. Neste livro consideramos que você tenha baixado o Java SE Development Kit 8 Downloads.



Nossos exemplos foram rodados em um computador com Ubuntu LTE 14.04 com Java 8.

## 1.3 Para quem é este livro

Este livro é para desenvolvedores com um conhecimento intermediário de Java que querem tornar o aprendizado de *padrões de projeto* seu próximo passo em sua carreira de programadores.

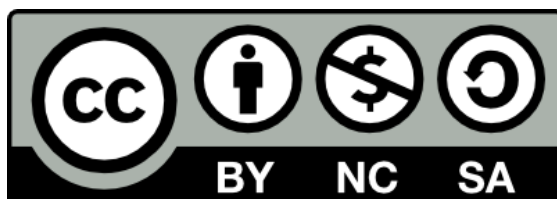
## 1.4 Licença

Este trabalho está licenciado sob uma Licença Creative Commons Atribuição-NãoComercial-Compartilha Igual 4.0 Internacional. Para ver uma cópia desta licença, visite <http://creativecommons.org/licenses/by-nc-sa/4.0/>.

Se você gostou deste livro e quer me recompensar monetariamente por ele, você fazer uma doação via paypal. O link para a doação é este que está no pdf.









## Capítulo 2

# Sobre os padrões de projeto

O nome do padrão é um identificador que podemos usar para descrever um problema de projeto, suas soluções e consequências. O uso do padrão nos permite projetar em um nível mais alto de abstração. Segundo GoF, a nomeação de um padrão imediatamente aumenta o nosso vocabulário de design. Ter um vocabulário para padrões nos permite falar sobre eles com os nossos colegas, reduzindo as confusões. O uso de padrões torna mais fácil para pensar em projetos e comunicá-las e seus *trade-offs* para os outros.

A nomes de *padrão de projeto* e sua descrição identifica os principais aspectos de uma estrutura de projeto comum que o tornam útil para a criação de um projeto orientado a objetos reutilizáveis. O *padrão de projeto* identifica as classes participantes e instâncias, os seus papéis e colaborações e a distribuição de responsabilidades. Cada *padrão de projeto* trata de um problema de projeto em particular. O padrão descreve quando ele pode ser aplicado, considera as restrições de projeto, as consequências e os trade-offs de seu uso.

A solução descreve os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações. A solução não descreve um desenho concreto particular ou aplicação, porque é um padrão como um modelo que pode ser aplicado em muitas situações diferentes. Em vez disso, o padrão fornece uma descrição abstrata de um problema de projeto e como um arranjo geral de elementos resolve o problema de projeto.

Por Propósito				
		Criação	Estruturais	Comportamentais
Por escopo	Para classe	Factory Method	Adapter (classe)	Interpreter Template Method
	Para objetos	Abstract Factory Builder Prototype Singleton	Adapter (objeto) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tabela 2.1: Padrões de projetos apresentados pelo GoF

Não iremos tratar de todos os padrões apresentados pelo GoF. Este livro concentrará nos itens em vermelho. Além dos padrões GoF mostrados na tabela 2.1, mostramos neste livro ainda três padrões adicionais - **MVC**, **Producer-Consumer** e **Object Pool**. Se vocês lerem os livros POSA identificarão muitos outros além destes apresentados neste livro.

## 2.1 Por que utilizar padrões de projeto

Os *padrões de projeto* ajudam-nos a analisar as áreas mais abstratas de um programa, fornecendo soluções concretas e bem testadas. Os *padrões de projeto* permite que nós possamos escrever nosso código mais rápido e proporcionam uma imagem mais clara de como o projeto está sendo implementando. O uso dos *padrões de projeto* incentivam reutilização de código. Os padrões nos permitem acomodar a mudança através do fornecimento de mecanismos bem testados para a delegação/composição e outras técnicas de reutilização não baseados em herança. Ao usarmos os *padrões de projeto*, esperamos que nosso código seja mais legível e de mais fácil manutenção. Os *padrões de projeto* proporcionam uma linguagem comum - como um jargão para programadores.

## 2.2 Quer saber mais

O uso de padrões originou-se com Christopher Alexander que criou o conceito para Arquitetura na década de 1970. Kent Beck e Ward Cunningham experimentaram a ideia de aplicar padrões a programação - especificamente a linguagem de padrões - e apresentaram seus resultados na conferência OOPSLA de 1987.

Os padrões de projeto ganharam popularidade com a publicação do livro da chamada gangue dos quatro (gang of four ou GoF). Estes quatro autores publicaram o livro *Design Patterns: Elements of Reusable Object-Oriented Software* em 1994.

Sugerimos para quem quer se aprofundar neste tema que leia os livros relacionados abaixo. O primeiro é o livro do GoF. Os cinco seguintes são os livros POSA.

Recomendo a leitura do “Head First Design Patterns” depois de ler o nosso livro. Ele permite uma transição mais tranquila para o livro do GoF - “Gang of Four”. Depois de ler o GoF, para quem quer realmente se aprofundar, leia a série POSA.

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons.
- Schmidt, Douglas C.; Stal, Michael; Rohnert, Hans; Buschmann, Frank (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons.
- Kircher, Michael & Jain, Prashant. (2004) *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. John Wiley & Sons.
- Buschmann, Frank; Henney, Kevlin & Schmidt, Douglas C. (2007) *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. John Wiley & Sons.
- Buschmann, Frank; Henney, Kevlin & Schmidt, Douglas C. (2007) *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*. John Wiley & Sons.
- Freeman, Eric T; Robson, Elisabeth; Bates, Bert; Sierra, Kathy (2004). *Head First Design Patterns*. O'Reilly Media.
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
- Hohpe, Gregor; Woolf, Bobby (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.



## Capítulo 3

# Padrão: Singleton

Há situações em que você precisa para criar apenas uma instância de um objeto em todo o tempo de vida de um programa. Esta pode ser uma instância de uma classe que você criou, uma lista, uma conexão de acesso a um banco de dados, etc. Por exemplo a criação de uma segunda conexão a um banco de dados pode ser indesejável, pois gasta recursos adicionais do gerenciador de banco de dados ou até mesmo licença de uso. Uma segunda instância pode ainda resultar numa erros lógicos ou mau funcionamento do programa. O padrão de projeto que permite que você para criar apenas uma instância de dados é chamada **Singleton**.

### 3.1 O padrão Singleton

Para uma classe seja um **Singleton**, devemos garantir que haverá apenas uma instância na aplicação. Devemos ter também um ponto de acesso a esta instância. Mas como garantir que haverá apenas uma instância? Para criar uma instância de uma classe, devemos chamar o seu construtor. Assim, para resolvermos o problema, devemos restringir o acesso ao construtor, tornando-o um método privado. Podemos desta forma, mediante um método público, realizar o controle da instanciação, de modo que a criação só possa ser feita uma vez.

Listagem 3.1: Singleton.java

```
1 class Singleton {  
2  
3     private static Singleton singleton = NULL;  
4  
5     private Singleton() {}  
6  
7     public static Singleton getSingleton() {  
8         if (NULL == singleton) singleton = new Singleton();  
9         return singleton;  
10    }  
11 }
```

O código acima pode ser problemático em ambientes *multi-threaded*, ou seja, ele não é uma solução *thread-safe*. Se uma *thread* chamar o método `getSingleton()` e for interrompida antes de realizar a instanciação, uma outra *thread* poderá chamar o método e realizar a instanciação. Neste caso, duas instâncias serão construídas, o que fere os requisitos do **Singleton**. Uma solução para este problema seria utilizar o atributo *synchronized* em `getSingleton()`, como visto no código abaixo, para que uma outra *thread* não possa acessá-lo até que a *thread* que o acessou pela primeira vez tenha terminado de fazê-lo.

Listagem 3.2: Singleton.java

```
1 class Singleton {
```

```

2
3     private static Singleton singleton = NULL;
4
5     private Singleton() {}
6
7     public static synchronized Singleton getSingleton() {
8         if (NULL == singleton) singleton = new Singleton();
9         return singleton;
10    }
11 }

```

Contudo existe ainda um problema com o código acima. O *synchronized* é uma operação bastante dispendiosa. Estima-se que métodos sincronizados sejam cerca de cem vezes mais lentos que métodos não sincronizados. Uma alternativa simples, rápida e *thread-safe* é a instanciação do **Singleton** assim que ele for declarado, como na terceira versão do nosso código logo abaixo.

Listagem 3.3: Singleton.java

```

1 class Singleton {
2
3     private static Singleton singleton = new Singleton();
4
5     private Singleton() {}
6
7     public static Singleton getSingleton() { return singleton; }
8 }

```

A figura 3.1 mostra o diagrama de classes de um *Singleton*. É constituído de uma única classe. Esta classe possui um atributo estático que contém a única instância gerada. O construtor da classe é protegido, assim não pode ser chamado fora da classe. A classe *Singleton* tem um método *getSingleton* (alguns autores usam *getInstance*) que retorna a instância do objeto criado e se não existir faz a criação da primeira e única instância. No diagrama mostrado na figura 3.2 vemos como o objeto da classe é criado.

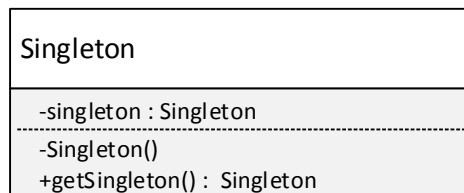


Figura 3.1: Diagrama de classe de um Singleton

## 3.2 Quando usar o Singleton

O padrão **Singleton** é o melhor candidato quando:

- precisamos controlar o acesso simultâneo a um recurso compartilhado
- precisamos de um ponto global de acesso para o recurso de múltiplas partes ou de diferentes partes do sistema
- precisamos ter apenas um objeto daquela classe

Alguns exemplos típicos de uso de um **Singleton** são:



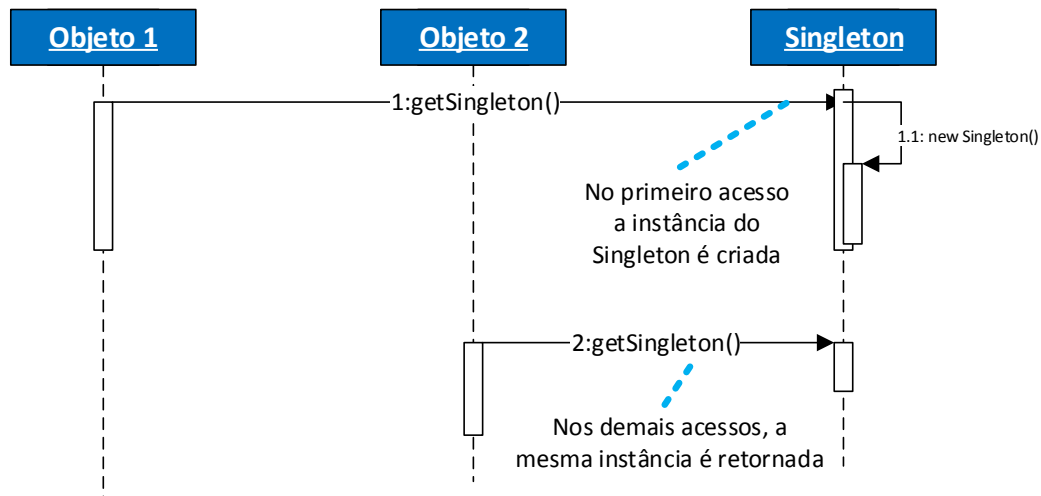


Figura 3.2: Diagrama de Sequência de um Singleton

- A classe de log e suas subclasses  $\Rightarrow$  precisamos de um ponto global de acesso para a classe que gera mensagens para log, não queremos que todos os logs fiquem em um único repositório
- O *spooler* de impressão  $\Rightarrow$  a nossa aplicação deve ter apenas uma única instância do *spooler*, a fim de evitar que um pedido para o mesmo provoque um conflito de recursos, como dois documentos sendo impressos misturados
- A gestão de uma conexão com um banco de dados
- O gerenciador de arquivos  $\Rightarrow$  o sistema operacional fornece o acesso aos arquivos controlando o acesso dos usuários
- A recuperação e armazenamento de informações em arquivos de configuração externos
- Um **Singleton** somente leitura para armazenar alguns estados globais - idioma do usuário, tempo, zona de tempo, caminho de aplicação, etc

### 3.3 Uma implementação

Vamos ver um exemplo simples onde precisamos de gerar um número em sequência independente da quantidade de *thread* ou aplicações que estejam chamando.

Listagem 3.4: Sequencia.java

```

1 class Sequencia {
2
3     private static Sequencia instancia = new Sequencia();
4     private static int num = 0;
5
6     /** protegendo o construtor */
7     private Sequencia() {}
8

```



```
9  /** devolve a instancia única criada */
10 public static Sequencia getInstancia() {return instancia; }
11
12 /** obtem um numero em sequencia */
13 public static int getNum() { return num++; }
14
15 }
```

Listagem 3.5: Teste.java

```
1  import java.lang.Thread;
2
3  /** classe interna para executarmos em threads */
4  class Teste implements Runnable {
5      int id;
6      int execucoes = 10;
7
8      Teste(int id) {
9          this.id = id;
10     }
11     /** a interface Runnable precisa que implementemos este metodo */
12     public void run() {
13         Sequencia seq = Sequencia.getInstancia();
14         for(int i = 0; i < execucoes; i++) {
15             System.out.printf("Thread %d - seq %d\n", id, seq.getNum());
16             try {
17                 Thread.sleep(50);
18             } catch (java.lang.InterruptedException e ) {
19                 // não estamos verificando nada !!!
20             }
21         }
22     }
23 }
```

Listagem 3.6: TesteSequencia.java

```
1  import java.lang.Thread;
2
3  class TesteSequencia {
4
5      public static void main(String[] args) {
6
7          for(int i = 0; i < 10; i++) {
8              Teste t = new Teste(i);
9              new Thread(t).start();
10          }
11      }
12  }
```

Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresenta na figura 3.3.

## 3.4 Conclusões

Além de ser um dos padrões mais simples, o **Singleton** é também um dos mais criticados e mal usados. Uma situação em que realmente é necessário que exista apenas uma instância de uma classe é difícil e o seu uso pode levar a muitos problemas. O uso abusivo de **Singletons** leva a soluções onde a dependência entre objetos é muito forte e a testabilidade é fraca. Um outro problema é que os **Singletons** dificultam o *hot redeploy* por permanecerem em cache, bloqueando mudanças de configuração. Por esses e outros problemas, deve-se ter cuidado com o uso abusivo de **Singletons**.





```
henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/singleton
h3dema@casa-desktop:singleton$ javac Sequencia.java TesteSequencia.java
h3dema@casa-desktop:singleton$ java TesteSequencia
Thread 1 - seq 1
Thread 9 - seq 9
Thread 8 - seq 8
Thread 7 - seq 7
Thread 6 - seq 6
Thread 5 - seq 5
Thread 2 - seq 3
Thread 4 - seq 4
Thread 3 - seq 2
Thread 0 - seq 0
Thread 1 - seq 10
Thread 8 - seq 12
Thread 9 - seq 11
Thread 3 - seq 18
Thread 4 - seq 17
```

Figura 3.3: Execução de TesteSequencia.





## Capítulo 4

# Padrão: Façade

Às vezes, um subsistema de classes e objetos torna-se tão complexo que é difícil entender como ele funciona. Ele se torna ainda mais difícil de entender, como usar este sistema? E qual a forma de diminuir a sua complexidade? O padrão de projeto da **Façade** foi projetado para resolver este problema. O padrão **Façade** oculta toda a complexidade de uma ou mais classes através de uma fachada (facade em inglês). A intenção desse padrão é simplificar uma interface.

### 4.1 O padrão Façade

O padrão de projeto **Façade** fornece uma interface unificada, em vez de um conjunto de interfaces de alguns subsistema complexo. **Façade** cria uma interface de alto nível que simplifica uso do subsistema. Esse padrão de projeto agrega classes que implementam a funcionalidade do subsistema sem escondê-los completamente. O padrão **Façade** funciona, basicamente, como um wrapper. Ele não deve adicionar qualquer nova funcionalidade. Ele deve apenas simplificar o acesso a um sistema.

Simplificando, o padrão **Façade** é um objeto acumulando um método em um nível bastante elevado de abstração para trabalhar com um subsistema complexo, como podemos ver na figura 4.1.

Com o padrão **Façade** podemos simplificar a utilização de um subsistema complexo apenas implementando uma classe que fornece uma interface única e mais razoável, porém se desejássemos acessar as funcionalidades de baixo nível do sistema isso seria perfeitamente possível. É importante ressaltar que o padrão **Façade** não “encapsula” as interfaces do sistema, o padrão Façade apenas fornece uma interface simplificada para acessar as suas funcionalidades. É importante compreender que o cliente não é privado de um acesso de baixo nível para as classes do subsistema se ele ou ela quer, é claro. Façade simplifica algumas operações com o subsistema, mas faz não impõe o recurso para o cliente.

Na concepção de sistemas complexos muitas vezes utilizamos o princípio da decomposição, do ponto de vista arquitetônico. Desta forma um sistema complexo é decomposto em subsistemas menores e mais simples. Esses subsistemas são muitas vezes desenvolvidos por diferentes equipes de desenvolvedores. Quando estes subsistemas são integrados, um problema que surge é um acoplamento forte. Isto é, um subsistema A depende muito da forma como foi feito o subsistema B. Se alguma mudança for necessária no subsistema B, pode ser necessário que o subsistema A tenha que sofrer profundas modificações em seu código a fim de trabalhar com o código modificado do subsistema B. Se usarmos o padrão **Façade**, os subsistemas podem se comunicar através da classe **Façade** e sua interface. E se interfaces **Façade** de A e B permanecem as mesmas, os códigos por trás das fachadas podem ser modificados sem afetar os outros módulos. No nosso exemplo de implementação, na seção 4.4, podemos trocar o funcionamento da classe *HD*, *CPU*

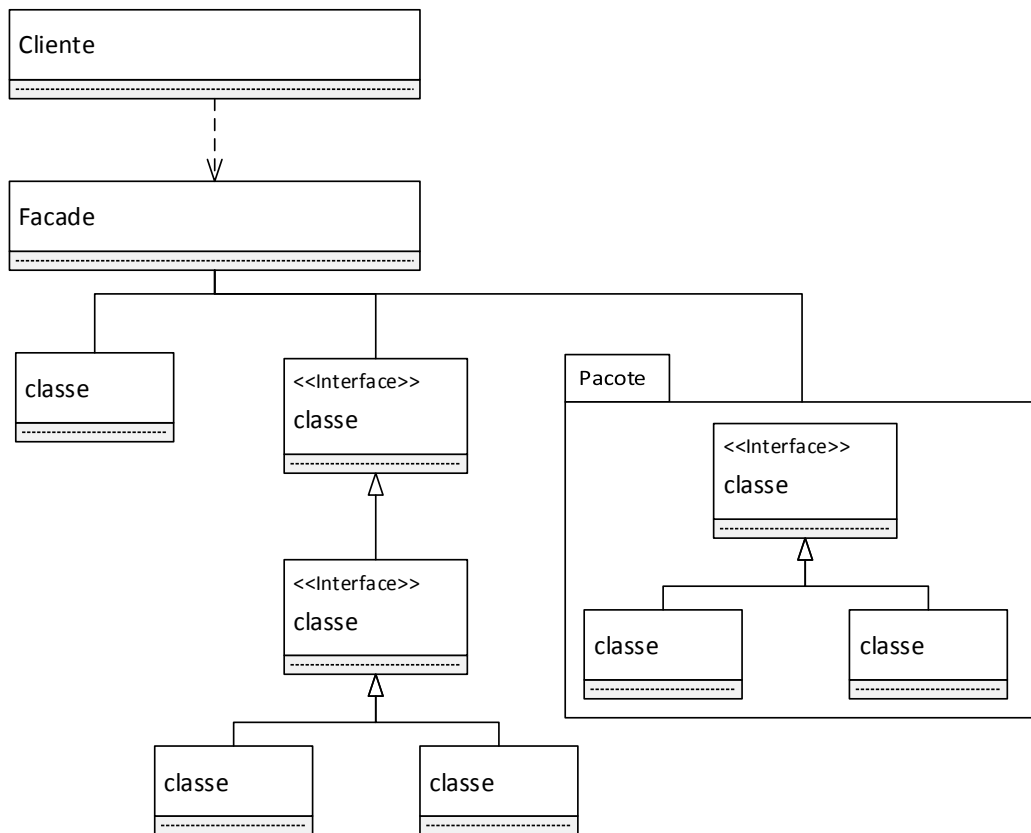


Figura 4.1: Uso de Façade para “esconder” um subsistema complexo.

ou *Memoria*, sem que a classe *Cliente* tenha que sofrer qualquer alteração - desde que a interface *ComputadorFacade* permaneça igual.

## 4.2 Problemas resolvidos pelo padrão Façade

O padrão **Façade** soluciona diversos problemas, como por exemplo os seguintes:

- Torna uma biblioteca de software fácil de usar e de testar, pois o padrão **Façade** possui métodos convenientes para tarefas comuns
- Reduz a dependência do uso de código externo, que estará relacionado somente ao código do **Façade**, mas não manterá qualquer relação com o código do cliente
- Fornece uma API melhor e mais clara para o código do cliente

## 4.3 Vantagens do padrão Façade

Vamos olhar para as vantagens do padrão **Façade**:

- Mantém acoplamento entre cliente e subsistemas



- fornece uma interface para um conjunto de interfaces em um subsistema (sem mudá-las)
- envolve um subsistema complicado com uma interface mais simples
- o subsistema ganha flexibilidade de implementação e os clientes ganham simplicidade

## 4.4 Uma implementação

No mundo físico, nós sempre nos deparamos com **Façades**. Por exemplo, quando ligamos o computador, o sistema operacional esconde todo o trabalho interno do computador porque o sistema operacional oferece uma interface simplificada para usar a máquina.

Listagem 4.1: CPU.java

```
1 public class CPU {
2
3     public void start() {
4         System.out.println("inicialização inicial");
5     }
6
7     public void execute() {
8         System.out.println("executa conteúdo residente em memória no processador");
9     }
10
11     public void load() {
12         System.out.println("carrega registrador");
13     }
14
15     public void free() {
16         System.out.println("libera registradores");
17     }
18 }
```

Listagem 4.2: HD.java

```
1 public class HD {
2
3     private static int idHD = 0;
4
5     private boolean bootable = false;
6     private int id;
7
8     public HD() { id = idHD++; }
9
10    public HD(boolean bootable) {
11        id = idHD++;
12        this.bootable = bootable;
13    }
14
15    public boolean isBootable() {
16        return bootable;
17    }
18
19    public String read(int posicao, int size) {
20        System.out.println("Lê "+size+" B de dados do HD "+id+" na posição ["+"+posicao+"+]");
21        return "Bootloader";
22    }
23
24    public void write(int posicao, String info) {
25        System.out.println("Escreve dados ["+"+info+""] no HD "+id+" na posição ["+"+posicao+"+]");
26    }
27 }
```



Listagem 4.3: Memoria.java

```
1 public class Memoria {
2
3     public void load(int posicaoMem, String info) {
4         System.out.printf("carrega_dados_%s_na_memória%d\n", info, posicaoMem);
5     }
6
7     public void free(int posicaoMem, String info) {
8         System.out.println("libera_dados_da_memória["+posicaoMem+"]");
9     }
10 }
```

Listagem 4.4: ComputadorFacade.java

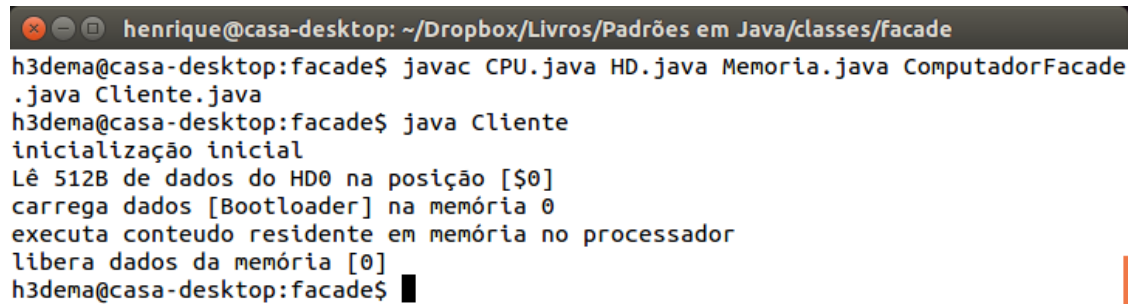
```
1 import java.util.List;
2 import java.util.ArrayList;
3
4 public class ComputadorFacade {
5     private final int BOOT_SECTOR = 0;
6     private final int SECTOR_SIZE = 512;
7     private final int BOOT_ADDRESS = 0;
8
9     private CPU cpu = null;
10    private Memoria memoria = null;
11    private List<HD> hds;
12
13    public ComputadorFacade() {
14        this(new CPU(), new Memoria(), new HD(true));
15    }
16
17    public ComputadorFacade(CPU cpu, Memoria memoria, HD hd0) {
18        this.cpu = cpu;
19        this.memoria = memoria;
20        hds = new java.util.ArrayList<>();
21        hds.add(hd0);
22    }
23
24    public void ligarComputador() {
25        String hdBootInfo;
26        cpu.start();
27        for(HD hd : hds) {
28            hdBootInfo = hd.read(BOOT_SECTOR, SECTOR_SIZE);
29            memoria.load(BOOT_ADDRESS, hdBootInfo);
30            cpu.execute();
31            memoria.free(BOOT_ADDRESS, hdBootInfo);
32        }
33    }
34 }
```

Listagem 4.5: Cliente.java

```
1 class Cliente {
2     /** utiliza o facade para simular um computador */
3
4     public static void main(String[] args) {
5         ComputadorFacade comp = new ComputadorFacade();
6         comp.ligarComputador();
7     }
8
9 }
```

Nosso exemplo somente imprimiu algumas mensagens na tela, porém poderia ser um projeto muito mais completo de simulação do funcionamento de um computador real. Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresenta na figura 4.2.



A terminal window with a dark background and light text. The title bar shows a window icon, a close button, and the text 'henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/facade'. The terminal content shows the compilation and execution of Java code for the Facade pattern. The commands are: 'javac CPU.java HD.java Memoria.java ComputadorFacade.java Cliente.java' and 'java Cliente'. The output shows the initialization of the system, reading data from the HD, loading it into memory, executing it on the processor, and finally liberating the memory.

```
henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/facade
h3dema@casa-desktop:facade$ javac CPU.java HD.java Memoria.java ComputadorFacade
.java Cliente.java
h3dema@casa-desktop:facade$ java Cliente
inicialização inicial
Lê 512B de dados do HD0 na posição [$0]
carrega dados [Bootloader] na memória 0
executa conteudo residente em memória no processador
libera dados da memória [0]
h3dema@casa-desktop:facade$ █
```

Figura 4.2: Execução de ClienteFacade.

## 4.5 Conclusões

**Façade** é usado quando é necessário para proporcionar uma interface simples de um complexo subsistema. **Façade** fornece flexibilidade para subsistema, porque toda a interação com o cliente passa através do **Façade**. Ele reduz a dependência de biblioteca externo que são usado dentro do **Façade**, mas não relacionadas com o código do cliente.







## Capítulo 5

# Padrão: Iterator

Padrão **Iterator** é um padrão comumente utilizado no ambiente de programação Java. Diversas classes fornecidas na API do Java utilizam este padrão. Ele é usado para obter uma maneira de acessar os elementos de uma coleção de objetos de forma sequencial, sem qualquer necessidade de conhecer a sua representação subjacente.

Padrão **Iterator** não serve apenas para seguir através de uma coleção. Nós podemos fornecer diferentes tipos de iteradores baseados em nossos requisitos. Este padrão esconde a implementação real do percurso através dos dados da coleção. O cliente precisa utilizar somente os métodos do iterador.

### 5.1 O padrão Iterator

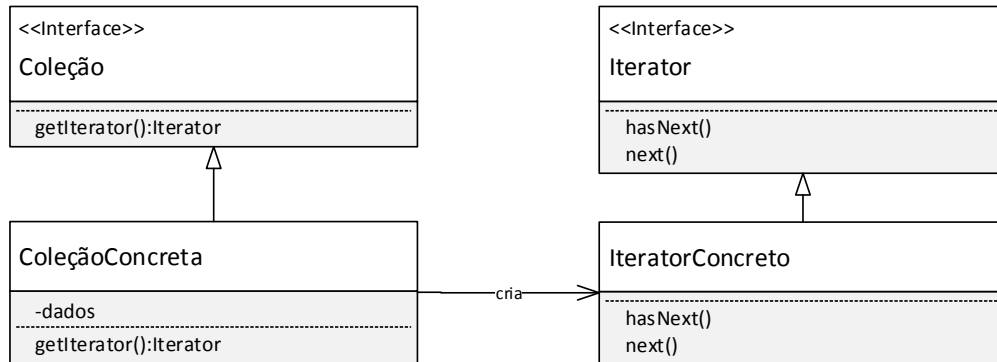
O padrão **Iterator** utiliza um diagrama de classes como o mostrado na figura 5.1. Identificamos quatro classes nestes diagrama:

- A classe *Coleção* define uma interface para a criação do objeto *Iterator*.
- A classe *ColeçãoConcreta* implementa a interface *Coleção*. O método que implementa retorna uma instância da classe *IteratorConcreto*.
- A classe *Iterator* define a interface para acesso aos elementos da coleção, e
- A classe *IteratorConcreto* implementa a *Iterator*. Ela permite acesso aos dados da coleção em *ColeçãoConcreta*. Esta classe mantém ainda a posição atual da leitura.

A API do Java utiliza iterator em diversas classes que são coleções. Vamos ver um exemplo usando a interface List. As classes derivadas de List devem implementar o método iterator() que retorna um iterator que permite acessar os objetos da lista. Uma das classes concretas que implementam List é ArrayList, que usaremos no exemplo.

Listagem 5.1: Exemplo usando Iterator com uma classe da API Java

```
1 import java.util.ArrayList;
2 import java.util.Arrays;
3 import java.util.Iterator;
4 import java.util.List;
5
6 class ExemploJavaIterator {
7
8     public static void main(String[] args) {
9
10         /** vamos começar criando um lista */
11         List<String> frutas = new ArrayList<>(
```

Figura 5.1: Diagrama de classe típico de um padrão **Iterator**

```

12     Arrays.asList("maça", "banana", "pera", "laranja", "goiaba"));
13     frutas.add("abacaxi");
14
15     System.out.println("1- usando Iterator no loop FOR");
16     for(String fruta : frutas)
17         System.out.println(fruta);
18
19     System.out.println("2- usando a classe Iterator");
20     Iterator<String> itFrutas = frutas.iterator();
21     while(itFrutas.hasNext())
22         System.out.println(itFrutas.next());
23 }
24 }
  
```

Note como inicializamos o nosso `ArrayList`. `Arrays.asList()` permite passar um conjunto de strings que formam a lista inicial. Logo a seguir usamos o método `.add()` para acrescentar mais um elemento à lista. Neste exemplo mostramos duas formas de ver os elementos da lista. Em um loop For podemos usar um `Iterator` de uma forma dissimulada. Não precisamos criar um `Iterator` como na segunda forma que necessita da classe `Iterator`. Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresenta na figura 5.2.

## 5.2 Vantagens do padrão **Iterator**

- Padrão **Iterator** é útil quando você deseja fornecer uma maneira padrão para o usuário interagir com uma coleção de dados e, ao mesmo, esconder do cliente a lógica de implementação do programa,
- A lógica para a iteração está embutido na própria coleção. Isto auxiliar o programa cliente a buscar os dados de forma padronizada.

## 5.3 Uma implementação

Observe que utilizamos a implementação do iterador com uma classe interna. Desta forma a implementação não pode ser usada por qualquer outra coleção. A utilização da classe interna permite acesso à coleção que muitas vezes é implementada como um conjunto de dados privados.



```

henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/iterator
h3dema@casa-desktop:iterator$ javac ExemploJavaIterator.java
h3dema@casa-desktop:iterator$ java ExemploJavaIterator
1 - usando Iterator no loop FOR
maça
banana
pera
laranja
goiaba
abacaxi
2 - usando a classe Iterator
maça
banana
pera
laranja
goiaba
abacaxi
h3dema@casa-desktop:iterator$ █

```

Figura 5.2: Execução de ExemploJavaIterator.

Listagem 5.2: Definindo a interface Iterator.java

```

1  /**
2   * define a interface que será implementada como Iterator
3   * precisamos de 2 métodos, um que identifica se existe um próximo elemento e
4   * outro para obter o próximo elemento
5   */
6  public interface Iterator {
7      /* verifica se existe novo elemento */
8      public boolean hasNext();
9      /* retorna o próximo, null se não houve */
10     public Object next();
11 }

```

Listagem 5.3: Colecao.java

```

1  /**
2   * interface da coleção de objetos que usará Iterator
3   */
4  public interface Colecao {
5      /* retorna o Iterator para a coleção de objetos */
6      public Iterator getIterator();
7  }

```

Listagem 5.4: ListaNomes.java

```

1  public class ListaNomes implements Colecao {
2
3      /* mantém a coleção de nomes neste array */
4      private String nomes[];
5
6      /* construtor */
7      public ListaNomes(String nomes[]) {
8          this.nomes = nomes;
9      }
10
11     /* implementando o método abstrato de Container */
12     @Override
13     public Iterator getIterator() {
14         return new LocalIterator();
15     }
16 }

```



```

15     }
16
17     /* classe privada que implementa o Iterator */
18     private class LocalIterator implements Iterator {
19
20         int index = 0;
21
22         @Override
23         public boolean hasNext() {
24             if(index < nomes.length){ return true; }
25             return false;
26         }
27
28         @Override
29         public Object next() {
30             if(this.hasNext()){
31                 return nomes[index++];
32             }
33             return null;
34         }
35     }
36 }

```

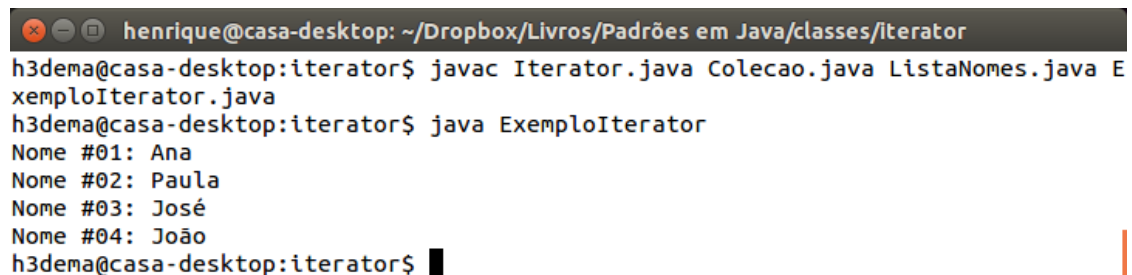
Listagem 5.5: ExemploIterator.java

```

1 public class ExemploIterator {
2
3     public static void main(String[] args) {
4         ListaNomes nomes = new ListaNomes(new String[]{"Ana", "Paula", "José", "Joã
5             o"});
6         int i = 1;
7         for(Iterator iter = nomes.getIterator(); iter.hasNext();){
8             String nome = (String)iter.next();
9             System.out.printf("Nome %02d: %s\n", i++, nome);
10        }
11    }

```

Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresenta na figura 5.3.



```

henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/iterator
h3dema@casa-desktop:iterator$ javac Iterator.java Colecao.java ListaNomes.java Ex
emploIterator.java
h3dema@casa-desktop:iterator$ java ExemploIterator
Nome #01: Ana
Nome #02: Paula
Nome #03: José
Nome #04: João
h3dema@casa-desktop:iterator$

```

Figura 5.3: Execução de ExemploIterator.



## Capítulo 6

# Padrão: Proxy

Às vezes você precisa trabalhar com um grande objeto tão grande que é melhor adiar a sua criação até o momento em que ele é realmente usado para salvar um pouco de memória e tempo. Quando ele é criado, é melhor não para criá-la novamente em cada nova solicitação, mas o uso o objeto criado anteriormente e crie uma nova referência. Quando todas as partes do código ter concluído o trabalho com ele, é necessário que alguma memória será liberado tão logo possível. Isso significa que precisamos para contar as referências ao objeto pesado, e para implementar isso, precisamos de um intermediário que faz todo esse trabalho intermediário. Um **Proxy** é a solução para este problema.

Um **Proxy** é um padrão de design que ajuda a separar o código do cliente do objeto que o código do cliente usa. Isso significa que o código de cliente usará um **Proxy** substituto objeto que age como um objeto real; No entanto, o objeto substituto vai delegar tudo chamadas para o objeto real.

O exemplo descrito anteriormente é conhecido como inicialização lenta. Você adiar a objeto de inicialização até que você realmente precisa dele. Mas não é o único caso de uso de um **Proxy**. Proxies ajudar a implementar o registro, facilitam as conexões de rede, controle acesso a objetos compartilhados, implementar a contagem de referência, e tem muitos outros usos.

### 6.1 O padrão de projeto Proxy

Um **Proxy** é uma classe, funcionando como uma interface para outra classe que tem a mesma interface como o **Proxy**. O código do cliente instancia e trabalha diretamente com o **Proxy**, Considerando que, o **Proxy** contém a instância e delegados todas as chamadas ao objeto real para ele, acrescentando própria lógica do **Proxy**. O **Proxy** serve como uma interface com muitas coisas: uma conexão de rede, um grande objeto na memória, um ficheiro, ou algum outro recurso que é dispendioso ou impossível duplicar. No diagrama da figura 6.1, a classe Proxy e TrabalhoReal são herdadas da mesma interface denominada Trabalho. A classe Cliente utiliza Proxy. A classe Proxy delega a execução para TrabalhoReal. Note que poderíamos utilizar uma classe TrabalhoReal2, sem alterar o comportamento para o cliente.

### 6.2 Problemas resolvidos pelo padrão Proxy

O padrão **Proxy** resolve os seguintes problemas que poderão surgir se manter objetos acoplamento forte:

- fornece um espaço reservado para um outro objeto para controlar o acesso a ele
- usa um nível extra de indireção ao apoio distribuído, controlado, ou de acesso inteligente

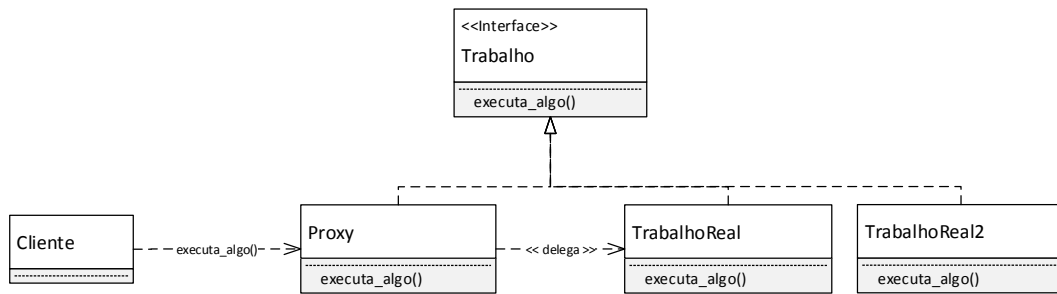


Figura 6.1: Uso de Proxy.

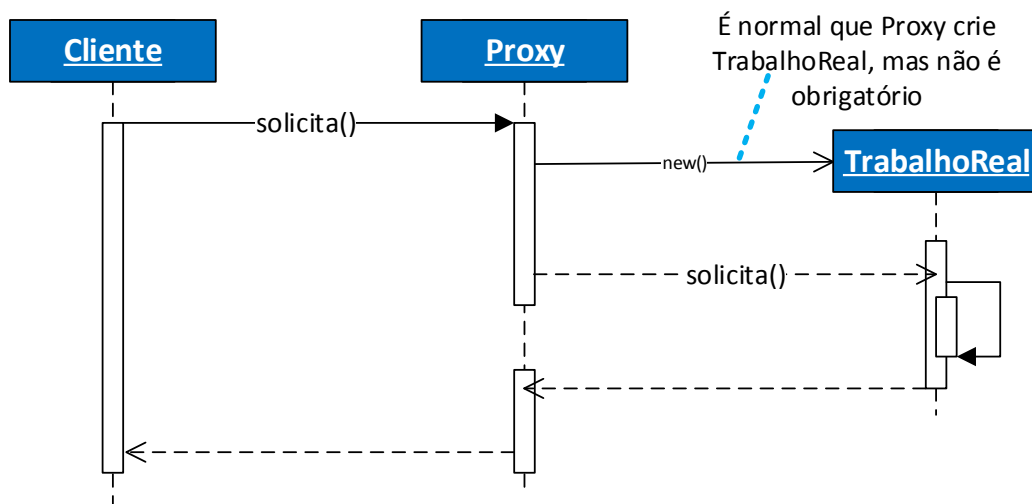


Figura 6.2: Sequencia na utilização de Proxy.

- adiciona um invólucro e delegação para proteger o componente real de complexidade indevida

O padrão **Proxy** pode ser normalmente utilizado quando você precisa estender um outro objeto de funcionalidades, especificamente as seguintes:

- Para controlar o acesso a um outro objeto, por exemplo, por razões de segurança.
- Para registrar todas as chamadas para o assunto com os seus parâmetros.
- Para se conectar ao Assunto, que está localizado no computador remoto ou outra espaço de endereço. Um **Proxy** tem uma interface de um objeto remoto, mas também lida com a rotina de ligação que é transparente para o chamador.
- Para instanciar um objeto pesado apenas quando é realmente necessário. Também pode cache um objeto pesado (ou parte dele).



- Para armazenar temporariamente alguns resultados de cálculo antes de voltar para múltiplos clientes que podem compartilhar estes resultados.
- Para contar as referências a um objeto.

### 6.3 Vantagens e desvantagens do Proxy

Os principais prós e contras de procuração são os seguintes:

- Um **Proxy** pode otimizar o desempenho de um aplicativo, usando o cache de objetos pesados ou utilizados com frequência.
- Um **Proxy** permite melhorar a segurança de um aplicativo, verificando acesso direitos em **Proxy** e delegando a `RealSubject` somente se os direitos são suficientes.
- Facilitar a interação entre sistemas remotos, um **Proxy** pode assumir a trabalho de conexões de rede e transmissão de rotina, delegação de chamadas para objetos remotos.

Às vezes o uso de padrão de **Proxy** pode aumentar o tempo de resposta para um objeto cliente. Por exemplo, se você usar o **Proxy** para a inicialização lenta e o recurso é solicitado pela primeira vez, o tempo de resposta será aumentado por tempo de inicialização. Da mesma forma o **Proxy** mascara se o recurso é local ou remoto. Neste segundo caso existe uma comunicação envolvida, que pode aumentar a ordem de grandeza do tempo de resposta. O **Proxy** mascara também o tempo de vida e o estado de um recurso que um cliente deseja. Um cliente ao chamar o **Proxy** não sabe que o recurso que deseja está disponível. Neste caso o **Proxy** tem que esperar até o recurso ficar disponível (bloqueando) ou enviar uma mensagem de recurso indisponível para o cliente. Este é um tipo de exceção não verificada em Java. O cliente também não tem como saber que o recurso que ficou disponível agora, é ou não o mesmo recurso que obteve na solicitação anterior. Desta forma, o recurso não pode manter um estado.

### 6.4 Uma implementação

Listagem 6.1: `Imagem.java`

```
1 import java.awt.image.BufferedImage;
2
3 public interface Imagem {
4
5     /** metodo a ser implementado pelas classes que usam esta interface */
6     public BufferedImage Imagem();
7
8 }
```

Listagem 6.2: `ImagemReal.java`

```
1 import java.net.URL;
2 import java.awt.image.BufferedImage;
3 import javax.imageio.ImageIO;
4 import java.io.IOException;
5
6 public class ImagemReal implements Imagem {
7
8     BufferedImage img = null;
9
10    public ImagemReal(URL url) {
11        //carrega a imagem a partir da url
12        carregaImagem(url);
13    }
```



```

13     }
14
15     public BufferedImage Imagem() {
16         return img;
17     }
18
19     /**
20      * le a imagem a partir da URL
21      * img == null se nao conseguiu carregar ou erro
22      */
23     private void carregaImagem(URL url) {
24         try {
25             img = ImageIO.read(url);
26             if (null != img) {
27                 System.out.println("Carregada a imagem de [" + url.toString() + ""]);
28             }
29         } catch (IOException e) {
30             img = null;
31         }
32     }
33 }

```

Listagem 6.3: ProxyImagem.java

```

1 import java.net.URL;
2 import java.awt.image.BufferedImage;
3
4 public class ProxyImagem implements Imagem {
5
6     private URL url;
7
8     public ProxyImagem(URL url) {
9         this.url = url;
10    }
11
12    /**
13     * este método delega para ImagemReal a apresentação real da imagem
14     */
15    public BufferedImage Imagem() {
16        ImagemReal real = new ImagemReal(url);
17        BufferedImage img = real.Imagem();
18        return img;
19    }
20
21 }

```

Listagem 6.4: Cliente.java

```

1 import java.net.URL;
2 import java.awt.image.BufferedImage;
3 import javax.swing.JFrame;
4 import java.awt.Graphics;
5 import java.awt.image.ColorConvertOp;
6 import java.awt.color.ColorSpace;
7
8 class Cliente extends JFrame {
9
10    BufferedImage img = null;
11
12    public Cliente(BufferedImage img) {
13        this.setTitle("Teste de Proxy");
14        /** dimensiona para caber a imagem */
15        int width = img.getWidth()+10;
16        int height = img.getHeight()+10;
17        this.setSize(width, height);

```





```

18     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19     this.img = img;
20 }
21
22 @Override
23 public void paint(Graphics g) {
24     if (null != img) {
25         // Coloca na tela a imagem"
26         g.drawImage(img, 5, 5, null);
27     }
28     else System.out.println("Sem imagem");
29 }
30
31 public static void main(String[] args) throws java.net.MalformedURLException
32 {
33     URL url = new URL("https://upload.wikimedia.org/wikipedia/commons/b/b4/
34         JPEG_example_JPG_RIP_100.jpg");
35     ProxyImagem img = new ProxyImagem(url);
36
37     Cliente c = new Cliente(img.Imagem());
38     c.setVisible(true);
39     c.repaint();
40 }

```

Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresenta na figura 6.3.

```

henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/proxy
h3dema@casa-desktop:proxy$ javac Imagem.java ProxyImagem.java Cliente.java
h3dema@casa-desktop:proxy$ java Cliente
Carregada a imagem de [https://upload.wikimedia.org/wikipedia/commons/b/b4/JPEG_
example_JPG_RIP_100.jpg]
h3dema@casa-desktop:proxy$

```

Figura 6.3: Execução de ClienteProxy.



Figura 6.4: JFrame criado pela classe Cliente.



## 6.5 Conclusões

Um proxy é uma classe, funcionando como uma interface para outra classe, que tem a mesma interface como o proxy. O código do cliente instancia e trabalha diretamente com o proxy, considerando que, os delegados de proxy trabalham para uma classe de cliente. Proxies têm muitos usos, particularmente para armazenamento em cache, contagem de referência e controle de acesso com o botão direito. Usuários de procuração devem ter cuidado para evitar o aumento no tempo de resposta.

## Capítulo 7

# O padrão Mediator

*Padrão de projeto* **Mediator** ou mediador é utilizado para fornecer de forma centralizada a comunicação entre diferentes objetos em um sistema. Este padrão é muito útil em um aplicativo corporativo, onde vários objetos estão interagindo uns com os outros. Se os objetos interagirem uns com os outros diretamente, os componentes do sistema estarão firmemente acoplados uns com os outros. Isto faz com que o custo de manutenção seja mais elevada. O aplicativo também fica menos flexível.

### 7.1 O padrão

O padrão de mediador se concentra em fornecer um mediador entre os objetos de comunicação. Ele ajuda na redução do acoplamento entre os objetos na implementação. As classes que se comunicam com o mediador são conhecidos como *Colegas* (do inglês *Colleagues*). A figura 7.1 mostra um exemplo de um diagrama de classes. O mediador pode ter uma interface que explicita a comunicação com *Colegas*. Na figura denominamos *MediatorInterface*. A implementação mediador é conhecido como o *MediatorImpl*. As classes que implementam a interface *ColegaInterface* conhecem seu mediador, e o mediador conhece seus colegas.

### 7.2 Considerações sobre o padrão

O padrão **Mediator** é útil quando a lógica de comunicação entre objetos é complexa. O padrão permite que tenhamos um ponto central de comunicação que cuida da lógica de comunicação. Por exemplo, o *Java Message Service* (JMS) utiliza o padrão **Mediator** junto com padrão **Observer** para permitir que aplicativos possam se inscrever e publicar dados para outros aplicativos. Outro exemplo na API Java que utiliza este padrão é a classe *java.util.Timer*. Esta classe chama uma classe que estende *java.util.TimerTask*. *TimerTask* é uma manipulação de thread que permite que registremos tarefas para serem executadas pelo temporizador.

Não devemos usar o padrão de mediador apenas para conseguir reduzir o acoplamento porque se o número de mediadores crescer, então a aplicação também se tornará difícil de manter.

### 7.3 Uma implementação

Vamos ver um exemplo de um chat via socket em Java.

Listagem 7.1: Definindo a classe que define o tipo de mensagem que pode ser enviada

```
1 import java.io.*;
2 /*
```

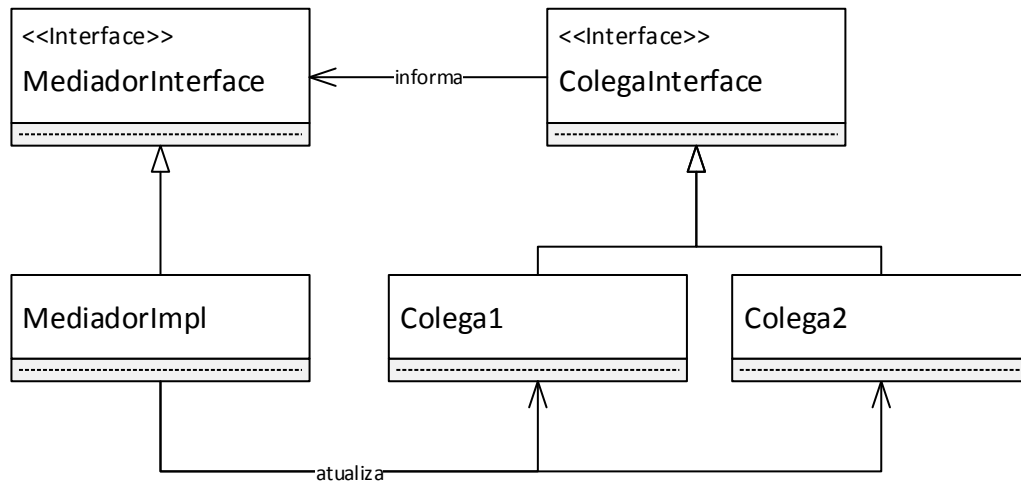


Figura 7.1: Diagrama de classes de um mediador.

```

3  * Define a estrutura das mensagens que serão transmitidas
4  */
5  public class MensagemChat implements Serializable {
6
7      protected static final long serialVersionUID = 1112122200L;
8
9      // Tipos de mensagem que podem ser usadas:
10     static final int USUARIOS = 0; // solicita lista de usuarios logado
11     static final int MSG      = 1; // envia texto
12     static final int LOGOUT   = 2; // informa desconexão
13
14     private int tipo;
15     private String msg;
16
17     /**
18      * construtor
19      */
20     MensagemChat(int tipo, String msg) {
21         this.tipo = tipo;
22         this.msg = msg;
23     }
24
25     int tipo() {
26         return tipo;
27     }
28
29     String mensagem() {
30         return msg;
31     }
32 }

```

Listagem 7.2: Definindo a classe do servidor

```

1  import java.io.*;
2  import java.net.*;
3  import java.text.SimpleDateFormat;
4  import java.util.*;
5

```



```
6  /*
7  * The server that can be run both as a console application or a GUI
8  */
9  public class ChatServer implements ServerInterface {
10
11     public static final int PORTA_PADRAO = 10000;
12
13     // a unique ID
14     private static int uniqueId;
15     // lista dos clientes
16     private ArrayList<ThreadCliente> clientes = new ArrayList<>();
17     // formatação para apresentação de hora
18     private SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");
19     // porta de conexão do socket
20     protected int port;
21     // variavel que indica se o servidor deve rodar
22     protected boolean executando;
23
24     /**
25      * construtores
26      */
27     public ChatServer() {
28         this(PORTA_PADRAO);
29     }
30
31     public ChatServer(int port) {
32         this.port = port;
33     }
34
35     public void escreveEvento(String txt) {
36         System.out.println(txt); // manda mensagem para console
37     }
38
39     public void escreveRoom(String txt) {
40         System.out.print(txt); // console
41     }
42
43     /*
44      * mostra o evento
45      */
46     private void display(String msg) {
47         String evento = sdf.format(new Date()) + " " + msg;
48         escreveEvento(evento);
49     }
50
51     // inicia o servidor
52     public void start() {
53         executando = true;
54         /* cria socket para servidor escutar clientes */
55         try {
56             ServerSocket serverSocket = new ServerSocket(port);
57             // loop para as conexoes
58             while(executando) {
59                 display("Servidor_␣aguardando_␣mensagens_␣na_␣porta_␣" + port + ".");
60
61                 Socket socket = serverSocket.accept(); // accept connection
62
63                 if(!executando) break; // força a saída
64
65                 ThreadCliente t = new ThreadCliente(socket); // conexao eh nova thread
66                 clientes.add(t);
67                 t.start();
68             }
69             // finalizando....
70             try {
71                 serverSocket.close();
```



```

72         for(int i = 0; i < clientes.size(); ++i) {
73             ThreadCliente tc = clientes.get(i);
74             tc.close();
75         }
76     } catch(Exception e) {
77         display("Erro fechando: " + e);
78     }
79 } catch (IOException e) {
80     String msg = sdf.format(new Date()) + " Erro de Socket em: " + e + "\n";
81     display(msg);
82 }
83 }
84
85 /*
86  * forma de fazer a GUI do servidor, fechar as conexões
87  */
88 protected void stop() {
89     executando = false;
90     // connect to myself as Client to exit statement
91     // Socket socket = serverSocket.accept();
92     try {
93         new Socket("localhost", port);
94     }
95     catch(Exception e) {
96         // nothing I can really do
97     }
98 }
99
100 /*
101  * enviar broadcast para todos os usuarios conectados
102  */
103 private synchronized void broadcast(String message) {
104     String time = sdf.format(new Date());
105     String messageLf = time + "\n" + message + "\n";
106     // display message on console or GUI
107     escreveRoom(messageLf);
108
109     ArrayList<ThreadCliente> remover = new ArrayList<>();
110     for(ThreadCliente ct : clientes) {
111         if(!ct.escreveMsg(messageLf)) {
112             remover.add(ct); // remover estes clientes da lista
113             display("Cliente " + ct.nome + " desconectou.");
114         }
115     }
116     // remove desconectados
117     for(ThreadCliente ct : remover) {
118         clientes.remove(ct);
119     }
120 }
121
122 // remove usuario que enviou mensagem LOGOUT
123 synchronized void usuarioDesconectou(int id) {
124     // scan the array list until we found the Id
125     for(int i = 0; i < clientes.size(); ++i) {
126         ThreadCliente ct = clientes.get(i);
127         if(ct.id == id) {
128             clientes.remove(i);
129             return;
130         }
131     }
132 }
133
134 public static void uso() {
135     System.out.println("Uso:");
136     System.out.println("\tjava ChatServer");
137     System.out.println("\tjava ChatServer <num_porta>");

```



```
138     System.out.println("Porta default=" + PORTA_PADRAO);
139     System.exit(0);
140 }
141
142 /*
143  * To run as a console application just open a console window and:
144  */
145 public static void main(String[] args) {
146     int porta = PORTA_PADRAO;
147     if (args.length > 1) uso();
148     if (args.length == 1) {
149         try {
150             porta = Integer.valueOf(args[0]);
151         } catch (NumberFormatException e) {
152             porta = PORTA_PADRAO;
153         }
154     }
155     System.out.println("Iniciando servidor na porta " + porta);
156     ChatServer server = new ChatServer(porta);
157     server.start();
158 }
159
160 /** *****
161  * classe interna
162  *
163  * Será criada uma instância desta classe para cada cliente
164  * ***** */
165 class ThreadCliente extends Thread {
166     Socket socket;
167     ObjectInputStream sInput; // entrada de dados
168     ObjectOutputStream sOutput; // saída de dados
169
170     int id;
171     String nome;
172     MensagemChat cm; // padrão de mensagem que receberá ou enviará
173     String ts; // timestamp do início da conexão
174
175     // Construtor
176     ThreadCliente(Socket socket) {
177         // a unique id
178         id = ++uniqueId;
179         this.socket = socket;
180         try {
181             // cria stream, primeiro de saída e depois de entrada a partir do
182             // socket
183             sOutput = new ObjectOutputStream(socket.getOutputStream());
184             sInput = new ObjectInputStream(socket.getInputStream());
185             // read the nome
186             nome = (String) sInput.readObject();
187             display(nome + " conectado.");
188         } catch (IOException e) {
189             display("Erro ao criar streams: " + e);
190             return;
191         } catch (ClassNotFoundException e) {}
192         ts = new Date().toString();
193     }
194
195     /*
196     * envia uma string para o cliente
197     */
198     private boolean escreveMsg(String msg) {
199         // só envia mensagem se o socket estiver conectado
200         if (!socket.isConnected()) {
201             close();
202             return false;
203         }
204     }
```



```

203         try { // envia a mensagem
204             sOutput.writeObject(msg);
205         } catch(IOException e) {
206             // informa erro ao enviar
207             display("Erro ao enviar mensagem para usr: " + nome);
208             display(e.toString());
209         }
210         return true;
211     }
212
213     /*
214     * roda até mensagem de parar
215     */
216     public void run() {
217         boolean executando = true;
218         while(executando) {
219             try {
220                 cm = (MensagemChat) sInput.readObject(); // Lê msg
221             } catch (IOException e) {
222                 display(nome + " Erro ao ler entrada: " + e);
223                 break;
224             }
225             catch(ClassNotFoundException e2) {
226                 break;
227             }
228             String message = cm.mensagem(); // mensagem
229             switch(cm.tipo()) { // tipo da mensagem
230
231                 case MensagemChat.MSG:
232                     broadcast(nome + ": " + message);
233                     break;
234                 case MensagemChat.LOGOUT:
235                     display(nome + " desconectou.");
236                     executando = false;
237                     break;
238                 case MensagemChat.USUARIOS:
239                     escreveMsg("Lista de usuários conectados" + sdf.format(new Date())
240                             + "\n");
241                     // scan clientes the users connected
242                     for(int i = 0; i < clientes.size(); ++i) {
243                         ThreadCliente ct = clientes.get(i);
244                         escreveMsg((i+1) + ") " + ct.nome + " conectado em " + ct.ts + "\n");
245                     }
246                     break;
247             }
248             // remove usuario "id" da lista de usuários conectados
249             usuarioDesconectou(id);
250             close();
251         }
252
253         // fecha streams e socket
254         private void close() {
255             // try to close the connection
256             try {
257                 if(sOutput != null) sOutput.close();
258             } catch(Exception e) {}
259             try {
260                 if(sInput != null) sInput.close();
261             } catch(Exception e) {};
262             try {
263                 if(socket != null) socket.close();
264             } catch (Exception e) {}
265         }
266     }

```





267 | }

Listagem 7.3: Definindo a classe do cliente

```
1 import java.net.*;
2 import java.io.*;
3 import java.util.*;
4
5 /*
6  * classe cliente
7  */
8 public class ChatClient {
9
10     public static final String USUARIO_PADRAO = "anonimo";
11
12     // para realizar I/O
13     private Socket socket;
14     private ObjectInputStream sInput;    // ler do socket
15     private ObjectOutputStream sOutput; // escrever no socket
16
17     // infos do servidor de chat
18     private String servidor;
19     private int porta;
20     private String usuario;
21
22     protected boolean conectado;
23
24     /*
25      * Construtor com usuário padrão
26      */
27     ChatClient(String servidor, int porta) {
28         this(servidor, porta, USUARIO_PADRAO);
29     }
30
31     ChatClient(String servidor, int porta, String usuario) {
32         this.servidor = servidor;
33         this.porta = porta;
34         this.usuario = usuario;
35     }
36
37     public void setServidor(String servidor) { this.servidor = servidor; }
38     public String getServidor() { return servidor; }
39     public void setPorta(int porta) { this.porta = porta; }
40     public int getPorta() { return porta; }
41     public void setUsuario(String usuario) { this.usuario = usuario; }
42
43     /*
44      * iniciar a comunicação com servidor
45      */
46     public boolean start() {
47         try {
48             socket = new Socket(servidor, porta);
49         } catch (Exception e) {
50             // informa se falhou e sai...
51             display("Erro ao conectar ao servidor:" + e + "\n");
52             return false;
53         }
54         // ok, conexão aceita
55         display("Conexão aceita " + socket.getInetAddress() + ":" + socket.getPort() + "\n");
56
57         // cria forma para ler e escrever mensagens
58         try {
59             sInput = new ObjectInputStream(socket.getInputStream());
60             sOutput = new ObjectOutputStream(socket.getOutputStream());
61         } catch (IOException e) {
```



```

62         display("Exceção na criação de streams: " + e + "\n");
63         return false;
64     }
65     // cria thread para escutar mensagens do servidor
66     new RecebeMsgServidor().start();
67     // envia usuario do usuário para servidor
68     try {
69         sOutput.writeObject(usuario);
70     } catch (IOException e) {
71         display("Erro de login: " + e + "\n");
72         desconectar();
73         return false;
74     }
75     conectado = true;
76     return conectado; // tudo certo !!
77 }
78
79 /*
80  * enviar mensagem
81  */
82 protected void display(String msg) {
83     System.out.println(msg);
84 }
85
86 /*
87  * enviando a mensagem ao servidor
88  */
89 protected void enviarMsg(MensagemChat msg) {
90     try {
91         sOutput.writeObject(msg);
92     } catch (IOException e) {
93         display("Erro de envio de msg: " + e + "\n");
94     }
95 }
96
97 // nesta classe não faz nada
98 // será sobreposto no GUI
99 protected void trataConexaoFalhou() { }
100
101 /*
102  * desconecta streams e fecha sockets
103  */
104 private void desconectar() {
105     try {
106         if(sInput != null) sInput.close();
107     } catch (Exception e) {}
108     try {
109         if(sOutput != null) sOutput.close();
110     } catch (Exception e) {}
111     try {
112         if(socket != null) socket.close();
113     } catch (Exception e) {}
114
115     conectado = false;
116     // coloca info na tela
117     trataConexaoFalhou();
118 }
119
120 private static void uso() {
121     System.out.println("Uso: > java ChatClient [usuario] [portaServidor] [
        enderecoServidor]");
122     System.out.println("\t* 0 valor padrão de portaServidor é " + ChatServer.
        PORTA_PADRAO + ".");
123     System.out.println("\t* Se enderecoServidor não for fornecido, \"localhost
        \" é utilizado.");

```



```
124     System.out.println("\t*Se_usuario_não_for_indicado,será_utilizado\""+
125         USUARIO_PADRAO+".");
126     System.exit(0);
127 }
128 /**
129  * main()
130  */
131 public static void main(String[] args) {
132     int portNumber = ChatServer.PORTA_PADRAO;
133     String serverAddress = "localhost";
134     String usuario = USUARIO_PADRAO;
135     switch(args.length) {
136         case 3: serverAddress = args[2];
137         case 2: {
138             try {
139                 portNumber = Integer.parseInt(args[1]);
140             } catch(Exception e) {
141                 portNumber = ChatServer.PORTA_PADRAO;
142             }
143         }
144         case 1: usuario = args[0];
145         case 0: break;
146         default: uso();
147     }
148     // cria o cliente para enviar mensagens
149     ChatClient client = new ChatClient(serverAddress, portNumber, usuario);
150     // testa se conseguiu conectar, senão sai
151     if(!client.start())
152         return;
153
154     Scanner scan = new Scanner(System.in);
155     // aguarda mensagens
156     while(true) {
157         System.out.print(">");
158         // read message from user
159         String msg = scan.nextLine();
160         // logout if message is LOGOUT
161         if(msg.equalsIgnoreCase("LOGOUT")) {
162             client.enviarMsg(new MensagemChat(MensagemChat.LOGOUT, ""));
163             break;
164         }
165         // message WhoIsIn
166         else if(msg.equalsIgnoreCase("USUARIOS")) {
167             client.enviarMsg(new MensagemChat(MensagemChat.USUARIOS, ""));
168         }
169         else {
170             // mensagem comum
171             client.enviarMsg(new MensagemChat(MensagemChat.MSG, msg));
172         }
173     }
174     // fim: desconectar
175     client.desconectar();
176 }
177
178 /** *****
179  * classe interna
180  *
181  * recebe mensagem do servidor e escreve na tela
182  * *****/
183 class RecebeMsgServidor extends Thread {
184
185     public void run() {
186         while(true) { // roda para sempre
187             try {
```



```

189         String msg = (String) sInput.readObject();
190         display(msg);
191     } catch(IOException e) {
192         display("Servidor_␣fechou_␣conexão:␣" + e + "\n");
193         trataConexaoFalhou();
194         break;
195     } catch(ClassNotFoundException e) {
196         // exceção
197     }
198 }
199 }
200 }
201 }

```

Podemos melhorar a aparência do nosso aplicativo criando interfaces gráficas para o cliente e o servidor.

Listagem 7.4: Definindo a classe com a interface GUI do servidor

```

1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  /*
6   * GUI do servidor de chat
7   */
8  public class ChatServerGUI extends ChatServer implements ActionListener,
9         WindowListener {
10
11     private static final long serialVersionUID = 1L;
12
13     private JFrame fr; // janela do GUI
14     private JButton btLigaDesligaServer; // liga/desliga servidor
15     private JTextArea chat;
16     private JTextArea evento;
17     private JTextField portaServidor;
18
19     private ChatServer server; // usando dentro de ServerRunning
20
21     // construtor
22     ChatServerGUI(int porta) {
23         super(porta);
24         server = this;
25         init(porta);
26     }
27
28     // inicia GUI
29     private void init(int port) {
30         fr = new JFrame("Servidor_␣de_␣Chat");
31         JFrame.setDefaultLookAndFeelDecorated(true);
32         fr.setAlwaysOnTop(true);
33         fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
34
35         // área superior da tela
36         JPanel pnSuperior = new JPanel(new BorderLayout());
37         pnSuperior.add(new JLabel("Número_␣da_␣porta:␣"), BorderLayout.LINE_START);
38         portaServidor = new JTextField("␣" + port);
39         pnSuperior.add(portaServidor, BorderLayout.CENTER);
40         btLigaDesligaServer = new JButton("Iniciar_␣Servidor");
41         btLigaDesligaServer.addActionListener(this);
42         pnSuperior.add(btLigaDesligaServer, BorderLayout.LINE_END);
43         fr.add(pnSuperior, BorderLayout.NORTH);
44
45         // área de notificação de eventos e chat
46         JPanel pnCentral = new JPanel(new GridLayout(2,1));
47         chat = new JTextArea(80,80);

```



```
47     chat.setEditable(false);
48     escreveRoom("Sala de chat.\n");
49     pnCentral.add(new JScrollPane(chat));
50     evento = new JTextArea(80,80);
51     evento.setEditable(false);
52     pnCentral.add(new JScrollPane(evento));
53     fr.add(pnCentral);
54
55     fr.addWindowListener(this); // a propria classe GUI tratará os eventos da
56                                 janela
57     fr.setSize(600, 400);
58     fr.setVisible(true);
59 }
60
61 // sobrepõe o método de escrita de eventos na tela
62 @Override
63 public void escreveEvento(String txt) {
64     evento.append(txt);
65     evento.setCaretPosition(chat.getText().length() - 1);
66 }
67
68 // sobrepõe o método de escrita de textos do chat na tela
69 @Override
70 public void escreveRoom(String txt) {
71     chat.append(txt);
72     chat.setCaretPosition(chat.getText().length() - 1);
73 }
74
75 // botao de LigaDesligado
76 public void actionPerformed(ActionEvent e) {
77     // para se estiver rodando
78     if(executando) {
79         stop();
80         portaServidor.setEditable(true);
81         btLigaDesligaServer.setText("Iniciar Servidor");
82         return;
83     }
84     // se não está rodando, deve iniciar
85     int port = 0;
86     try {
87         port = Integer.parseInt(portaServidor.getText().trim());
88     }
89     catch(Exception er) {
90         escreveEvento("Porta "+port+" inválida");
91         return;
92     }
93     this.port = port; // define a porta para execução
94     new ServerRunning().start(); // roda a thread do servidor de chat
95     btLigaDesligaServer.setText("Parar Servidor");
96     portaServidor.setEditable(false);
97 }
98
99 /*
100  * terminado operações no fechamento da janela principal
101  */
102 public void windowClosing(WindowEvent e) {
103     System.out.println("Saindo do programa....");
104     if(executando) {
105         try {
106             stop(); // fecha as conexões existentes
107         }
108         catch(Exception eClose) {
109         }
110     }
111     System.exit(0);
112 }
```



```

112
113 // ignorando demais WindowListener
114 public void windowClosed(WindowEvent e) {}
115 public void windowOpened(WindowEvent e) {}
116 public void windowIconified(WindowEvent e) {}
117 public void windowDeiconified(WindowEvent e) {}
118 public void windowActivated(WindowEvent e) {}
119 public void windowDeactivated(WindowEvent e) {}
120
121 public static void main(String[] arg) {
122     // inicia o servidor na porta padrao
123     new ChatServerGUI(ChatServer.PORTA_PADRAO);
124 }
125
126 /*
127  * A thread to run the Server
128  */
129 class ServerRunning extends Thread {
130     public void run() {
131         server.start(); // executa esta linha ate ser parado
132         //
133         escreveEvento("Servidor parado.\n");
134         // garante que botao e texto da porta estao liberados
135         btLigaDesligaServer.setText("Iniciar Servidor");
136         portaServidor.setEditable(true);
137     }
138 }
139
140 }

```

Listagem 7.5: Definindo a classe com a interface GUI do cliente

```

1
2 import javax.swing.*;
3 import java.awt.*;
4 import java.awt.event.*;
5
6
7 /*
8  * The Client with its GUI
9  */
10 public class ChatClientGUI extends ChatClient implements ActionListener {
11
12     private static final long serialVersionUID = 1L;
13     // entrada de dados : usuário e depois mensagens
14     private JFrame fr; // janela do GUI
15     private JLabel label;
16     private JTextField tf;
17
18     // to Logout and get the list of the users
19     private JButton login;
20     private JButton logout;
21     private JButton listaUsr;
22
23     // campo de texto com as mensagens
24     private JTextArea ta;
25
26     // campos de entrada de dados do servidor
27     private JTextField tfServidor, tfPorta;
28
29     // Construtor
30     ChatClientGUI(String servidor, int porta) {
31         super(servidor, porta);
32         init(servidor, porta);
33     }
34

```



```
35 private void init(String servidor, int porta) {
36     fr = new JFrame("Cliente de Chat");
37     JFrame.setDefaultLookAndFeelDecorated(true);
38     fr.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39
40     JPanel np = new JPanel(new GridLayout(3,1));
41
42     // linha 1 - parte superior da tela
43     JPanel serverAndPort = new JPanel(new GridLayout(1, 5, 1, 3));
44     serverAndPort.add(new JLabel("Servidor: "));
45     tfServidor = new JTextField(servidor);
46     serverAndPort.add(tfServidor);
47     serverAndPort.add(new JLabel("Porta: "));
48     tfPorta = new JTextField("" + porta);
49     tfPorta.setHorizontalAlignment(SwingConstants.RIGHT);
50     serverAndPort.add(tfPorta);
51     serverAndPort.add(new JLabel(""));
52     np.add(serverAndPort);
53
54     // linha 2
55     label = new JLabel("Entre seu nome de usuário:", SwingConstants.LEFT);
56     // linha 3
57     np.add(label);
58     tf = new JTextField(ChatClient.USUARIO_PADRAO);
59     tf.setBackground(Color.WHITE);
60     np.add(tf);
61     fr.add(np, BorderLayout.NORTH);
62
63     // parte central da tela
64     ta = new JTextArea("Bem vindo!\n", 80, 80);
65     JPanel centerPanel = new JPanel(new GridLayout(1,1));
66     centerPanel.add(new JScrollPane(ta));
67     ta.setEditable(false);
68     fr.add(centerPanel, BorderLayout.CENTER);
69
70     // parte inferior com botões
71     login = new JButton("Conectar");
72     login.addActionListener(this);
73     logout = new JButton("Desconectar");
74     logout.addActionListener(this);
75     listaUsr = new JButton("Usr. Conectados");
76     listaUsr.addActionListener(this);
77
78     logout.setEnabled(false); // só fica ativo depois do login
79     listaUsr.setEnabled(false); // só fica ativo depois do login
80
81     JPanel southPanel = new JPanel();
82     southPanel.add(login);
83     southPanel.add(logout);
84     southPanel.add(listaUsr);
85     fr.add(southPanel, BorderLayout.SOUTH);
86
87     fr.setSize(600, 600);
88     fr.setVisible(true);
89     tf.requestFocus();
90
91 }
92
93 /*
94  * enviar mensagem
95  * sobrepõe o método em ChatClient para poder escrever na tela
96  */
97 @Override
98 protected void display(String msg) {
99     ta.append(msg);
100     ta.setCaretPosition(ta.getText().length() - 1);
```



```

101     }
102
103     // called by the GUI is the connection failed
104     // we reset our buttons, label, textfield
105     @Override
106     protected void trataConexaoFalhou() {
107         super.trataConexaoFalhou(); // chama o procedimento da superclasse
108         login.setEnabled(true);
109         logout.setEnabled(false);
110         listaUsr.setEnabled(false);
111         label.setText("Entre seu nome de usuário");
112         tf.setText(ChatClient.USUARIO_PADRAO);
113         // reset port number and host name as a construction time
114         tfPorta.setText("" + getPorta());
115         tfServidor.setText(getServidor());
116         // let the user change them
117         tfServidor.setEditable(false);
118         tfPorta.setEditable(false);
119         // don't react to a <CR> after the nome
120         tf.removeActionListener(this);
121     }
122
123     /*
124      * trata eventos relacionados aos botões
125      */
126     public void actionPerformed(ActionEvent e) {
127         Object o = e.getSource(); // verifica qual botao foi acionado
128         if(o == logout) {
129             enviarMsg(new MensagemChat(MensagemChat.LOGOUT, ""));
130             return;
131         }
132         if(o == listaUsr) {
133             enviarMsg(new MensagemChat(MensagemChat.USUARIOS, ""));
134             return;
135         }
136         // envia mensagem, note que botao login esta desabilitado se conectado ==
137          true
138         if(conectado) {
139             enviarMsg(new MensagemChat(MensagemChat.MSG, tf.getText()));
140             tf.setText("");
141             return;
142         }
143         // tenta executar login
144         if(o == login) {
145             // decodifica os campos
146             String usuario = tf.getText().trim();
147             if(usuario.length() == 0) {
148                 JOptionPane.showMessageDialog(fr,
149                     "Nome de usuário não pode estar vazio.",
150                     "Erro", JOptionPane.ERROR_MESSAGE);
151                 return;
152             }
153             // empty serverAddress ignore it
154             String servidor = tfServidor.getText().trim();
155             if(servidor.length() == 0) {
156                 JOptionPane.showMessageDialog(fr,
157                     "Campo com endereço do servidor está vazio.", "Erro", JOptionPane.
158                     ERROR_MESSAGE);
159                 return;
160             }
161             String porta_str = tfPorta.getText().trim();
162             if(porta_str.length() == 0) {
163                 JOptionPane.showMessageDialog(fr,
164                     "Campo com valor da porta está vazio.", "
165                     Erro", JOptionPane.ERROR_MESSAGE);

```



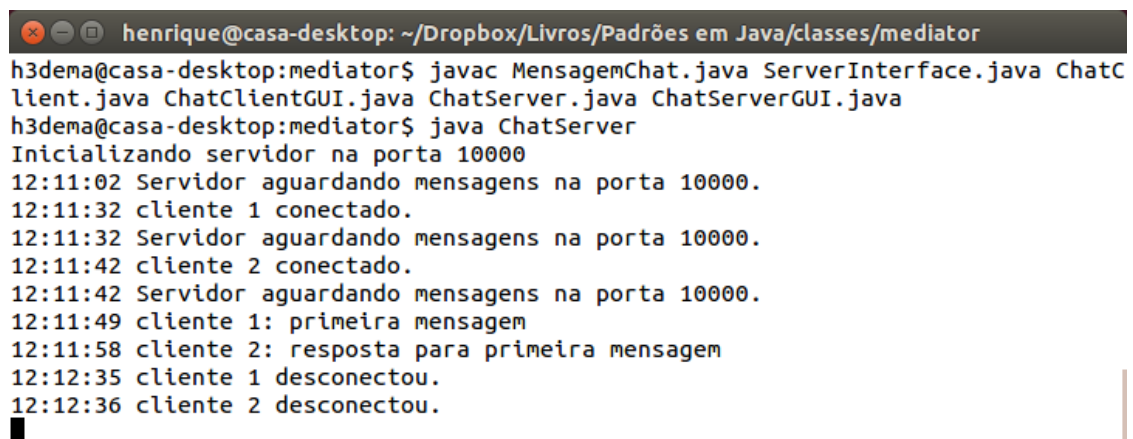


```

162         return;
163     }
164     int porta = 0;
165     try { // tenta converter porta
166         porta = Integer.parseInt(porta_str);
167     }
168     catch(Exception en) {
169         JOptionPane.showMessageDialog(fr,
170                                     "Erro de conversão do valor da porta.", "
171                                     Erro", JOptionPane.ERROR_MESSAGE);
172
173         return; // se deu erro, sai
174     }
175
176     // verifica se está rodando
177     setServidor(servidor);
178     setPorta(porta);
179     setUsuario(usuario);
180     if(!start()) return; // se deu erro, sai
181     tf.setText("");
182     label.setText("Entre com a mensagem");
183     conectado = true;
184     // ** se fez login, acerta os botões apropriadamente
185     login.setEnabled(false);
186     logout.setEnabled(true);
187     listaUsr.setEnabled(true);
188     // desabilita campos relacionados ao servidor
189     tfServidor.setEditable(false);
190     tfPorta.setEditable(false);
191     // configura actionlistener
192     tf.addActionListener(this);
193 }
194
195 // cria um cliente GUI
196 public static void main(String[] args) {
197     new ChatClientGUI("localhost", ChatServer.PORTA_PADRAO);
198 }
199 }

```

Vamos compilar as classes indicadas nas listagens. Iremos executar a versão do servidor em linha de comando e do cliente em tela gráfica. As telas mostrando a interação entre dois clientes GUI são mostradas nas figuras 7.2 e 7.3.



```

henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/mediator
h3dema@casa-desktop:mediator$ javac MensagemChat.java ServerInterface.java ChatC
lient.java ChatClientGUI.java ChatServer.java ChatServerGUI.java
h3dema@casa-desktop:mediator$ java ChatServer
Inicializando servidor na porta 10000
12:11:02 Servidor aguardando mensagens na porta 10000.
12:11:32 cliente 1 conectado.
12:11:32 Servidor aguardando mensagens na porta 10000.
12:11:42 cliente 2 conectado.
12:11:42 Servidor aguardando mensagens na porta 10000.
12:11:49 cliente 1: primeira mensagem
12:11:58 cliente 2: resposta para primeira mensagem
12:12:35 cliente 1 desconectou.
12:12:36 cliente 2 desconectou.

```

Figura 7.2: Execução do servidor.



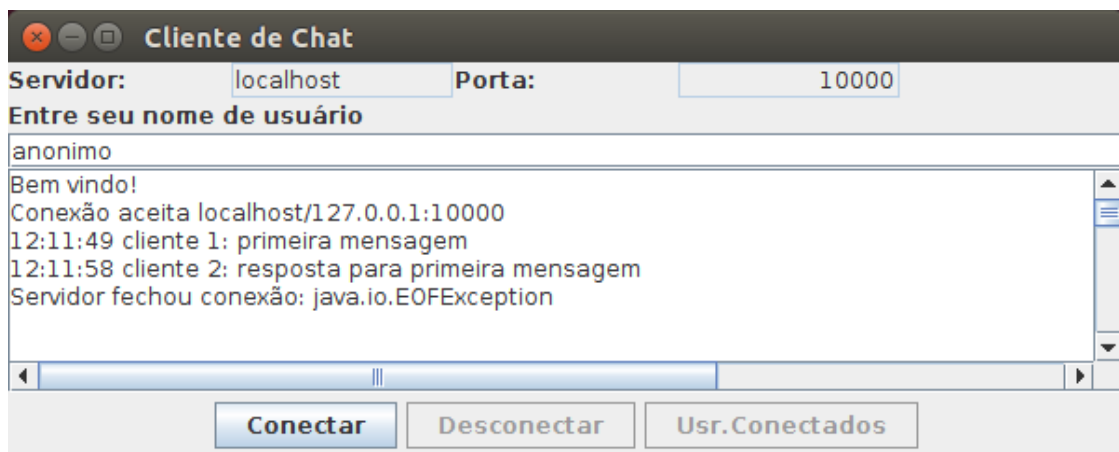


Figura 7.3: Execução vista a partir de um dos clientes GUI.

## Capítulo 8

# Padrão: Observer

O padrão de projeto **Observer** tenta facilitar a utilização de relacionamentos do tipo um-para-muitos em problemas de engenharia de software. Há muitas situações que lidam com um-para-muitos relacionamentos: vários leitores se inscrever em um blog, vários ouvintes de eventos inscrever-se a lidar com rato clica em um item de interface do usuário, ou vários aplicativos de telefone se inscrever para receber um notificação quando eles obter dados a partir da Internet.

O padrão de projeto **Observer** é muito semelhante à subscrição de um jornal no seguinte aspectos:

- O assinante abre inscrição para o jornal
- Você se inscrever para o jornal
- Alguém assina o jornal
- Quando há um novo jornal, você e que alguém começa um novo jornal
- Se você não quiser receber o jornal mais, você cancelar a sua assinatura e você não receberá próxima jornal (mas outros o farão)

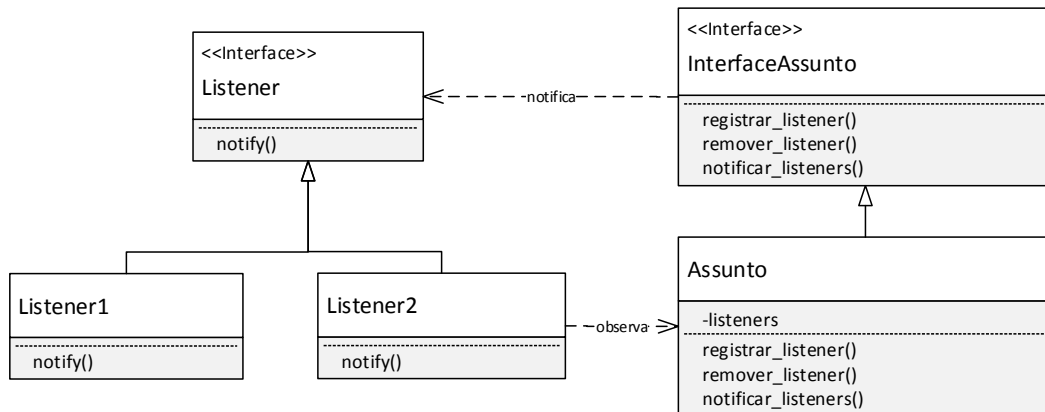
### 8.1 O padrão Observer

Este é um tipo de padrão de publicação de assinante que usa qualquer tipo de informação, tais como novos dados recebidos de alguns outros recursos, atualização de um dados já registrado, um sinal de linha ou um sinal do sistema operacional. Baseia em uma relação de editor/assinante (em inglês publisher/subscriber). Esta informação deverá ser transmitida aos assinantes e o padrão de projeto **Observer** serve para o gerenciamento de subscrição e da entrega.

No padrão **Observer**, um objeto chamado o Assunto mantém uma lista com um conjunto de outros objetos chamados observadores, que são os assinantes deste Assunto. No caso de existir quaisquer mudanças de estado, ele notifica-los, chamando um dos seus métodos (no nosso exemplo, denominamos notify).

Mostramos no diagrama da figura 8.1 um exemplo deste padrão. O observador é uma interface que tem o resumo método de notificação. No exemplo, esta interface é denominada Listener e possui um método notify que deve ser chamado para notificá-la de uma alteração. Temos duas classes concretas, denominadas Listener1 e Listener2, que são derivados a partir da interface e Listener. Estas classes precisam implementar o método abstrato notify.

A classe que faz a notificação é a classe Assunto que mantém um conjunto de instâncias de observadores concretas, na lista denominada listeners. Esta classe pode acrescentar novas instâncias Listener chamando registrar\_listener. Uma instância pode ser removida chamando o remover\_listener, quando o objeto não quiser mais receber notificações. Quando algum evento

Figura 8.1: Uso do padrão **Observer**.

acontece, a classe Assunto chama o método `notificar_listeners`. Este método varre o conjunto de observador, chamando o seu método de notificação.

A classe Assunto é a parte estática do sistema. Durante todo o tempo de vida da aplicação, existe somente um apenas um Assunto. Já os observadores são a parte dinâmica, podem haver muitos ou mesmo zero observadores. A quantidade destes objetos registrados junto a Assunto pode sofrer mudanças durante o tempo de vida da aplicação.

O padrão Observer é usado quando uma mudança em um objeto conduz a uma alteração nos outros objetos, e você não sabe quantos objetos devem ser alterados.

## 8.2 Problemas resolvidos pelo padrão Observer

Se há uma exigência de que um objeto particular mudar o seu estado, e dependendo essas mudanças alguns ou um grupo de objetos alterar automaticamente seu estado, precisamos para implementar o padrão **Observer** para reduzir o acoplamento entre objetos.

Um exemplo do mundo real pode ser encontrado em serviços de *microblogging*, como o Twitter. Quando você posta uma nova mensagem, todos os seus seguidores (observadores) serão notificados.

Em geral, utilizamos este padrão para reduzir o acoplamento. Se temos um objeto que precisa compartilhar seu estado com os outros, sem saber quem são esses objetos, o padrão **Observer** é a forma correta de implementar esta relação.

## 8.3 Vantagens do padrão Observer

O padrão de projeto **Observer** tem as seguintes vantagens:

- A manutenção de um baixo acoplamento entre sujeito e observadores. o Assunto só conhece a lista de observadores e suas interfaces; ele não se preocupa com a sua classe concreta, detalhes de implementação, e assim por diante.
- Capacidade de transmissão de mensagens entre sujeito e observadores.
- O número de observadores pode ser alterado em tempo de execução.
- O Sujeito pode manter qualquer número de observadores.



## 8.4 Uma implementação

Listagem 8.1: Definindo a interface InterfaceAssunto.java

```
1  /** interface que define os métodos do Assunto */
2  public interface InterfaceAssunto {
3
4      public void registrar_listener(Listener l);
5      public void remover_listener(Listener l);
6
7      public void notificarListeners();
8
9  }
```

Listagem 8.2: Definindo a classe concreta Assunto.java

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  /** classe concreta que trabalha com observadores/listeners */
5  public class Assunto implements InterfaceAssunto {
6
7      private List<Listener> listeners = new ArrayList<>();
8      private String valor;
9
10     public String getEstado() { return valor; }
11
12     public void setEstado(String valor) {
13         System.out.println("Alteração do valor para [" + valor + "]");
14         this.valor = valor;
15         notificarListeners();
16     }
17
18     @Override
19     public void registrar_listener(Listener l){
20         listeners.add(l);
21     }
22
23     @Override
24     public void remover_listener(Listener l){
25         listeners.remove(l);
26     }
27
28     @Override
29     public void notificarListeners(){
30         for (Listener l : listeners) {
31             l.update();
32         }
33     }
34 }
```

Listagem 8.3: Definindo a interface Listener.java

```
1  /** interface que define os observadores */
2  public abstract class Listener {
3
4      protected Assunto assunto;
5
6      public abstract void update();
7  }
```

Listagem 8.4: Definindo a classe concreta ObserverString.java

```
1  /* classe concreta de observador */
```



```
2 public class ObserverString extends Listener {
3
4     /* construtor*/
5     public ObserverString(Assunto assunto){
6         this.assunto = assunto; // note que não precisamos guardar esta referência
7         // a notificação poderia vir toda como parametro no update
8         this.assunto.registrar_listener(this); // adicionamos a classe que
           observamos
9     }
10
11     @Override
12     public void update() {
13         System.out.println( "ObserverString:_" + assunto.getEstado() );
14     }
15 }
```

Listagem 8.5: Definindo a classe concreta ObserverLower.java

```
1 /* classe concreta de observador */
2 public class ObserverLower extends Listener {
3
4     /* construtor*/
5     public ObserverLower(Assunto assunto){
6         this.assunto = assunto; // note que não precisamos guardar esta referência
7         // a notificação poderia vir toda como parametro no update
8         this.assunto.registrar_listener(this); // adicionamos a classe que
           observamos
9     }
10
11     @Override
12     public void update() {
13         System.out.println( "ObserverLower_:_" + assunto.getEstado().toLowerCase()
14         );
15     }
16 }
```

Listagem 8.6: Definindo a classe concreta ObserverUpper.java

```
1 /* classe concreta de observador */
2 public class ObserverUpper extends Listener {
3
4     /* construtor*/
5     public ObserverUpper(Assunto assunto){
6         this.assunto = assunto; // note que não precisamos guardar esta referência
7         // a notificação poderia vir toda como parametro no update
8         this.assunto.registrar_listener(this); // adicionamos a classe que
           observamos
9     }
10
11     @Override
12     public void update() {
13         System.out.println( "ObserverUpper_:_" + assunto.getEstado().toUpperCase()
14         );
15     }
16 }
```

Listagem 8.7: Definindo a classe principal ExemploObserver.java

```
1 public class ExemploObserver {
2
3     public static void main(String[] args) {
4
5         Assunto publicacao = new Assunto();
6     }
```



```
7      new ObserverString(publicacao);
8      new ObserverLower(publicacao);
9      new ObserverUpper(publicacao);
10
11      publicacao.setEstado("Texto_1");
12      publicacao.setEstado("Novo_Texto");
13  }
14 }
```

Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresentado na figura 8.2.

```
henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/observer
h3dema@casa-desktop:observer$ javac InterfaceAssunto.java Assunto.java Listener.
java ObserverString.java ObserverUpper.java ObserverLower.java ExemploObserver.j
ava
h3dema@casa-desktop:observer$ java ExemploObserver
Alteração do valor para [Texto 1]
ObserverString: Texto 1
ObserverLower : texto 1
ObserverUpper : TEXTO 1
Alteração do valor para [Novo Texto]
ObserverString: Novo Texto
ObserverLower : novo texto
ObserverUpper : NOVO TEXTO
h3dema@casa-desktop:observer$
```

Figura 8.2: Execução de ExemploObserver.

## 8.5 Conclusões

O padrão de projeto **Observer** é usado quando você precisa implementar um-para-muitos relacionamentos, por exemplo, a transmitir a mesma informação a vários ouvintes chamados observadores. The Observer padrão de design mantém fraco acoplamento entre o sujeito e os observadores porque a única coisa que o sujeito sabe sobre os observadores, é a interface, ou seja, que método para chamar a notifiá-la. O número de observadores pode ser arbitrária e alterado no o tempo de execução.







## Capítulo 9

# Padrão: Decorator

O padrão **Decorator** adiciona, de forma dinâmica, atributos e comportamentos adicionais a um objeto. O uso de um **Decorator** em nossa vida diária seria a aplicação de moldura em uma imagem. A imagem é o objeto que tem suas características próprias. Para a exibição da imagem, nós adicionamos uma moldura na imagem a fim de decorá-la, acrescentando características como suporte, resistência e beleza ao objeto original - a imagem.

### 9.1 O padrão Decorator

O padrão **decorator** anexa responsabilidades adicionais a um objeto dinamicamente. Os **decorators** fornecem uma alternativa flexível de subclasse para estender a funcionalidade. O padrão **decorator** tem como características:

- têm o mesmo supertipo que os objetos que eles decoram;
- podemos usar um ou mais decoradores para englobar um objeto;
- podemos passar um objeto decorado no lugar do objeto original (englobado), pois o decorador tem o mesmo supertipo que o objeto decorado
- adiciona seu próprio comportamento antes e/ou depois de delegar o objeto que ele decora o resto do trabalho;
- Os objetos podem ser decorados a qualquer momento, então podemos decorar os objetos de maneira dinâmica no tempo de execução com quantos decoradores desejarmos.

No diagrama de classes da figura 9.1 mostramos um exemplo de como são montadas as decorações para uma classe. A interface *Componente* é implementada pela classe *ComponenteReal1*. Este é o objeto que poderá ser decorado dinamicamente, adicionando um novo comportamento. As classes *Decorator* implementam a mesma interface abstrata que o componente que irão decorar. Já as classes *ConcreteDecorator* possuem uma variável de instância para a classe que será decorada. Os *ConcreteDecorator* também podem adicionar mais comportamentos (método *addedBehavior()* no diagrama *ConcreteDecoratorB*) ou mais atributos (*addedState* no diagrama *ConcreteDecoratorA*). Portanto a ideia basicamente é que através dos **Decorators** possamos adicionar comportamentos aos componentes bases

O Java também utiliza bastante o *padrão de projeto Decorator*. A API *java.io* é amplamente baseada nesse padrão de projeto. A figura 9.2 mostra um exemplo disto. A classe *InputStream* é uma classe abstrata que possui diversas implementações, como *FileInputStream*, *BufferedInputStream*, *GzipInputStream*, *ObjectInputStream*, etc. Cada uma das classes concretas possui um construtor que recebe como parâmetro uma instância da classe abstrata. Por exemplo, suponha

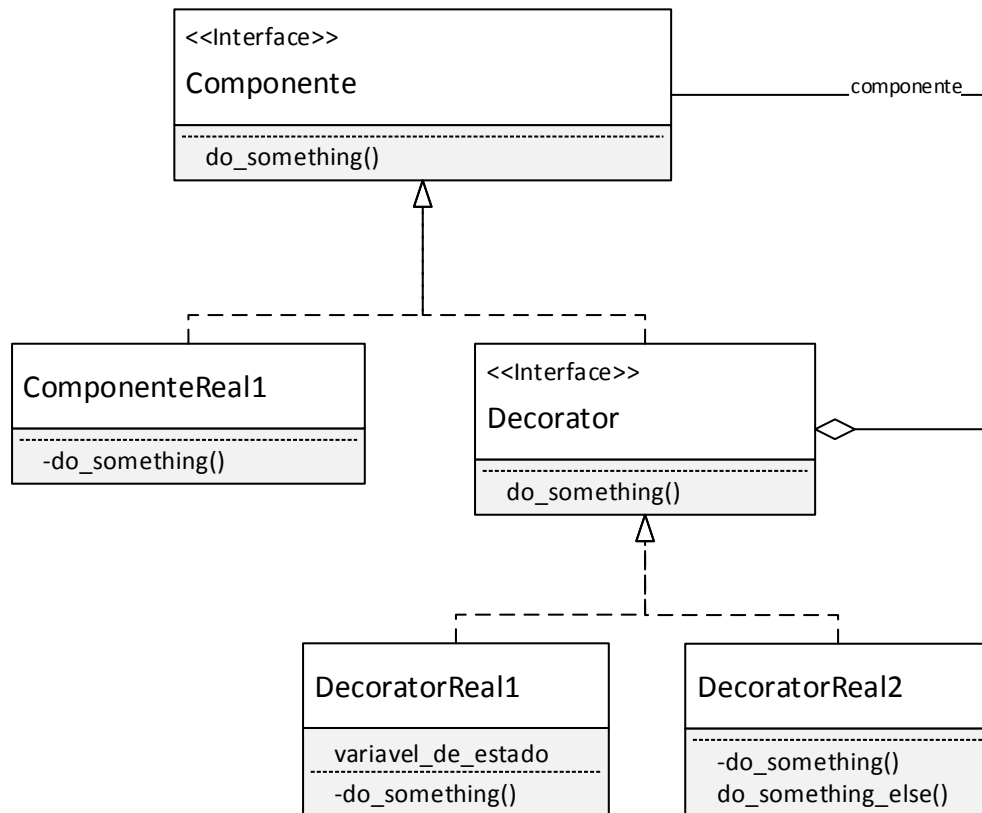


Figura 9.1: Diagrama de classes usando o Padrão Decorator.

que serializamos um conjunto de objetos em um arquivo compactado. Este objetos podem ser lidos pela sequencia abaixo:

```

// abre um stream com o arquivo onde estão os objetos compactados
FileInputStream fis = new FileInputStream("objetos.gz");

// para acelerar, fazemos um buffer em memória
BufferedInputStream bis = new BufferedInputStream(fis);

// descompactamos a stream já bufferizada
GzipInputStream gis = new GzipInputStream(bis);

// deserializamos os objetos da stream descompactada
ObjectInputStream ois = new ObjectInputStream(gis);

...

// podemos agora ler os objetos dentro do arquivo
Classe1 objetoClasse1 = (Classe1) ois.readObject();
  
```



Para fechar os arquivos, basta fechar a stream mais externa, que ela se encarrega de informar às demais que devem fechar também. Desta forma, no nosso exemplo, para fechar basta:

```
// fechamos todas as streams
ois.close();
```

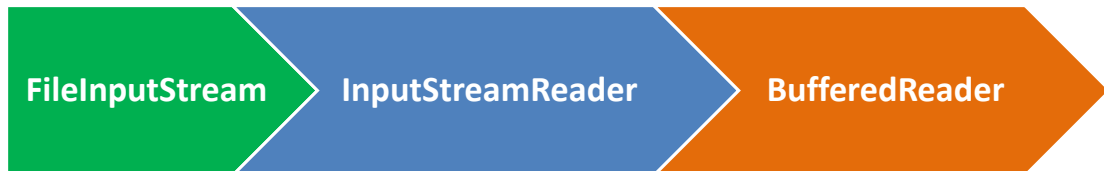


Figura 9.2: Diagrama de classe usando o Padrão **Decorator**.

Na figura 9.2 vemos uma sequência comum utilizada para leitura de dados de um arquivo. Abrimos o arquivo para leitura usando a classe `FileInputStream`. Utilizamos um decorator `InputStreamReader`, que faz a ponte entre uma stream de bytes e uma stream de caracteres. Utilizamos logo a seguir como decorator a classe `BufferedReader`, que lê o texto a partir de um fluxo de caracteres de entrada (fornecido por `InputStreamReader`), bufferizando os dados, de modo a permitir a leitura eficiente de caracteres, matrizes e linhas. Esta classe já possui métodos como `readLine()` que retorna uma `String` contendo uma linha lida do arquivo de dados.

## 9.2 Quando usar o padrão Decorator

O **Decorator** é mais utilizado quando quisermos adicionar responsabilidades a objetos dinamicamente e quando a extensão por subclasses é impraticável, pois teríamos muitas alterações e dessa forma diversas subclasses. O padrão **Decorator** deve ser usado quando:

- Objeto responsabilidades e comportamentos devem ser dinamicamente modificável; e
- Implementações concretas deve ser dissociado da responsabilidades e comportamentos.

Quando realizamos muita subclassificação obtemos um efeito não benéfico. A medida que você adiciona mais comportamentos a uma classe base, em breve você vai lidar com um pesadelo de manutenção. Isto ocorre porque uma nova classe é criada para cada combinação possível. Enquanto o decorator pode causar a sua própria questões, ele fornece uma alternativa melhor para demais subclasses.

Uma outra desvantagem do padrão é que teremos inúmeras classes pequenas. Isto pode tornar bastante complicado para um desenvolvedor que está tentando entender o funcionamento da aplicação.

## 9.3 Uma implementação

Vamos ver nesta seção um exemplo de uma classe que envia emails. Nosso sistema simples utiliza o recursos fornecido pelo padrão **Decorator** para acrescentar uma mensagem padronizada de aviso no final da mensagem de email. Utilizamos também um decorator para criptografar a mensagem (ou descriptografar).

Listagem 9.1: `IEmail.java`

```
1 public interface IEmail {
2     /**
3      * metodo a ser construido pelas classes derivadas desta interface
4      */
5 }
```



```

5 public String getConteudo();
6 }

```

Listagem 9.2: EmailSimples.java

```

1 /**
2  * classe concreta que implementa a interface IEmail
3  */
4 public class EmailSimples implements IEmail {
5
6     private String conteudo;
7
8     public EmailSimples(String valor) {
9         this.conteudo = valor;
10    }
11
12    /**
13     * retorna o conteudo do email
14     */
15    @Override
16    public String getConteudo() {
17        return conteudo;
18    }
19 }

```

Listagem 9.3: EmailDecorator.java

```

1 /**
2  * interface que define o decorator
3  */
4 public abstract class EmailDecorator implements IEmail {
5
6     IEmail emailOriginal;
7 }

```

Listagem 9.4: ExternalEmailDecorator.java

```

1 /**
2  * classe que implementa um Decorator concreto para IEmail
3  */
4 public class ExternalEmailDecorator extends EmailDecorator {
5
6     private final String DISCLAIMER = "\n\n"+
7     "=====\n"+
8     "Aviso: Embora a empresa tenha tomado todas as precauções para garantir que
9     nenhum vírus esteja presente neste e-mail, a empresa não pode
10    aceitar a responsabilidade por qualquer perda ou dano decorrente do uso
11    deste e-mail ou anexos.\n"+
12    "\n"+
13    "Este e-mail e quaisquer arquivos transmitidos com ele são confidenciais.
14    Destinam-se exclusivamente para o uso do indivíduo ou entidade a quem
15    se dirige.\n"+
16    "Se você recebeu esta mensagem por engano, por favor avise o gerente do
17    sistema.\n"+
18    "Esta mensagem contém informação confidencial e destina-se apenas para o
19    indivíduo nomeado.\n"+
20    "Se você não for o destinatário nomeado, você não deve divulgar, distribuir
21    ou copiar este e-mail.\n"+
22    "Por favor notifique o remetente imediatamente por e-mail se você tiver
23    recebido este e-mail por engano e elimine-o de seu sistema. Se você não
24    for o destinatário pretendido, esteja notificado de que a divulgação,
25    cópia, distribuição ou qualquer ação resultante do conteúdo desta
26    informação é estritamente proibida.";
27 }

```



```
16 public ExternalEmailDecorator(IEmail email) {
17     /** registra a classe original do Email */
18     emailOriginal = email;
19 }
20
21 @Override
22 public String getConteudo() {
23     // obtem o conteudo original do email que sera decorado
24     return addDisclaimer(emailOriginal.getConteudo());
25 }
26
27 /**
28  * procedimento que realiza a alteracao da funcionalidade
29  * @returns mensagem original adicionada do DISCLAIMER padronizado
30  */
31 private String addDisclaimer(String msg) {
32     return msg + DISCLAIMER;
33 }
34 }
```

Listagem 9.5: SecureEmailDecorator.java

```
1 /**
2  * outra classe concreta para uso do Decorator com IEmail
3  *
4  * implementa a encriptacao da mensagem
5  */
6
7 public class SecureEmailDecorator extends EmailDecorator {
8
9     String chave1 = "AbcDefGhi2468012"; // chave de 128 bit = 16 caracteres
10    String chave2 = "EstaEAChaveSecrt"; // chave de 128 bit
11
12    /**
13     * constroi classe, usa chaves de criptografia default
14     */
15    public SecureEmailDecorator(IEmail email) {
16        emailOriginal = email;
17    }
18
19    /**
20     * constroi classe, informando novas chaves de criptografia
21     */
22    public SecureEmailDecorator(IEmail email, String chave1, String chave2) {
23        this(email);
24        this.chave1 = chave1;
25        this.chave2 = chave2;
26    }
27
28    @Override
29    public String getConteudo() {
30        // encripta o texto original
31        return encriptar(emailOriginal.getConteudo());
32    }
33
34    private String encriptar(String msg) {
35        //encriptar a mensagem usando a classe Encryptor
36        return Cripto.encripta(chave1, chave2, msg);
37    }
38 }
```

Listagem 9.6: UnsecureEmailDecorator.java

```
1 /**
2  * outra classe concreta para uso do Decorator com IEmail
```



```
3  *
4  * implementa a decriptacao da mensagem
5  */
6
7  public class UnsecureEmailDecorator extends EmailDecorator {
8
9      /** devem ser as mesmas chaves de SecureEmailDecorator para poder
10       * descriptografar */
11      String chave1 = "AbcDefGhi2468012";
12      String chave2 = "EstaEAChaveSecrt";
13
14      /**
15       * constroi classe, usa chaves de criptografia default
16       */
17      public UnsecureEmailDecorator(IEmail email) {
18          emailOriginal = email;
19      }
20
21      /**
22       * constroi classe, informando novas chaves de criptografia
23       */
24      public UnsecureEmailDecorator(IEmail email, String chave1, String chave2) {
25          this(email);
26          this.chave1 = chave1;
27          this.chave2 = chave2;
28      }
29
30      @Override
31      public String getConteudo() {
32          // decripta o texto original
33          return decriptar(emailOriginal.getConteudo());
34      }
35
36      private String decriptar(String msg) {
37          // decriptar a mensagem usando a classe Encryptor
38          return Cripto.decripta(chave1, chave2, msg);
39      }
40  }
```

Listagem 9.7: EmissorEmail.java

```
1  public class EmissorEmail {
2
3      public void sendEmail(IEmail email) {
4          //read the email to-address, to see if it's going outside of the company
5          //if so decorate it
6          ExternalEmailDecorator e = new ExternalEmailDecorator(email);
7          SecureEmailDecorator es = new SecureEmailDecorator(e);
8          UnsecureEmailDecorator ue = new UnsecureEmailDecorator(es);
9          String conteudo = ue.getConteudo();
10         //enviar o email, que simulamos com um println
11         System.out.println("Enviado o email:");
12         System.out.println(conteudo);
13     }
14
15     public static void main(String[] args) {
16         EmissorEmail ee = new EmissorEmail();
17         EmailSimples email = new EmailSimples("Mensagem de teste da classe Email com decoradores");
18         ee.sendEmail(email);
19     }
20
21 }
```

Para funcionamento dos **Decorators** SecureEmailDecorator e UnsecureEmailDecorator precisamos da classe Cripto. Esta classe está detalhada na seção 17.



Podemos compilar as classes e, depois, rodar a classe principal. Obtemos:

```
henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/decorator
h3dema@casa-desktop:decorator$ javac IEmail.java EmailSimples.java EmailDecorator.j
ava ExternalEmailDecorator.java SecureEmailDecorator.java UnsecureEmailDecorator.jav
a EmissorEmail.java
h3dema@casa-desktop:decorator$
h3dema@casa-desktop:decorator$ java -cp ../utils/ApacheCommonsCodec/commons-codec-
1.10.jar EmissorEmail
Enviado o email:
Mensagem de teste da classe Email com decoradores

=====
Aviso: Embora a empresa tenha tomado todas as precauções para garantir que nenhum ví
rus estejam presentes neste e-mail, a empresa não pode aceitar a responsabilidade po
r qualquer perda ou dano decorrente do uso deste e-mail ou anexos.

Este e-mail e quaisquer arquivos transmitidos com ele são confidenciais. Destinam-se
exclusivamente para o uso do indivíduo ou entidade a quem se dirigem.
Se você recebeu esta mensagem por engano, por favor avise o gerente do sistema.
Esta mensagem contém informação confidencial e destina-se apenas para o indivíduo no
meado.
Se você não for o destinatário nomeado, você não deve divulgar, distribuir ou copiar
este e-mail.
Por favor notifique o remetente imediatamente por e-mail se você tiver recebido este
e-mail por engano e elimine-o de seu sistema. Se você não for o destinatário preten
dido, esteja notificado de que a divulgação, cópia, distribuição ou qualquer ação re
sultante do conteúdo desta informação é estritamente proibida.
h3dema@casa-desktop:decorator$
```

Figura 9.3: Execução de EmissorEmail.

Note que para rodar utilizando o comando *java* precisamos indicar o caminho da classe Base64 do pacote Apache Commons. Utilizamos para isto o parâmetro *-cp* para indicar o caminho da biblioteca.

## 9.4 Conclusões

O padrão Decorator usa a herança apenas para ter uma correspondência de tipo e não para obter o comportamento. Assim, quando compõe-se um decorador com um componente, adiciona-se um novo comportamento, nota-se que estamos adquirindo um novo comportamento e não herdando-o de alguma superclasse. Isso nos dá muito mais flexibilidade para compor mais objetos sem alterar uma linha de código, tudo em tempo de execução e não em tempo de compilação como ocorre com a herança.







## Capítulo 10

# Padrão: Factory

Na terminologia desenvolvimento orientado a objeto, uma fábrica é uma classe para a criação de outro objectos. Normalmente, essa classe tem métodos que aceitam alguns parâmetros e retornos algum tipo de objecto, dependendo dos parâmetros passados. Neste capítulo iremos abordar três assuntos:

1. Como fazemos para criar uma fábrica simples
2. O que é o **Factory Method**? Quando devemos usá-lo? Como implementar o **Factory Method**? Veremos um exemplo da construção de uma ferramenta que pode ser conectado a uma variedade de recursos de web.
3. O que é o **Abstract Factory**? Quando devemos usá-lo? Qual é a diferença para o **Factory Method**?

### 10.1 Por que devemos usar este padrão ?

Então, por que deveríamos nos incomodar com as fábricas em vez de usar objeto direto instanciação?

1. Fábricas fornecer baixo acoplamento, que separa a criação do objeto de usar implementação de classe específico.
2. Uma classe que usa o objeto criado não precisa saber exatamente qual classe é criado. Tudo o que precisa de saber é a interface criada classe ", ou seja, que métodos de classe criado 'pode ser chamado e com que argumentos. Adicionando novas classes é feita apenas em fábricas, desde que as novas classes cumprir a interface, sem modificar o código do cliente.
3. A classe de fábrica pode reutilizar objetos existentes, enquanto a instanciação direta sempre cria um novo objeto

### 10.2 Factory Method

Como mostrado no diagrama da figura 10.1, normalmente este padrão de projeto tem uma classe abstrata, que denominamos no exemplo como **Criador**, que contém o *factory\_method* que é responsável para a criação de algum tipo de objetos. O método *outro\_metodo* então trabalha com o objeto criado.

A classe **CriadorReal** pode redefinir o *factory\_method* para alterar o objeto criado no tempo de execução. O método *outro\_metodo* não se importa com o objeto que é criado contanto este novo

objeto implemente a interface do produto e forneça para a aplicação todos os métodos definidos na interface.

A essência deste padrão é definir uma interface para criar um objeto, mas deixar que a classe que implementa a interface decida qual classe instanciar. A interface é *factory\_method* nas classes **CriadorReal** e **Criador**, que decide qual subclasse de Produto irá criar. O funcionamento do **Factory Method** é baseado em herança. A criação do objeto é delegada às subclasses que implementam **Factory Method** para a criação do objeto.

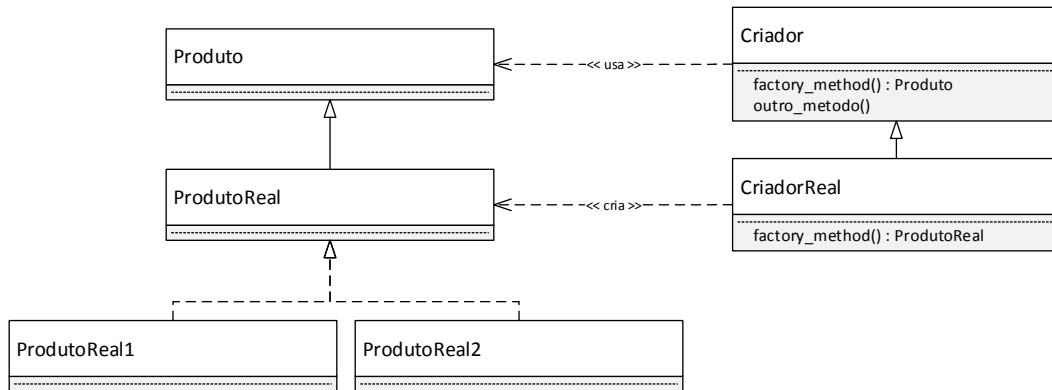


Figura 10.1: Exemplificando o padrão Factory Method

### 10.2.1 Vantagens de usar o padrão Factory Method

As principais vantagens da utilização do padrão **Factory Method** são:

- O uso deste padrão torna o código mais universal, pois não está vinculado a classes concretas (como **ProdutoReal**, **ProdutoReal1** e **ProdutoReal2**). O código fica vinculado a interfaces (**Produto**) fornecendo de baixo acoplamento. Desta forma o padrão separa as interfaces de suas implementações.
- Ele dissocia o código que cria objetos a partir do código que os usa, reduzindo a complexidade da manutenção. Para adicionar uma nova classe, você precisa adicionar uma cláusula *else* adicional.

## 10.3 Implementação com o padrão Factory Method

Listagem 10.1: Definindo a interface Forma.java

```
1  /* interface que define o objeto genérico que será desenhado */
2  public interface Forma {
3      /* método que todas as formas devem implementar */
4      void desenha();
5  }
```

Listagem 10.2: Definindo a classe concreta Elipse.java

```
1  public class Elipse implements Forma {
2
3      public static final String NOMEFORMA = "Elipse";
```



```
4
5     @Override
6     public void desenha() {
7         System.out.println("Dentro de Elipse:desenha().");
8     }
9
10 }
```

Listagem 10.3: Definindo a classe concreta Circunferencia.java

```
1 public class Circunferencia implements Forma {
2
3     public static final String NOMEFORMA = "Circunferencia";
4
5     @Override
6     public void desenha() {
7         System.out.println("Dentro de Circunferencia:desenha().");
8     }
9
10 }
```

Listagem 10.4: Definindo a classe concreta Quadrado.java

```
1 public class Quadrado implements Forma {
2
3     public static final String NOMEFORMA = "Quadrado";
4
5     @Override
6     public void desenha() {
7         System.out.println("Dentro de Quadrado:desenha().");
8     }
9
10 }
```

Listagem 10.5: Definindo a classe concreta Retangulo.java

```
1 public class Retangulo implements Forma {
2
3     public static final String NOMEFORMA = "Retângulo";
4
5     @Override
6     public void desenha() {
7         System.out.println("Dentro de Retangulo:desenha().");
8     }
9
10 }
```

Listagem 10.6: Definindo a classe FabricaForma.java que implementa **Factory Method**

```
1 /** classe que fabrica as formas conforme parametro de getForma() */
2 public class FabricaForma {
3
4     /**
5      * método que efetua a fabricação das formas
6      */
7     public Forma getForma(String tipo){
8
9         switch (tipo) {
10             case Retangulo.NOMEFORMA: return new Retangulo();
11             case Quadrado.NOMEFORMA: return new Quadrado();
12             case Circunferencia.NOMEFORMA: return new Circunferencia();
13             case Elipse.NOMEFORMA: return new Elipse();
14             default: return null;
15         }
16     }
```



```

16     }
17 }

```

Listagem 10.7: Definindo a classe principal ExemploFactoryMethod.java

```

1 public class ExemploFactoryMethod {
2
3     public static void main(String[] args) {
4         /* cria a fabrica */
5         FabricaForma fabrica = new FabricaForma();
6
7         Forma f1 = fabrica.getForma("Circunferencia");
8         f1.desenha();
9
10        Forma f2 = fabrica.getForma("Quadrado");
11        f2.desenha();
12
13        Forma f3 = fabrica.getForma("Elipse");
14        f3.desenha();
15
16        Forma f4 = fabrica.getForma("Retângulo");
17        f4.desenha();
18    }
19 }

```

Podemos compilar as classes e, depois, rodar a classe principal. Obtemos:

```

henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/factory_method
h3dema@casa-desktop:factory_method$ javac Forma.java Elipse.java Circunferencia.java
Quadrado.java Retangulo.java FabricaForma.java ExemploFactoryMethod.java
h3dema@casa-desktop:factory_method$ java ExemploFactoryMethod
Dentro de Circunferencia:desenha().
Dentro de Quadrado:desenha().
Dentro de Elipse:desenha().
Dentro de Retangulo:desenha().
h3dema@casa-desktop:factory_method$

```

Figura 10.2: Execução de ExemploFactoryMethod.

## 10.4 Abstract Factory

**Abstract Factory** é usado quando você precisa criar uma família de objetos que realizam juntos alguma tarefa.

Se o objetivo do **Factory Method** é criar objetos de classes relacionadas, o objetivo do padrão **Abstract Factory** é criar famílias de objetos relacionados sem depender de suas classes específicas. No diagrama da figura 10.3, vemos um exemplo onde todas as fábricas (**Fabrica1** e **Fabrica2**) são derivadas da interface **Abstract Factory**. Desta forma as duas fábricas devem implementar os métodos para criar instâncias de duas interfaces **ProdutoAbstrato** e **Outro-ProdutoAbstrato**.

A ideia é que as fábricas que criam os objetos devem ter a mesma interface, enquanto que os objetos concretos são criados em cada fábrica de forma diferente. Então, se você deseja obter um comportamento diferente, pode alterar a fábrica em tempo de execução e obter um conjunto completo de objetos diferentes.



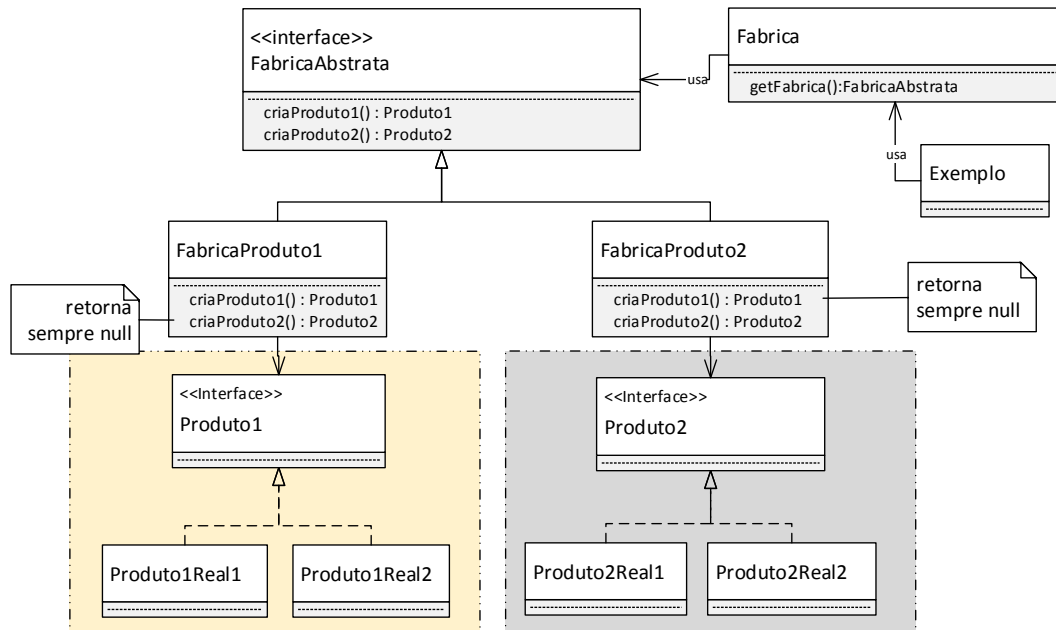


Figura 10.3: Exemplificando o padrão AbstractFactory

#### 10.4.1 Vantagens de usar o padrão Abstract Factory

A vantagem de usar **Abstract Factory** é que isola a criação de objetos a partir de o cliente que precisa deles, dando ao cliente apenas a possibilidade de acessá-los através de uma interface, o que torna a manipulação mais fácil.

As principais vantagens da utilização do padrão **Abstract Factory** são os seguintes:

- simplifica a substituição de famílias de produtos
- assegura a compatibilidade dos produtos da família do produto
- isola as classes concretas do cliente

### 10.5 Uma implementação usando o padrão Abstract Factory

Vemos na figura 10.3 um exemplo deste padrão. Se os produtos de uma família são destinados a trabalhar juntos, a classe *FabricaAbstrata* facilita para na criação de fábricas capazes de gerar os objetos a partir de uma única família de cada vez. Por outro lado, a adição de novos tipos de produtos para as fábricas existentes é fácil pois a interface *FabricaAbstrata* depende somente de cada fábrica. Cada fábrica pode ser alterada, atualizando os produtos que podem ser criados. Contudo a criação de uma nova fábrica alterar a interface de fábrica e todos os seus descendentes como por exemplo *Fabrica*.

Listagem 10.8: Interface para o primeiro tipo de produto - Produto1

```

1 public interface Produto1 {
2
3     public String getNome();

```



```
4 public double getPreco();
5 public void atualizaPreco(double novo_preco);
6
7 }
```

Listagem 10.9: Implementação de Produto1 - Produto1Real

```
1 class Produto1Real implements Produto1 {
2
3     private String nome;
4     private double preco;
5
6     Produto1Real(String nome, double preco) {
7         this.nome = nome;
8         this.preco = preco;
9     }
10
11     @Override
12     public void atualizaPreco(double novo_preco) {
13         preco = novo_preco;
14     }
15
16     @Override
17     public String getNome() { return nome; }
18
19     @Override
20     public double getPreco() { return preco; }
21
22     public String toString() {
23         return "Produto_["+nome+"]_custa_"+preco;
24     }
25 }
```

Listagem 10.10: Outra implementação de Produto1 -Produto1Especial

```
1 class Produto1Especial implements Produto1 {
2
3     private String nome;
4     private double preco;
5
6     Produto1Especial(String nome, double preco) {
7         this.nome = nome;
8         this.preco = preco;
9     }
10
11     @Override
12     public void atualizaPreco(double novo_preco) {
13         preco = novo_preco;
14     }
15
16     @Override
17     public String getNome() { return nome; }
18
19     @Override
20     public double getPreco() { return preco; }
21
22     public String toString() {
23         return "Produto_["+nome+"]_’especial’_custa_"+preco;
24     }
25 }
```

Listagem 10.11: Interface para o segundo tipo de produto - Produto2

```
1 public interface Produto2 {
```



```
2
3 public String getNome();
4 public double getPreco();
5 public void atualizaMargem(double nova_margem);
6 public void atualizaCusto(double novo_custo);
7
8 }
```

Listagem 10.12: Implementação de Produto2 - Produto2Real

```
1 class Produto2Real implements Produto2 {
2
3     private String nome;
4     private double custo;
5     private double margem;
6
7     Produto2Real(String nome, double custo, double margem) {
8         this.nome = nome;
9         this.custo = custo;
10        this.margem = margem;
11    }
12
13    @Override
14    public void atualizaMargem(double nova_margem) {
15        margem = nova_margem;
16    }
17
18    @Override
19    public void atualizaCusto(double novo_custo) {
20        custo = novo_custo;
21    }
22
23    @Override
24    public String getNome() { return nome; }
25
26    @Override
27    public double getPreco() { return custo*(1+margem); }
28
29    public String toString() {
30        return "Produto_["+nome+"]_custo_"+getPreco();
31    }
32 }
```

Listagem 10.13: Outra implementação de Produto2 -Produto2Especial

```
1 class Produto2Especial implements Produto2 {
2
3     private String nome;
4     private double custo;
5     private double margem;
6
7     Produto2Especial(String nome, double custo, double margem) {
8         this.nome = nome;
9         this.custo = custo;
10        this.margem = margem;
11    }
12
13    @Override
14    public void atualizaMargem(double nova_margem) {
15        margem = nova_margem;
16    }
17
18    @Override
19    public void atualizaCusto(double novo_custo) {
20        custo = novo_custo;
21    }
22 }
```



```

21     }
22
23     @Override
24     public String getNome() { return nome; }
25
26     @Override
27     public double getPreco() { return custo*(1+margem); }
28
29     public String toString() {
30         return "Produto_"+"nome+"+"_'especial'_"custa_"+"getPreco();
31     }
32 }

```

Listagem 10.14: Interface para as fábricas - FabricaAbstrata

```

1  /**
2   * classe abstract que implementa a geracao de
3   * fabrica para a criaçao de objetos dos dois tipos de classe
4   */
5  public abstract class FabricaAbstrata {
6
7      abstract Produto1 getProduto1(String tipo, String nome, double preco);
8      abstract Produto2 getProduto2(String tipo, String nome, double custo, double
9          margem);
10 }

```

Listagem 10.15: Fábrica capaz de produzir os objetos do tipo Produto1 -Fabrica1

```

1  class Fabrica1 extends FabricaAbstrata {
2      // implementa somente o metodo que gera o Produto1
3      @Override
4      Produto1 getProduto1(String tipo, String nome, double preco) {
5          switch (tipo) {
6              case "Normal" : return new Produto1Real(nome, preco);
7              case "Especial" : return new Produto1Especial(nome, preco);
8              default: return null;
9          }
10     }
11     @Override
12     Produto2 getProduto2(String tipo, String nome, double custo, double margem) {
13         return null;}
14 }

```

Listagem 10.16: Fábrica capaz de produzir os objetos do tipo Produto2 -Fabrica2

```

1  class Fabrica2 extends FabricaAbstrata {
2      // tem que existir esta implementaçao minima que retorna null
3      @Override
4      Produto1 getProduto1(String tipo, String nome, double preco) { return null; }
5      // implementa somente o metodo que gera o Produto1
6      @Override
7      Produto2 getProduto2(String tipo, String nome, double custo, double margem) {
8          switch (tipo) {
9              case "Normal" : return new Produto2Real(nome, custo, margem);
10             case "Especial" : return new Produto2Especial(nome, custo, margem);
11             default: return null;
12         }
13     }
14 }

```

Listagem 10.17: Fábrica capaz de produzir as fábricas - Fabrica

```

1  /**

```





```

2  * classe que devolve via getFabrica() a fabrica especificada em "tipo"
3  */
4  class Fabrica {
5
6      /**
7       * @param tipo define qual a fábrica será criada
8       */
9      FabricaAbstrata getFabrica(String tipo) {
10         switch (tipo) {
11             case "Fabrica1": return new Fabrica1();
12             case "Fabrica2": return new Fabrica2();
13             default: return null;
14         }
15     }
16 }

```

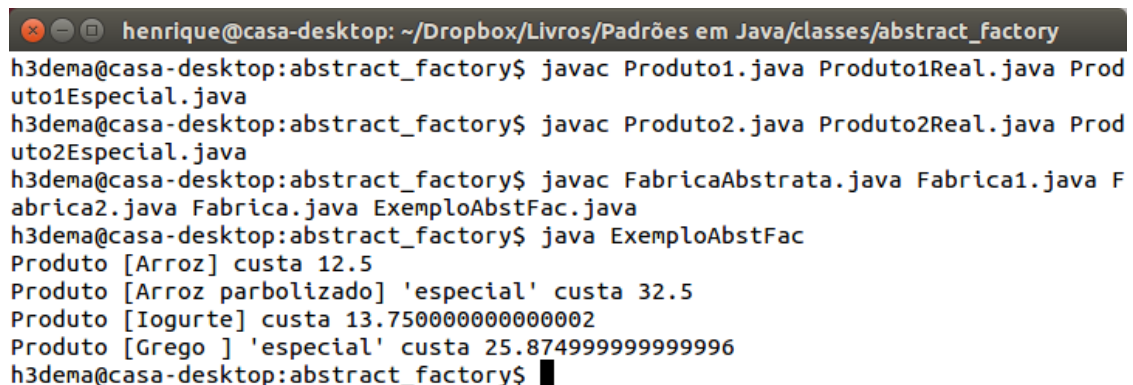
Listagem 10.18: Definindo a classe principal ExemploAbstFac.java

```

1  class ExemploAbstFac {
2
3      public static void main(String[] args) {
4          Fabrica fab = new Fabrica();
5          FabricaAbstrata fabProduto1 = fab.getFabrica("Fabrica1");
6          FabricaAbstrata fabProduto2 = fab.getFabrica("Fabrica2");
7
8          Produto1 p1 = fabProduto1.getProduto1("Normal", "Arroz", 12.5 );
9          Produto1 p2 = fabProduto1.getProduto1("Especial", "Arroz parbolizado", 32.5
10         );
11          Produto2 p3 = fabProduto2.getProduto2("Normal", "Iogurte", 12.5, 0.1 );
12          Produto2 p4 = fabProduto2.getProduto2("Especial", "Grego", 22.5, 0.15);
13
14          System.out.println(p1);
15          System.out.println(p2);
16          System.out.println(p3);
17          System.out.println(p4);
18      }
19
20 }

```

Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresenta na figura 10.4.



```

henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/abstract_factory
h3dema@casa-desktop:abstract_factory$ javac Produto1.java Produto1Real.java Produto1Especial.java
h3dema@casa-desktop:abstract_factory$ javac Produto2.java Produto2Real.java Produto2Especial.java
h3dema@casa-desktop:abstract_factory$ javac FabricaAbstrata.java Fabrica1.java Fabrica2.java Fabrica.java ExemploAbstFac.java
h3dema@casa-desktop:abstract_factory$ java ExemploAbstFac
Produto [Arroz] custa 12.5
Produto [Arroz parbolizado] 'especial' custa 32.5
Produto [Iogurte] custa 13.750000000000002
Produto [Grego ] 'especial' custa 25.874999999999996
h3dema@casa-desktop:abstract_factory$

```

Figura 10.4: Execução de ExemploAbstFac.



## 10.6 Abstract Factory ou Factory Method ?

Para saber quando utilizamos **Abstract Factory** ou **Factory Method**, devemos olhar para as instâncias que são geradas em cada caso. Vemos portanto que

- Devemos usar o padrão **Factory Method** quando há uma necessidade de dissociar um cliente a partir de um determinado produto que ele usa. Este padrão serve para aliviar um cliente da responsabilidade de criar e configurar as instâncias de um produto.
- Use o padrão **Abstract Factory** quando os clientes deve ser dissociado da classes de produtos. Este padrão também pode impor restrições especificando quais classes devem ser usados com os outros, criando de forma independente famílias de objetos. Muitas vezes observamos que para a criação de cada família é utilizado o **Factory Method**.

## Capítulo 11

# Padrão: Composite

Padrão **Composite** é utilizado onde precisamos tratar um todo um grupo de objetos da mesma forma, isto é, como um único objeto. O padrão **Composite** arranja os objetos em termos de uma estrutura de árvore para representar parte, bem como toda a hierarquia. Este tipo de *padrão de projeto* cria uma estrutura de árvore com o grupo de objetos.

Esse padrão cria uma classe que contém um grupo de seus próprios objetos. Essa classe fornece maneiras de modificar seu grupo de objetos.

### 11.1 O padrão Composite

Cliente usa a interface da classe - *Componente* para interagir com objetos na coleção. Se o elemento for uma folha, a requisição é tratada diretamente pelo objeto. Se o elemento é uma *Composicao*, então ele geralmente encaminha as solicitações aos seus componentes filhos.

O que torna o padrão **Composite** interessante é o poder de recursão na estrutura de objetos que compõem a coleção. A estrutura de classes deste padrão é mostrada na figura 11.1. Cada elemento é descrito na tabela 11.1.

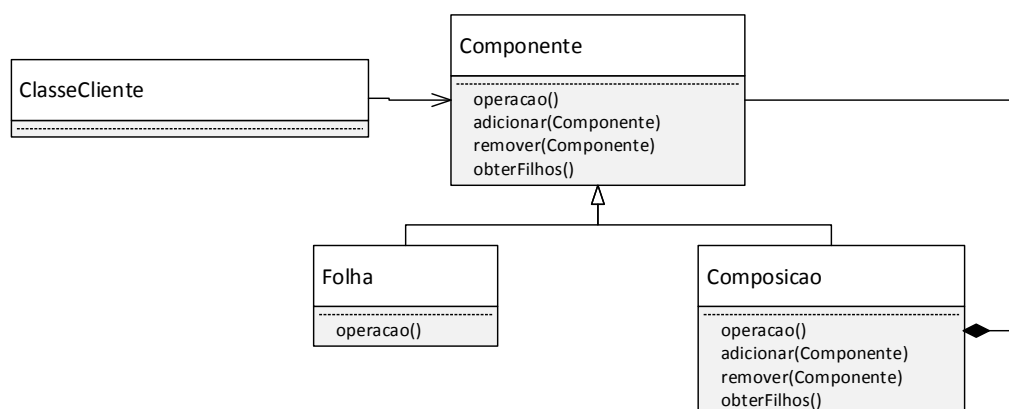


Figura 11.1: Estrutura de classes do padrão **Composite**.

Classe	Características
Componente	Declara interface para objetos na composição; Implementa comportamento padrão para a interface comum a todas as classes, conforme apropriado; Declara uma interface para acessar e gerenciar seus componentes filhos.
Folha	Representa objetos Folha na coleção; Não tem filhos; Define o comportamento de objetos primitivos na composição.
Composição	Define o comportamento para componentes de ter filhos; Armazena componentes filhos; Implementa operações relacionadas com as crianças na interface do Componente.
Cliente	Manipula objetos na composição através da interface do componente.

Tabela 11.1: Elementos (classes) do padrão **Composite**

## 11.2 Quando usar o padrão Composite

Devemos utilizar o padrão Composite quando:

- Queremos representar hierarquias de objetos do tipo parte-todo;
- Queremos que o cliente seja capaz de ignorar a diferença entre as composições de objetos e os objetos individuais. As classes cliente deverão tratar todos os objetos na estrutura composta de maneira uniforme.

Este padrão apresenta algumas desvantagens, como:

- Uma vez que a estrutura da árvore é definida, o padrão torna a árvore excessivamente geral.
- A classe Folha tem que criar alguns métodos vazios (sem propósito, exceto implementar a interface).

## 11.3 Uma implementação

Apresentaremos a seguir uma implementação que utiliza o padrão Composite. Iremos mostrar toda a estrutura hierárquica de uma empresa utilizando este padrão. Para simplificar não criaremos a interface *Componente* e já implementaremos diretamente a composição representada pela classe *Empregado*. Esta classe possui uma composição com si própria. Desta forma podemos cadastrar as pessoas e suas relações hierárquicas. Os subordinados de um empregado portanto serão os objetos filhos, como veremos na implementação.

Neste nosso exemplo, temos somente duas classes. A classe *Empregado* que implementa o padrão **Composite** para representar a hierarquia. Esta classe é mostrada na listagem 11.1. A classe *ExemploComposite*, mostrada na listagem 11.2, é a classe principal que instancia *Empregado* criando a empresa.

Listagem 11.1: Definindo a classe *Empregado.java*

```

1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Empregado {
5
6     private static int id_count = 0;
7
8     private int id;
```



```
9 private String nome;
10 private String depto;
11 private float salarioBase;
12 private Integer ramal = null;
13 private String email = null;
14 private List<Empregado> subordinados;
15
16 /** constructor */
17 public Empregado(String nome, String depto, int salInicial) {
18     this.id = id_count++;
19     this.nome = nome;
20     this.depto = depto;
21     this.salarioBase = salInicial;
22     subordinados = new ArrayList<Empregado>();
23 }
24
25 /** dados dos funcionario */
26 public String getNome() { return nome; }
27
28 public String getDepto() { return depto; }
29 public void setDepto(String depto) { this.depto = depto; }
30
31 public Integer getRamal() { return ramal; }
32 public void setRamal(Integer ramal) { this.ramal = ramal; }
33
34
35 public String getEmail() { return email == null ? "Funcionario sem email" :
    email; }
36 public void setEmail(String email) { this.email = email; }
37
38 /** subordinados */
39 public void adicionaSubordinado(Empregado e) {
40     subordinados.add(e);
41 }
42
43 public void removeSubordinado(Empregado e) {
44     subordinados.remove(e);
45 }
46
47 public List<Empregado> getSubordinado(){ return subordinados; }
48
49 public String toString(){
50     return ("Funcionario [Id: " + id + " ] Nome: " + nome + ", depto: " +
        depto + ", salário: " + salarioBase);
51 }
52 }
```

Neste exemplo criamos primeiro o CEO. Abaixo dele estão todos os demais empregados da empresa. Criamos 2 diretores, que são inseridos como componentes (filhos) do CEO. Criamos ainda 4 empregados. Dois são colocados como

Listagem 11.2: Definindo a classe principal ExemploComposite.java

```
1 public class ExemploComposite {
2
3     public static void main(String[] args) {
4
5         Empregado CEO = new Empregado("João","Presidente", 30000);
6
7         Empregado vpVendas = new Empregado("Roberto","Diretoria_Vendas", 20000);
8
9         Empregado vpMarketing = new Empregado("Maria","Diretoria_Marketing",
            20000);
10
11         Empregado f1 = new Empregado("Laura","Marketing", 10000);
12         Empregado f2 = new Empregado("José","Marketing", 10000);
```

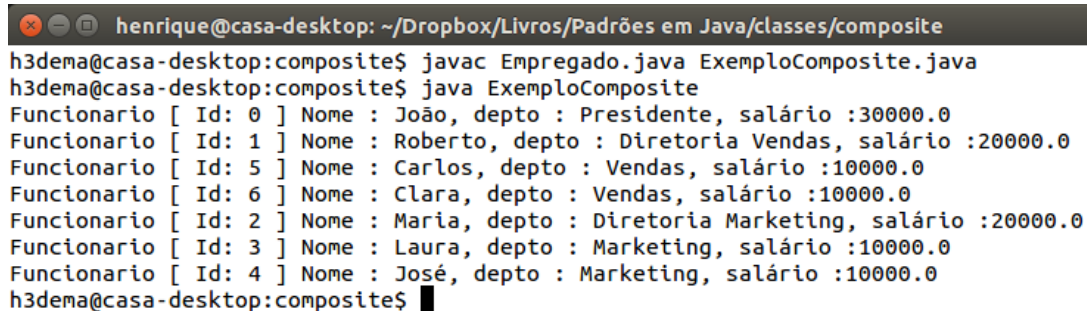


```

13      Empregado v1 = new Empregado("Carlos","Vendas", 10000);
14      Empregado v2 = new Empregado("Clara","Vendas", 10000);
15
16      CEO.adicionaSubordinado(vpVendas);
17      CEO.adicionaSubordinado(vpMarketing);
18
19      vpMarketing.adicionaSubordinado(f1);
20      vpMarketing.adicionaSubordinado(f2);
21
22      vpVendas.adicionaSubordinado(v1);
23      vpVendas.adicionaSubordinado(v2);
24
25      // Vamos agora imprimir todos os funcionários da empresa a partir do
26      // Presidente
27      System.out.println(CEO);
28      // Agora olhamos os subordinados do presidente
29      for (Empregado f : CEO.getSubordinado()) {
30          System.out.println(f);
31
32          for (Empregado s : f.getSubordinado()) {
33              System.out.println(s);
34          }
35      }
36  }
37 }

```

Podemos compilar as classes e, depois, rodar a classe principal. Obtemos:



```

henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/composite
h3dema@casa-desktop:composite$ javac Empregado.java ExemploComposite.java
h3dema@casa-desktop:composite$ java ExemploComposite
Funcionario [ Id: 0 ] Nome : João, depto : Presidente, salário :30000.0
Funcionario [ Id: 1 ] Nome : Roberto, depto : Diretoria Vendas, salário :20000.0
Funcionario [ Id: 5 ] Nome : Carlos, depto : Vendas, salário :10000.0
Funcionario [ Id: 6 ] Nome : Clara, depto : Vendas, salário :10000.0
Funcionario [ Id: 2 ] Nome : Maria, depto : Diretoria Marketing, salário :20000.0
Funcionario [ Id: 3 ] Nome : Laura, depto : Marketing, salário :10000.0
Funcionario [ Id: 4 ] Nome : José, depto : Marketing, salário :10000.0
h3dema@casa-desktop:composite$

```

Figura 11.2: Execução de ExemploComposite.



## Capítulo 12

# Padrão: Command

Imagine que você está escrevendo o programa de impressora e deseja implementar o spooler da impressora. Qual é a maneira mais fácil de fazê-lo? Crie uma classe com Spooler métodos para adicionar e remover as tarefas da impressora. A maneira mais fácil de executar as tarefas da impressora é para criar um objeto, que contém todas as informações necessárias: o texto a imprimir, o número de cópias, cor, qualidade, e assim por diante. O spooler precisará chamar o método execute de o trabalho de impressão, eo trabalho de impressão vai cuidar de tudo por si só. É assim que o padrão de comando funciona: você cria um objeto, o que representa e encapsula todas as informações necessárias para chamar um método em um momento posterior. Este informações incluem o nome do método, o objeto que possui o método, e valores para os parâmetros do método.

Você é capaz de passar esses objetos de comando para qualquer código que sabe como chamar a sua método execute, salve esses objetos, e retorno de métodos como qualquer outro objeto. Com o padrão **Command**, você é capaz de passar a instância de comando como um observador e quando o assunto notifica os observadores sobre algum evento, a instância **Command** será chamado e algum trabalho encapsulado em que vai ser feito.

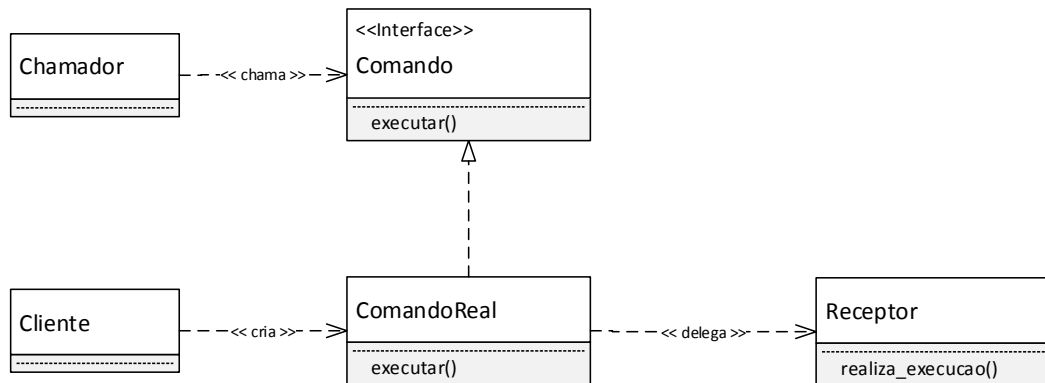
### 12.1 O padrão Command

Para entender como o padrão de comando funciona, vamos definir algumas terminologias:

- **Command**: Esta é uma interface para a execução de uma operação.
- **ConcreteCommand**: Essa classe estende a interface de comando e implementa o método execute. Essa classe cria uma ligação entre o ação eo receptor.
- **Cliente**: Essa classe cria a classe ConcreteCommand e associa-o com o receptor.
- **Chamador (Invoker)**: Esta classe pede o comando para realizar o pedido.
- **Receptor**: Esta classe sabe como executar a operação.

No diagrama da figura 12.1, a classe Chamador chama o método execute de um objeto com a interface de comando. Na verdade, é um objeto da classe ConcreteCommand, em que o método de execução chama um objeto da classe do receptor de que faz algum trabalho real.

O padrão de projeto Command fornece uma interface para chamar um método para executar algum trabalho e encapsula todas as informações necessárias para fazê-lo em uma chamada de objeto, que pode ser executado separadamente após instanciação.

Figura 12.1: Uso do padrão **Command**.

## 12.2 Problemas resolvidos com o padrão Command

É melhor aplicar o comando nos seguintes casos:

- Quando você precisa manter um histórico de pedidos. Um chamador pode salvar comando instâncias depois de chamar o método execute para implementar a funcionalidade história em algum lugar.
- Quando você precisa para implementar a funcionalidade de retorno de chamada. Se você passar para o dois objetos invoker um após o outro, o segundo objeto será um callback pela primeira vez.
- Quando você precisar que os pedidos sejam manipuladas em alturas variantes ou em ordens variantes.
- Para conseguir isso, você pode passar os objetos de comando para diferentes invokers, que são invocados por condições diferentes.
- Quando o solicitante deve ser dissociada da manipulação de objeto a invocação.
- Quando você precisa para implementar a funcionalidade de desfazer. Para conseguir isso, você precisa definir um método que cancela uma operação realizada na executar método. Por exemplo, se você criou um arquivo, você precisa excluí-lo.

## 12.3 Vantagens e desvantagens do padrão Command

Os prós e contras de o padrão de design Command são as seguintes:

- É útil na criação de uma estrutura, especialmente quando a criação de e um pedido de execução não são dependentes uns dos outros. Isso significa que o Instância de comando pode ser instanciado pelo Cliente, mas algum tempo depois correr por o Invoker, eo cliente e Invoker pode não saber nada sobre entre si.
- Este padrão de ajuda em termos de extensibilidade como podemos adicionar um novo comando sem alterar o código existente.





- Ele permite que você crie uma sequência de comandos chamados macro. Para executar o macro, criar uma lista de instâncias de comando e chamar o método execute de todos os comandos.
- A principal desvantagem do padrão de comando é o aumento do número de aulas para cada comando individual.

## 12.4 Uma implementação

Listagem 12.1: Definindo a classe AcaoBolsa.java

```
1  /** classe que fará as requisições */
2  public class AcaoBolsa {
3
4      private String nome;
5      private int qtd;
6
7      public AcaoBolsa(String nome, int qtd) {
8          this.nome = nome;
9          this.qtd = qtd;
10     }
11
12     public void comprar(){
13         System.out.println("Ação [Nome: "+nome+",Quantidade: " + qtd +"]
14         compradas");
15     }
16     public void vender(){
17         System.out.println("Ação [Nome: "+nome+",Quantidade: " + qtd +"]
18         vendidas");
19     }
20 }
```

Listagem 12.2: Definindo a classe de interface dos comandos OrdemBolsa.java

```
1  public interface OrdemBolsa {
2
3      void executar();
4
5  }
```

Listagem 12.3: Definindo a classe comando ComprarAcaoBolsa.java

```
1  /** comando de compra */
2  public class ComprarAcaoBolsa implements OrdemBolsa {
3
4      private AcaoBolsa acao;
5
6      public ComprarAcaoBolsa(AcaoBolsa acao){
7          this.acao = acao;
8      }
9
10     public void executar() {
11         acao.comprar();
12     }
13 }
```

Listagem 12.4: Definindo a classe comando VenderAcaoBolsa.java

```
1  /** comando de venda */
2  public class VenderAcaoBolsa implements OrdemBolsa {
3
```



```

4     private AcaoBolsa acao;
5
6     public VenderAcaoBolsa(AcaoBolsa acao){
7         this.acao = acao;
8     }
9
10    public void executar() {
11        acao.vender();
12    }
13 }

```

Listagem 12.5: Definindo a classe Broker.java

```

1 import java.util.ArrayList;
2 import java.util.List;
3
4     public class Broker {
5
6         private List<OrdemBolsa> listaDeOrdensBolsa = new ArrayList<>();
7
8         public void aceitarOrdem(OrdemBolsa ordem){
9             listaDeOrdensBolsa.add(ordem);
10        }
11
12        public void realizaOrdensBolsa(){
13            for (OrdemBolsa o : listaDeOrdensBolsa) {
14                o.executar();
15            }
16            listaDeOrdensBolsa.clear();
17        }
18    }

```

Listagem 12.6: Definindo a classe principal ExemploCommand.java

```

1 public class ExemploCommand {
2
3     public static void main(String[] args) {
4         Broker broker = new Broker();
5
6         /** cria as ordens para o Broker */
7         broker.aceitarOrdem(
8             new ComprarAcaoBolsa(new AcaoBolsa("ECORODOVIASON_N1", 755)));
9         broker.aceitarOrdem(
10            new VenderAcaoBolsa(new AcaoBolsa("ELETROBRASPNB_N1", 904)));
11        broker.aceitarOrdem(
12            new ComprarAcaoBolsa(new AcaoBolsa("ELETROBRASON_N1", 617)));
13        broker.aceitarOrdem(
14            new ComprarAcaoBolsa(new AcaoBolsa("SABESPON_N1", 1781)));
15        broker.aceitarOrdem(
16            new VenderAcaoBolsa(new AcaoBolsa("VALEON_N1", 1782)));
17        broker.aceitarOrdem(
18            new VenderAcaoBolsa(new AcaoBolsa("BRADESPARPN_N1", 989)));
19
20        broker.realizaOrdensBolsa();
21    }
22 }

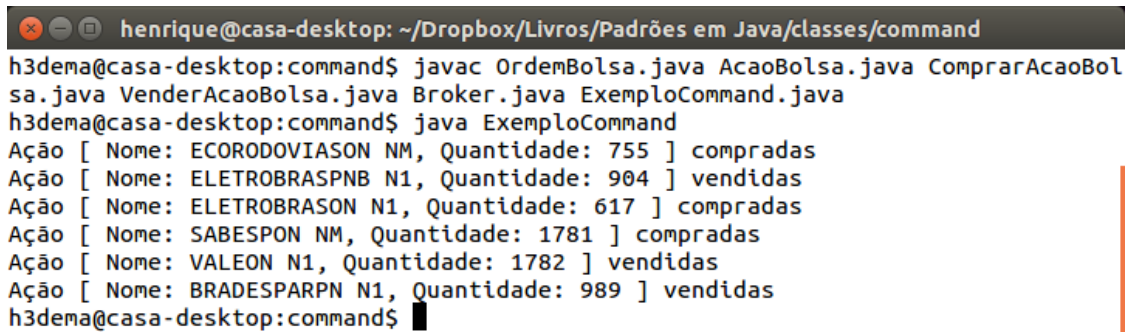
```

Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresentado na figura 12.2.

## 12.5 Conclusões

O padrão de projeto Command fornece uma interface para chamar um método para executar algum trabalho e encapsula todas as informações necessárias para fazê-lo em uma chamada de



A terminal window with a dark background and light text. The title bar shows the user 'henrique@casa-desktop' and the path '~/Dropbox/Livros/Padrões em Java/classes/command'. The terminal content shows the compilation and execution of Java code for the Command pattern. The output lists six actions: three purchases (compradas) and three sales (vendidas) with their respective item names and quantities.

```
henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/command
h3dema@casa-desktop:command$ javac OrdemBolsa.java AcaoBolsa.java ComprarAcaoBol
sa.java VenderAcaoBolsa.java Broker.java ExemploCommand.java
h3dema@casa-desktop:command$ java ExemploCommand
Ação [ Nome: ECORODOVIASON NM, Quantidade: 755 ] compradas
Ação [ Nome: ELETROBRASPNB N1, Quantidade: 904 ] vendidas
Ação [ Nome: ELETROBRASON N1, Quantidade: 617 ] compradas
Ação [ Nome: SABESPON NM, Quantidade: 1781 ] compradas
Ação [ Nome: VALEON N1, Quantidade: 1782 ] vendidas
Ação [ Nome: BRADESPARPN N1, Quantidade: 989 ] vendidas
h3dema@casa-desktop:command$
```

Figura 12.2: Execução de ExemploCommand.

objeto, que pode ser executado separadamente e mais tarde após instanciação. O padrão de projeto Command pode ser usada para conseguir operações de anulação se implementar um método que cancela o resultado executar a função. O padrão de desenho de comando pode ser usado para implementar a história do executado operações e macros como um conjunto de instâncias de comando, que pode ser executado em uma sequência.





## Capítulo 13

# Padrão: Strategy

O padrão **Strategy** (também conhecido como o padrão *policy*) é um *padrão de projeto* que permite que o comportamento de um algoritmo possa ser selecionado em tempo de execução.

Por exemplo, uma classe que executa a validação de dados de entrada pode utilizar um padrão **Strategy** para selecionar um algoritmo de validação com base no tipo de dados, a fonte de dados, a escolha feita pelo usuário ou outros fatores que discriminam cada dado. Esses fatores não são conhecidos para cada caso até o tempo de execução. Isto pode exigir que o processo de validação seja radicalmente diferente para cada entrada. As estratégias de validação, se encapsuladas separadamente do objeto validação, podem ser utilizadas por outros objetos de validação em diferentes áreas do sistema (ou mesmo sistemas diferentes) sem duplicação de código.

### 13.1 O padrão Strategy

O padrão Strategy é usado para gerenciar os algoritmos, relações e responsabilidades entre objetos.

O padrão **Strategy** incorpora dois princípios:

1. encapsular o conceito que varia e
2. fazer a programação baseada em uma interface, não uma implementação.

O padrão **Strategy** é fácil de implementar, sendo um dos *padrões de projeto* mais simples. Considere:

1. Implementar uma interface **Strategy** para seus objetos
2. Implementar classes *ConcreteStrategy* que implementam a interface **Strategy**
3. Em sua classe Context, manter uma referência particular a um objeto derivado de **Strategy**
4. Em sua classe Context, implementar métodos públicos do tipo setter e getter para o objeto derivado de **Strategy**

No diagrama da figura 13.1 a classe *Contexto* é composta de por uma *Estratégia*. O *Contexto* poderia ser qualquer coisa que exigiria mudar comportamentos, como por exemplo uma classe que fornece a funcionalidade de classificação. **Estratégia** é simplesmente implementada como uma interface, para que possamos trocar entre as **EstrategiasConcretas** sem afetar nosso contexto.

A interface **Strategy** define o comportamento de seus objetos de estratégia; por exemplo, a interface de estratégia para as Border é a interface `javax.swing.Border`. As classes **ConcreteStrategy** concretas implementar a interface *Estratégia*; por exemplo, para as `EmptyBorder`, `CompoundBorder`, `BevelBorder`, `SoftBevelBorder`, `EtchedBorder`, `LineBorder`, `TitledBorder`, `MatteBorder`, etc.

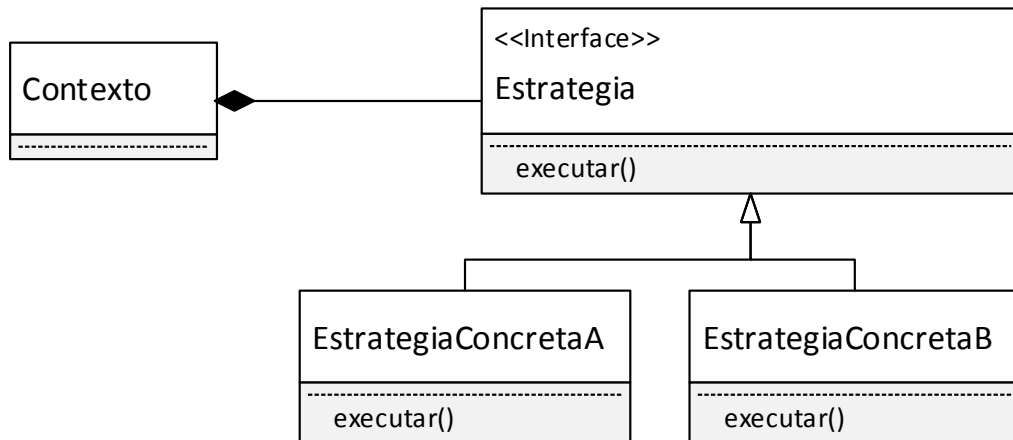


Figura 13.1: Classe que compõem o padrão **Strategy**.

A classe *JComponent* mantém uma referência particular a um objeto *Border*. Como *Border* é uma interface, e não uma classe, os componentes *Swing* podem ter qualquer classe de borda que implementa a interface de *Border*. É isto que significa programar para uma interface e não uma implementação.

## 13.2 Quando usar o padrão Strategy

A utilização deste padrão permite:

- definir uma família de algoritmos
- encapsular cada algoritmo, e
- fazer com que os algoritmos possam ser trocados dentro dessa família.

O padrão **Strategy** permite que o algoritmo varie independentemente dos clientes que o utilizam. Um requisito essencial para a linguagem de programação é a capacidade de armazenar uma referência a um código em uma estrutura de dados para depois recuperá-lo.

O padrão Strategy é para ser usado onde você quer escolher o algoritmo para usar em tempo de execução. Um bom uso do padrão Strategy seria salvar arquivos em formatos diferentes, correndo vários algoritmos de ordenação, ou compactação de arquivos.

O padrão Strategy fornece uma maneira para definir uma família de algoritmos, encapsula cada um como um objeto, e torná-los intercambiáveis.

## 13.3 Uma implementação

Listagem 13.1: Definindo a interface StrategyFormaCompressao.java

```

1 import java.util.ArrayList;
2 import java.io.File;
3 import java.io.OutputStream;
4 import java.io.IOException;
5
  
```



```

6  /** Strategy Interface */
7  public interface StrategyFormaCompressao {
8
9      public void executar(OutputStream out, ArrayList<File> files) throws
        IOException;
10 }

```

Listagem 13.2: Classe StrategyZip.java que faz compactação com zip

```

1  import java.util.ArrayList;
2  import java.io.File;
3  import java.io.OutputStream;
4  import java.util.zip.ZipOutputStream;
5  import java.util.zip.ZipEntry;
6  import java.nio.file.Files;
7  import java.io.IOException;
8
9  public class StrategyZip implements StrategyFormaCompressao {
10
11      // executa compressao com ZIP
12      @Override
13      public void executar(OutputStream out, ArrayList<File> files) throws
        IOException {
14          ZipOutputStream zip = new ZipOutputStream(out);
15
16          for(File f : files) {
17              byte[] arq = Files.readAllBytes(f.toPath());
18              int len = arq.length;
19              ZipEntry entry = new ZipEntry(f.getName());
20              zip.putNextEntry(entry);
21              zip.write(arq, 0, len);
22              zip.closeEntry();
23          }
24          zip.finish();
25          zip.close();
26      }
27 }

```

Listagem 13.3: Classe StrategyGzip.java que faz compactação com gzip

```

1  /**
2   * a classe GZIPOutputStream da API do Java não suporta multiplos arquivos
3   * por isto é necessário a utilização da classe TarArchiveOutputStream da
4   * Apache Commons
5   * que é capaz de agrupar todos os arquivos em um único OutputStream que por
6   * sua vez
7   * pode ser enviado para GZIPOutputStream
8   *
9   * mais informações em http://commons.apache.org/proper/commons-compress/
10  */
11 import java.util.ArrayList;
12 import java.io.File;
13 import java.io.OutputStream;
14 import java.nio.file.Files;
15 import java.io.IOException;
16
17 import java.util.zip.GZIPOutputStream;
18 import org.apache.commons.compress.archivers.tar.TarArchiveOutputStream;
19
20 public class StrategyGzip implements StrategyFormaCompressao {
21
22     // executa compressao com GZIP
23     @Override
24     public void executar(OutputStream out, ArrayList<File> files) throws
        IOException {

```



```

23     GZIPOutputStream zip = new GZIPOutputStream(out);
24     TarArchiveOutputStream tar = new TarArchiveOutputStream(zip);
25
26     for(File f : files) {
27         byte[] arq = Files.readAllBytes(f.toPath());
28         int len = arq.length;
29         tar.write(arq, 0, len);
30         tar.closeArchiveEntry();
31     }
32     tar.finish();
33     tar.close();
34 }
35 }

```

Listagem 13.4: Classe que utiliza as estratégias: Compressao.java

```

1  import java.util.ArrayList;
2  import java.io.File;
3  import java.io.FileOutputStream;
4  import java.io.BufferedOutputStream;
5
6  public class Compressao {
7      /** esta é a estratégia de compressão utilizada em tempo de execução */
8      private StrategyFormaCompressao compressao = null;
9      private String nomeArquivo;
10
11     public Compressao(String saida) {
12         this.nomeArquivo = saida;
13     }
14
15     public void setFormaCompressao(StrategyFormaCompressao compressao){
16         this.compressao = compressao;
17     }
18
19     /** chama a estratégia selecionada */
20     public void createArchive(ArrayList<File> files) {
21         if (null != compressao) {
22             try {
23                 BufferedOutputStream out = new BufferedOutputStream(new
24                     FileOutputStream(new File(nomeArquivo)));
25                 compressao.executar(out, files);
26             } catch (Exception e) {}
27         }
28     }
29 }

```

Listagem 13.5: Classe principal ClientCompressao.java

```

1  import java.util.ArrayList;
2  import java.io.File;
3
4  public class ClientCompressao{
5
6      public static void main(String[] args) {
7          // cria a lista de arquivos a serem compactados
8          ArrayList<File> arqs = new ArrayList<>();
9          arqs.add(new File("Compressao.java"));
10
11          Compressao z = new Compressao("arquivo.zip");
12          // define a forma de compactação
13          z.setFormaCompressao(new StrategyZip());
14          // compacta
15          z.createArchive(arqs);
16      }
17 }

```





Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresenta na figura 8.2. Note que no diretório vemos os arquivos *.class* gerados pela compilação das classes. Vemos ainda o *arquivo.zip* que é o arquivo compactado gerado pelo nosso exemplo.

```
henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/strategy
h3dema@casa-desktop:strategy$ ls
arquivo.zip      ExemploStrategy.java      StrategyZip.java
ClientCompressao.java  StrategyFormaCompressao.java
Compressao.java    StrategyGzip.java
h3dema@casa-desktop:strategy$ javac -cp ../../utils/ApacheCommonsCompress/commons-compress-1.9.jar StrategyFormaCompressao.java StrategyZip.java StrategyGzip.java
a Compressao.java ClientCompressao.java
h3dema@casa-desktop:strategy$ java -cp ../../utils/ApacheCommonsCompress/commons-compress-1.9.jar ClientCompressao
h3dema@casa-desktop:strategy$ ls
arquivo.zip      Compressao.java      StrategyGzip.class
ClientCompressao.class  ExemploStrategy.java  StrategyGzip.java
ClientCompressao.java  StrategyFormaCompressao.class  StrategyZip.class
Compressao.class      StrategyFormaCompressao.java  StrategyZip.java
h3dema@casa-desktop:strategy$
```

Figura 13.2: Execução de ClientCompressao.

Utilizamos o parâmetro *-cp* do java e do javac para poder definir onde está a biblioteca da Apache que utilizamos nestas classes.

## 13.4 Conclusões





## Capítulo 14

# Padrão: MVC

Muitas aplicações começam a partir de algo pequeno, como várias centenas ou poucos milhares de linhas de código, como um protótipo funcional de um aplicativo ou uma pequena aplicação. À medida que a aplicação evolui, os programadores adicionam novos recursos. O código de aplicação vai ficando mais complexo e torna-se muito mais difícil de entender como funciona e fica mais difícil modificá-lo, especialmente para um programador que não participou do projeto original. O **Model-View-Controller (MVC)** serve como base para uma arquitetura de software que é facilmente mantida e modificada.

### 14.1 Ideia principal do padrão MVC

A ideia principal do **MVC** é dividir um aplicativo em três partes: o modelo (**model**), a visão<sup>1</sup> (**view**), e o controlador (**controller**). O modelo compreende os dados e a lógica de negócios do aplicativo, a visão são as interfaces de usuário (a janela na tela e os relatórios), e o controlador é a aplicação que une estes dois. Enquanto a visão e controlador dependem do modelo, o modelo é independente da visão ou o controlador. Esta é uma característica fundamental da divisão. Este padrão permite que o programador trabalhe com o modelo, e, portanto, a lógica de negócios do aplicativo, independentemente da apresentação visual.

O diagrama da figura 14.1 mostra o fluxo de interação entre o usuário, o controlador, o modelo e a visão em um modelo **MVC**. Neste padrão, um usuário faz uma solicitação para o aplicativo e o controlador faz o processamento inicial. Depois que ele manipula o modelo para criar, atualizar ou excluir dados deste modelo. O modelo, então, retorna algum resultado para o controlador, que transmite o resultado para a visão, que processa os dados, gerando uma saída para o usuário.

Desta forma podemos resumir que:

Modelo - o conhecimento da aplicação
Visão - a aparência de conhecimento
Controlador - a cola entre o Modelo e Visão

#### 14.1.1 O Modelo

O modelo é uma pedra angular da aplicação, porque, enquanto a vista e controlador dependem do modelo, o modelo é independente da apresentação ou o controlador. O modelo fornece conhecimento de dados e como trabalhar com esses dados. O modelo tem um estado e métodos para alterar seu estado, mas não contém informações sobre como este conhecimento pode ser visualizado.

---

<sup>1</sup>Uma tradução melhor seria chamar **view** de apresentação, contudo perdemos a letra inicial. Por isto mantivemos uma tradução aproximada.

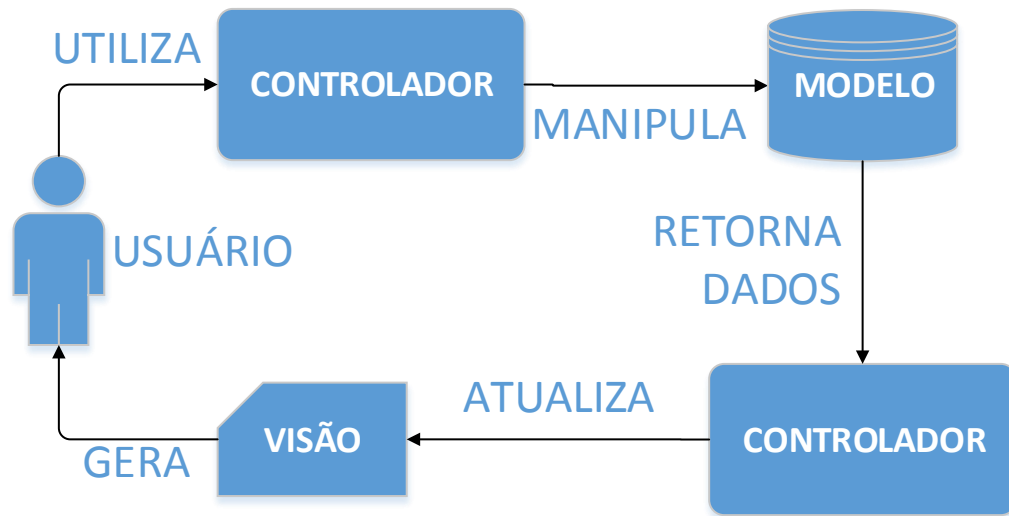


Figura 14.1: Interação entre as partes do padrão MVC

Esta independência torna o trabalho de forma independente, cobrindo o modelo com testes e substituindo os controladores / visões sem alterar a lógica de negócios de uma aplicação.

O modelo é responsável por manter a integridade de dados do programa. Devemos:

- Criar modelos de dados e interface de trabalho com eles
- Avaliar os dados e relatar todos os erros para o controlador
- Evite trabalhar diretamente com a interface do usuário

### 14.1.2 A Visão

A visão recebe os dados do modelo através do controlador. Cada visão é responsável pela visualização de uma parte dos dados. Uma visão não deve conter uma lógica complexa. O recomendável é que toda a lógica na visão deve ir para os modelos e controladores.

As interfaces com o usuário são sensíveis a mudanças, pois o usuário está sempre querendo mudar funcionalidades e a interface das aplicações. Se o sistema não suporta estas mudanças, temos um grande problema.

Se você precisar alterar o método de visualização, como por exemplo, se você precisa de seu aplicativo da Web para ser processado de maneira diferente dependendo se o usuário está usando um telefone celular ou navegador de desktop. Você pode alterar o modo de exibição de acordo com a necessidade. Basta ter duas visões, por exemplo. Devemos notar ainda que a mesma aplicação possui diferentes requisitos para a visão dependendo do usuário. Por exemplo, imagine o que um digitador e um gerente precisam para um sistema de controle de almoxarifado.

As principais recomendações são:

- Tente mantê-los simples
- utilizar apenas comparações simples e loops
- O uso de qualquer outra lógica que não loops e instruções condicionais (if-thenelse) porque a separação de interesses requer todo esse complexo lógica a ser realizados em modelos



- Evite acessando os dados diretamente

### 14.1.3 O Controlador

A responsabilidade direta dos controladores é receber dados a partir do pedido e enviá-lo a outras partes do sistema. Só que neste caso, o controlador é "fino" e destina-se apenas como um (camada de cola) ponte entre os componentes individuais o sistema.

Para trabalhar corretamente com controladores devemos:

- Dados passar de solicitações do usuário para o modelo para o processamento, recuperação e salvar os dados
- Dados passe para vistas para render
- Pega todos os erros de solicitação e erros de modelos
- Trabalhar com a lógica de banco de dados e negócios diretamente

Precisamos de modelos inteligentes, controladores finos e visões burras.
--

## 14.2 Vantagens do padrão MVC

O padrão **MVC** traz um muitas vantagens para sua aplicação. Algumas delas são:

- A decomposição permite dividir logicamente a aplicação em três partes relativamente independentes com acoplamento frouxo e irá diminuir a sua complexidade.
- Desenvolvedores tipicamente especializar em uma área, por exemplo, um desenvolvedor pode criar uma interface de usuário ou modificar a lógica de negócios. Assim, é possível limitar sua área de responsabilidade apenas alguma parte do código.
- **MVC** torna possível alterar a visualização, modificando assim a visão sem mudanças na lógica de negócios.
- **MVC** torna possível alterar a lógica de negócios, modificando assim o modelo sem mudanças na visualização.
- **MVC** torna possível alterar a resposta a uma ação do utilizador (clicar sobre o botão com o mouse, entrada de dados) sem alterar a aplicação de pontos de vista; é suficiente utilizar um controlador diferente.

## 14.3 Uma implementação

Para exemplificar o uso deste modelo vamos construir uma aplicação simples. Vamos fazer um sistema de votação. Neste sistema apresentamos ao usuário um conjunto de opções. Ele deverá escolher uma das opções e o sistema contabiliza a quantidade de votos obtidos por cada opção.

Para construir este sistema no modelo **MVC**, temos que pensar nas três camadas de forma separada. Começamos pelo modelo. Nesta parte pensamos a lógica do negócio (*business logic*). Recomendamos que mesmo para pequenas aplicações, você crie um pacote separado para armazenar as classes de modelo.

Começamos pela classe *Votacao*, mostrada na Listagem 14.1. Esta classe implementa os métodos que normalmente pensaríamos para guardar as opções e registrar os votos para cada opção. Contudo no padrão **MVC** precisamos que o modelo seja independente da apresentação. Desta forma temos que criar uma classe baseada em eventos que permita que as interfaces se



registrem para saber que existe alguma mudança. Isto pode ser feito utilizando o padrão Observer. Criamos portanto uma classe Enquete que é derivada da classe Votacao e que implementa *listeners*, como mostrado na figura 14.2.

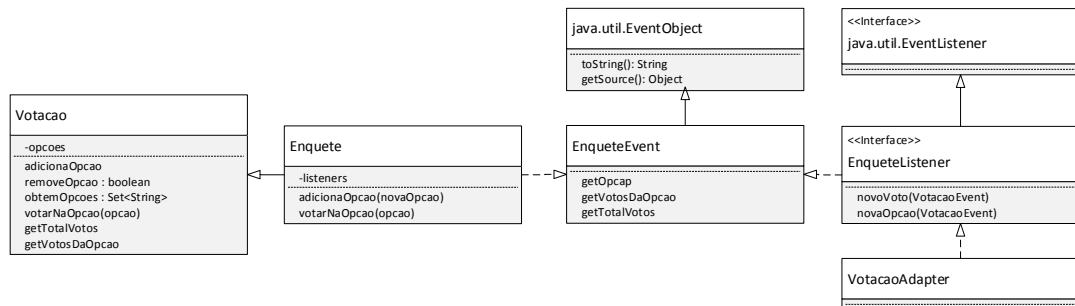


Figura 14.2: Modelo para Votacao implementando padrão Observer

Listagem 14.1: Votacao.java

```

1 package votacao.model;
2
3 import java.util.HashMap;
4 import java.util.Map;
5 import java.util.Set;
6
7 /**
8  * classe que retém as opções e seus votos
9  */
10 public class Votacao {
11
12     private Map<String,Integer> opcoes = new HashMap<String, Integer>();
13
14     /**
15      * Adiciona uma opção para ser votada
16      * @param novaOpcao nome da opção
17      */
18     public void adicionaOpcao(String novaOpcao){
19         opcoes.put(novaOpcao,new Integer(0));
20     }
21
22     /**adi
23      * Remove uma opção previamente cadastrada
24      * @param opcao nome da opção
25      * @returns true se conseguiu remover
26      */
27     public boolean removeOpcao(String opcao){
28         return opcoes.remove(opcao) != null; // retorna null se não achou
29     }
30
31     /**
32      * Retorna um iterator com as opções disponíveis
33      * @return Iterator opções disponíveis na enquete
34      */
35     public Set <String> obtemOpcoes(){
36         return opcoes.keySet();
37     }
38
39     /**
40      * Incrementa um voto para opçãoStringo
41      * @param opcao opção que receberá voto
  
```



```
42     */
43     public void votarNaOpcao(String opcao){
44         Integer qtd = (Integer)opcoes.get(opcao);
45         ++qtd;
46         opcoes.put(opcao,qtd);
47     }
48
49     /**
50      * Retorna a soma dos votos de todas as opções da enquete
51      * @return int soma dos votos de todas as opções da enquete
52      */
53     public int getTotalVotos(){
54
55         Integer votos = 0;
56         for(Integer qtd : opcoes.values()){
57             votos+= qtd;
58         }
59         return votos.intValue();
60     }
61
62     /**
63      * Retorna a quantidade de votos de uma opção individual
64      * @param opt opção que se quer o voto
65      * @return int quantidade de votos da opção
66      */
67     public int getVotosDaOpcao(String opcao){
68         return (opcoes.get(opcao)).intValue();
69     }
70 }
```

Listagem 14.2: Enquete.java

```
1 package votacao.model;
2
3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Enquete extends Votacao {
7
8     private List <EnqueteListener> listeners = new LinkedList<>();
9
10    /**
11     * Adiciona uma opção para ser votada
12     * @param novaOpcao nome da opção
13     */
14    public void adicionaOpcao(String novaOpcao){
15        super.adicionaOpcao(novaOpcao);
16        this.disparaNovaOpcao(novaOpcao);
17    }
18
19    /**
20     * Incrementa um voto para opção
21     * @param opcao opção que receberá voto
22     */
23    public void votarNaOpcao(String opcao){
24        super.votarNaOpcao(opcao);
25        this.disparaNovoVoto(opcao);
26    }
27
28    /**
29     * Adiciona um EnqueteListener, um objeto interessado em
30     * receber eventos lançados pela Enquete
31     * @see EnqueteListener
32     * @param listener objeto interessado em receber eventos
33     */
34    public synchronized void addListener(EnqueteListener listener){
```



```

35     if (!listeners.contains(listener)){
36         this.listeners.add(listener);
37     }
38 }
39
40 /**
41  * Informa aos objetos interessados nos eventos lançados
42  * pela Enquete que um novo voto foi contabilizado.
43  */
44 private synchronized void disparaNovoVoto(String opcao){
45     for(EnqueteListener l : this.listeners){
46         l.novoVoto(new EnqueteEvent(this,opcao));
47     }
48 }
49
50 /**
51  * Informa aos objetos interessados nos eventos lançados
52  * pela Enquete que uma nova opção foi adicionada.
53  */
54 private synchronized void disparaNovaOpcao(String opcao){
55     for(EnqueteListener l : this.listeners){
56         l.novaOpcao(new EnqueteEvent(this,opcao));
57     }
58 }
59
60 }

```

Listagem 14.3: EnqueteListener.java

```

1 package votacao.model;
2
3 import java.util.EventListener;
4
5 /**
6  * esta é a interface dos listeners
7  * assim garantimos que todos os eventos implementarão os dois métodos listados
8  * aqui
9  */
10 public interface EnqueteListener extends EventListener {
11
12     /**
13      * Este método é chamado quando um novo voto é adicionado.
14      * @param event Evento gerado pela Enquete.
15      */
16     public void novoVoto(EnqueteEvent event);
17
18     /**
19      * Este método é chamado quando uma nova opção é adicionada à Enquete.
20      * @param event Evento gerado pela Enquete.
21      */
22     public void novaOpcao(EnqueteEvent event);
23 }

```

Listagem 14.4: EnqueteEvent.java

```

1 package votacao.model;
2
3 import java.util.EventObject;
4
5 public class EnqueteEvent extends EventObject {
6
7     private String opcao = null;
8     private int votos = 0;
9
10    /**

```





```
11     * Construtor simples
12     */
13     public EnqueteEvent(Enquete e){
14         super(e);
15     }
16
17     public EnqueteEvent(Enquete e, String opcao){
18         this(e);
19         this.opcao = opcao;
20     }
21
22     /**
23      * Retorna a opção associada ao evento gerado.
24      * A opção pode ser uma nova opção adicionada à Enquete
25      * ou a opção escolhida para adicionar um novo voto.
26      * @return String opção
27      */
28     public String getOpcao() {
29         return opcao;
30     }
31
32     /**
33      * Retorna o numero de votos da opcao
34      * @return int votos
35      */
36     public int getVotosDaOpcao() {
37         return ((Enquete)this.source).getVotosDaOpcao(opcao);
38     }
39
40     /**
41      * Retorna o numero de votos da opcao
42      * @return int votos
43      */
44     public float getVotosDaOpcaoPercentual() {
45         Enquete e = (Enquete)this.source;
46         float total = e.getTotalVotos();
47         return total == 0 ? 0 : e.getVotosDaOpcao(opcao)/total;
48     }
49
50     /**
51      * Retorna o total de votos da enquete
52      * @return int
53      */
54     public int getTotalVotos() {
55         return ((Enquete)this.source).getTotalVotos();
56     }
57 }
58 }
```

Vamos criar agora as interfaces.

Listagem 14.5: TelaResultado.java

```
1 package votacao.view;
2
3 import java.awt.GridLayout;
4 import java.awt.Label;
5 import java.util.HashMap;
6 import java.util.Map;
7
8 import javax.swing.JFrame;
9
10 import votacao.model.EnqueteListener;
11 import votacao.model.EnqueteEvent;
12
13 public class TelaResultado extends JFrame implements EnqueteListener {
```



```

14
15  /** classe auxiliar para armazenar os 2 labels no HashMap*/
16  class LabelsResult {
17      private Label valor;
18      private Label percentual;
19
20      LabelsResult(Label valor, Label percentual) {
21          this.valor = valor;
22          this.percentual = percentual;
23      }
24
25      Label getValor() { return this.valor; }
26      Label getPercentual() { return this.percentual; }
27  }
28
29  private Map<String, LabelsResult> labels = new HashMap<>();
30
31  public TelaResultado() {
32      super("Tela de Resultados");
33      this.setSize(250,120);
34      this.setLayout(new GridLayout(0,3));
35      this.add(new Label("Apuração de Votos"));
36      this.add(new Label());
37      this.add(new Label());
38  }
39
40  /**
41   * @see enquete.model.EnqueteListener#novaOpcao(EnqueteEvent)
42   */
43  public void novaOpcao(EnqueteEvent event) {
44      String opcao = event.getOpcao();
45
46      if(!labels.containsKey(opcao)){
47          Label label = new Label(opcao+"-");
48          Label votos = new Label(""+event.getVotosDaOpcao());
49          Label percentual = new Label(""+event.getVotosDaOpcaoPercentual()+"%");
50          // apresenta na tela os labels
51          this.add(label);
52          this.add(votos);
53          this.add(percentual);
54          // armazena a referencia para atualizações
55          labels.put(opcao, new LabelsResult(votos, percentual));
56      }
57  }
58
59  /**
60   * @see enquete.model.EnqueteListener#novoVoto(EnqueteEvent)
61   */
62  public void novoVoto(EnqueteEvent event) {
63      String opcao = event.getOpcao();
64      LabelsResult c = labels.get(opcao);
65      Label votos = c.getValor();
66      votos.setText(""+event.getVotosDaOpcao());
67      atualizaPercentuais();
68  }
69
70
71  private void atualizaPercentuais() {
72      float total = 0;
73      for(LabelsResult c : labels.values()) {
74          total += Integer.valueOf(c.getValor().getText());
75      }
76      for(LabelsResult c : labels.values()) {
77          int valor = Integer.valueOf(c.getValor().getText());
78          float valorPerc = total == 0 ? 0 : valor*100/total;
79          Label percentual = c.getPercentual();

```



```

80     percentual.setText(""+valorPerc+"%");
81     }
82 }
83 }

```

Listagem 14.6: TelaVotacao.java

```

1  package votacao.view;
2
3  import java.awt.Label;
4  import java.awt.Button;
5  import java.awt.GridLayout;
6  import java.awt.event.ActionListener;
7  import java.awt.event.WindowAdapter;
8  import java.awt.event.WindowEvent;
9  import java.util.ArrayList;
10 import java.util.Collection;
11
12 import javax.swing.JFrame;
13
14 import votacao.model.EnqueteListener;
15 import votacao.model.EnqueteEvent;
16
17 public class TelaVotacao extends JFrame implements EnqueteListener{
18
19     private Collection<String> botoes = new ArrayList<>();
20
21     private ActionListener controller;
22
23     public TelaVotacao(ActionListener controller){
24         super("Tela de Votação");
25         this.controller = controller;
26         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27         setSize(300,120);
28         setLayout(new GridLayout(0,1)); // as opções serão colocadas uma abaixo
           da outra
29         add(new Label("Clique em uma das opções abaixo para votar:"));
30
31         addWindowListener(new WindowAdapter() {
32             public void windowClosing(WindowEvent e) {
33                 System.exit(0); // sai do sistema
34             }
35         });
36     }
37
38     /**
39      * @see enquete.model.EnqueteListener#novaOpcao(EnqueteEvent)
40      */
41     public void novaOpcao(EnqueteEvent event) {
42         String opcao = event.getOpcao();
43         Button botao;
44
45         if(!botoes.contains(opcao)){
46             botoes.add(opcao);
47             botao = new Button(opcao);
48             botao.setActionCommand(opcao);
49             botao.addActionListener(controller);
50             this.add(botao);
51         }
52     }
53
54     /**
55      * @see enquete.model.EnqueteListener#novoVoto(EnqueteEvent)
56      */
57     public void novoVoto(EnqueteEvent event) {
58         // não precisa fazer nada para nosso exemplo

```



```

59     }
60
61 }

```

Vamos fazer o controlador que une as telas com o modelo.

Listagem 14.7: TelaVotacaoCtrl.java

```

1  package votacao.control;
2
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5
6  import votacao.model.Enquete;
7
8  public class TelaVotacaoCtrl implements ActionListener {
9
10     private Enquete enquete;
11
12     public TelaVotacaoCtrl(Enquete enquete){
13         this.enquete = enquete;
14     }
15
16     /**
17      * Evento lançado pelo clique nos botoes da TelaVotacao
18      * @see java.awt.event.ActionListener#actionPerformed(ActionEvent)
19      */
20     public void actionPerformed(ActionEvent event) {
21         enquete.votarNaOpcao(event.getActionCommand());
22     }
23 }

```

Agora o programa que utiliza o padrão MVC.

Listagem 14.8: Votacao.java

```

1  package votacao;
2
3  import votacao.model.Enquete;
4  import votacao.control.TelaVotacaoCtrl;
5  import votacao.view.TelaVotacao;
6  import votacao.view.TelaResultado;
7
8  public class Votacao{
9
10     public static void main(String[] args) {
11
12         // Modelo
13         Enquete enquete= new Enquete();
14
15         // Controlador da Interface "TelaVotacao"
16         TelaVotacaoCtrl ctrl = new TelaVotacaoCtrl(enquete);
17
18         // Interface que altera o estado do modelo
19         TelaVotacao votacao = new TelaVotacao(ctrl);
20         votacao.setLocation(50,50);
21
22         // Interface que exibe o resultado da votacao
23         TelaResultado resultado = new TelaResultado();
24         resultado.setLocation(50,200);
25
26         // Adicionando as interfaces interessadas na mudança do
27         // estado do modelo
28         enquete.addListener(votacao);
29         enquete.addListener(resultado);
30     }

```

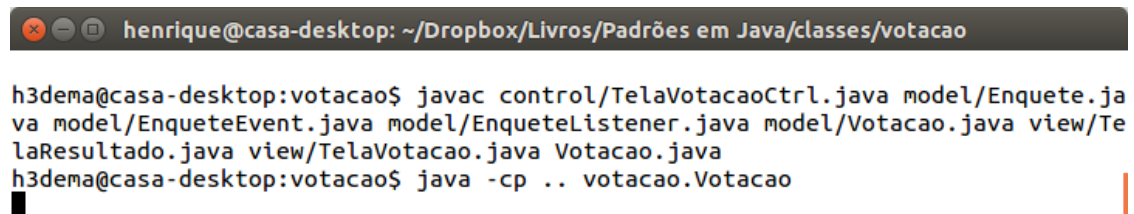


```

31 // Povoando o modelo
32 // note que as opções tem que ser preenchidas depois de criados Listeners,
33 // pois os eventos preenchem os valores nas telas
34 enquete.adicionaOpcao("Java_1.6");
35 enquete.adicionaOpcao("Java_1.7");
36 enquete.adicionaOpcao("Java_1.8");
37 enquete.adicionaOpcao("Java_1.9");
38
39 // Exibindo as interfaces
40 votacao.setVisible(true);
41 resultado.setVisible(true);
42 }
43
44 }

```

Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresenta na figura 14.3. Observem como compilamos as classes. Foi necessário passar o subdiretório para cada uma delas. Para rodar a classe principal também existe um truque. Vejam que estamos dentro do diretório *votacao*. A classe principal esta no pacote *votacao*. Portanto para rodar a classe principal, precisaríamos estar no diretório pai do diretório atual. Uma forma de fazer isto é mudar o classpath. Desta forma passamos a linha de comando do *java* o parâmetro “-cp ..”. Com isto o *java* irá procurar a partir do diretório pai e todas as classes serão achadas. O *java* utiliza a informação na diretiva *package* (primeira linha de cada classe que criamos neste exemplo) para montar os caminhos.



```

henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/votacao

h3dema@casa-desktop:votacao$ javac control/TelaVotacaoCtrl.java model/Enquete.java
model/EnqueteEvent.java model/EnqueteListener.java model/Votacao.java view/TelaResultado.java
view/TelaVotacao.java Votacao.java
h3dema@casa-desktop:votacao$ java -cp .. votacao.Votacao

```

Figura 14.3: Execução de Votacao.

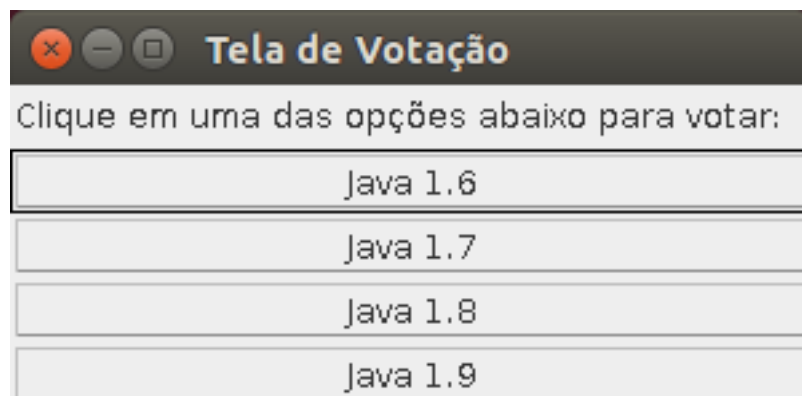


Figura 14.4: Tela de entrada de dados de Votacao.





A screenshot of a Java Swing window titled "Tela de Resultados". The window has a dark gray title bar with standard OS window controls (close, maximize, minimize). The content area is light gray and displays a table of election results. The table has three columns: the candidate name, the number of votes, and the percentage of the total. The data is as follows:

Apuração de		
Java 1.6 -	2	40.0%
Java 1.7 -	1	20.0%
Java 1.8 -	1	20.0%
Java 1.9 -	1	20.0%

Figura 14.5: Tela de Resultados de Votacao.

## Capítulo 15

# Padrão: Producer-Consumer

O *padrão de projeto* denominado Produtor-Consumidor (em inglês, **Producer-Consumer**) é uma solução padronizada que reduz o acoplamento entre Produtor e Consumidor, mediante a colocação de uma fila entre os produtores e os consumidores. Desta forma uma fila compartilhada é utilizada para controlar o fluxo. Esta separação permite codificar o produtor e o consumidor separadamente. Este padrão também aborda a questão de velocidades diferentes para produzir um item ou consumir um item. Usando este padrão, o Produtor e Consumidor podem trabalhar com velocidades diferentes.

Este padrão pode ser representando pelo diagrama mostrado na figura 15.1. A classe Produtor produz um Item e coloca este item na Fila. O Consumidor verifica a fila e consome um Item que está na Fila. Se a Fila estiver vazia o Consumidor espera. O Produtor não precisa saber quantos objetos da classe Consumidor existem, pois sua relação é somente com a Fila. Um ponto importante é que as operações de colocar um item na fila e remover um item da fila precisam ser atômicas. Um item é colocado na fila atômicamente, desta forma um consumidor não consegue obter este item parcialmente, pois ele só estará disponível quando estiver completamente registrado. Um item é retirado atômicamente, desta forma dois consumidores não podem obtê-lo ao mesmo tempo.

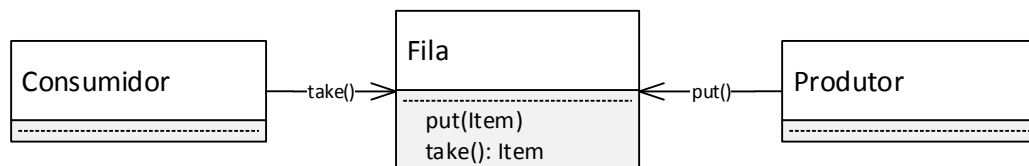


Figura 15.1: Execução de ExemploCommand.

### 15.1 Vantagens do padrão Producer-Consumer

A utilização deste padrão é bastante útil. Ele resolve um problema em computação denominado Condição de Corrida. Este padrão é comumente usado durante em código multithreaded. Podemos relacionar alguns de seus benefícios como:

- Este é um *padrão de projeto* simples. Nós podemos codificar as classes Produtor e Consumidor de forma independente. Os objetos desta classe podem rodar ao mesmo tempo. Estes objetos só precisam de saber qual o objeto compartilhado.

- O Produtor não precisa de saber quem é o Consumidor ou quantos consumidores existem. O mesmo acontece com o Consumidor.
- O Produtor e o Consumidor podem trabalhar com velocidades diferentes. Não há risco do Consumidor consumir um produto ainda não produzido inteiramente (por receber, receber somente uma parte de uma mensagem). Na verdade, podemos monitorar a velocidade do consumidor e com isto adequar o número de consumidores para uma melhor utilização dos recursos.
- Separar as funcionalidades de Produtor e de Consumidor resulta em código mais limpo, legível e gerenciável.

## 15.2 Uma implementação

Vamos mostrar um exemplo simples onde temos um produtor que gerar uma sequência de números inteiros e temos 2 consumidores que buscam estes números. A classe principal é apresentada na listagem 15.3. Note que o produtor e os consumidores são gerados como threads, assim não temos como saber quando irão rodar.

A implementação de fila que utilizamos foi feita com a classe *java.util.concurrent.BlockingQueue*. Existe um motivo de utilizarmos esta classe e não usarmos por exemplo *java.util.List*. Precisamos que as execuções de *put()* e *take()* sejam atômicas. Ou seja, ou todo o trabalho é feito, ou nada é feito. Isto garante a consistência dos dados, pois queremos que um *take()* só possa ser feito quando *put()* terminar. Assim um consumidor não consegue acessar uma informação incompleta. As implementações de *BlockingQueue* são *thread-safe*, enquanto em *List* não é.

A classe produtora é mostrada na listagem 15.1.

Listagem 15.1: Classe produtora

```
1 import java.util.concurrent.BlockingQueue;
2
3 import java.util.logging.Logger;
4 import java.util.logging.Level;
5
6 class Produtor implements Runnable {
7
8     private static int idcount = 0;
9
10    private int id;
11    private int num_produzir;
12    private final BlockingQueue<Integer> fila;
13
14    public Produtor(BlockingQueue<Integer> fila, int num_produzir) {
15        this.fila = fila;
16        this.id = idcount++;
17        this.num_produzir = num_produzir;
18    }
19
20    @Override
21    public void run() {
22        for(int i=0; i < num_produzir; i++){
23            try {
24                System.out.println("Produtor_" + id + "_" + "Produziu item#" + i);
25                fila.put(i);
26            } catch (InterruptedException ex) {
27                Logger.getLogger(Produtor.class.getName()).log(Level.SEVERE, null, ex);
28            }
29        }
30    }
31 }
```





Listagem 15.2: Classe consumidora

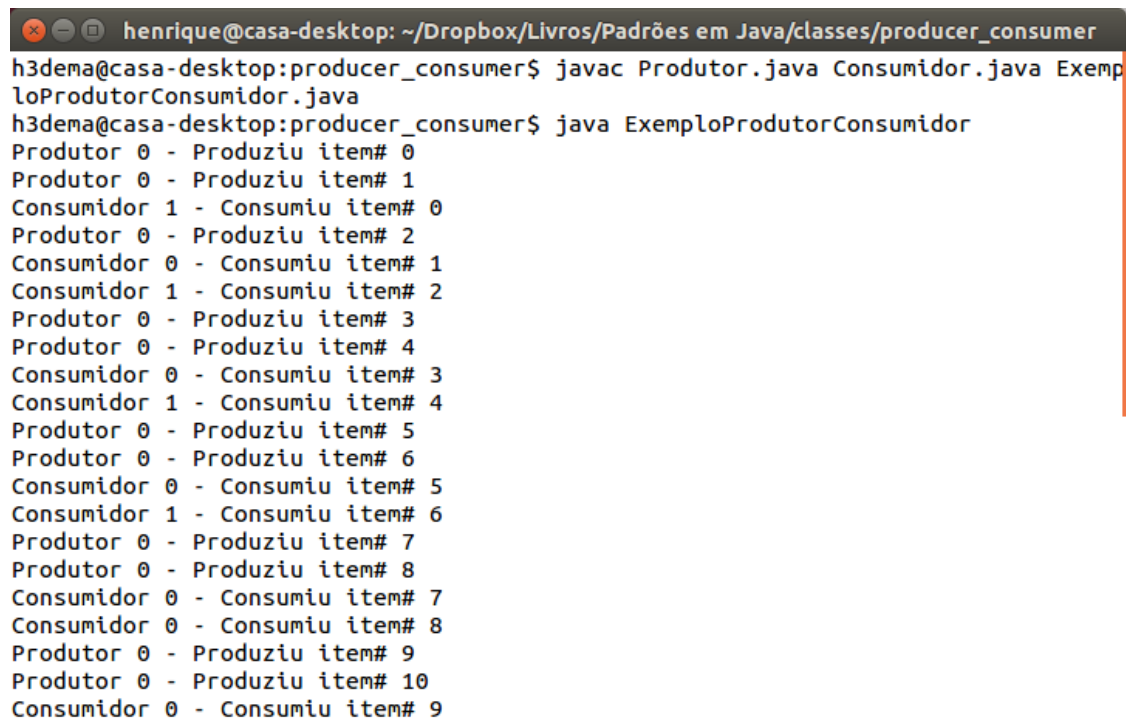
```
1 import java.util.concurrent.BlockingQueue;
2
3 import java.util.logging.Logger;
4 import java.util.logging.Level;
5
6 class Consumidor implements Runnable {
7
8     private static int idcount = 0;
9
10    private int id;
11    private final BlockingQueue<Integer> fila;
12
13    public Consumidor(BlockingQueue<Integer> fila) {
14        this.fila = fila;
15        this.id = idcount++;
16    }
17
18    @Override
19    public void run() {
20        while(true){
21            try {
22                System.out.println("Consumidor_" + id + " - Consumiu item#" + fila.take());
23            } catch (InterruptedException ex) {
24                Logger.getLogger(Consumidor.class.getName()).log(Level.SEVERE, null, ex);
25            }
26        }
27    }
28
29 }
30 }
```

Listagem 15.3: Classe principal ExemploProdutorConsumidor

```
1 /**
2  * exemplo de padrão Producer-Consumer
3  * usando BlockingQueue da API do Java
4  *
5  */
6 import java.util.concurrent.BlockingQueue;
7 import java.util.concurrent.LinkedBlockingQueue;
8
9 import java.util.logging.Level;
10 import java.util.logging.Logger;
11
12 public class ExemploProdutorConsumidor {
13
14     public static void main(String args[]){
15
16         // cria a fila compartilhada, que separa produtor de consumidor
17         BlockingQueue<Integer> fila = new LinkedBlockingQueue<>();
18
19         // criando thread para o produtor e consumidor
20         Thread produtor = new Thread(new Produtor(fila, 20));
21         Thread consumidor1 = new Thread(new Consumidor(fila));
22         Thread consumidor2 = new Thread(new Consumidor(fila));
23
24         //rodando
25         produtor.start();
26         consumidor1.start();
27         consumidor2.start();
28     }
29 }
30 }
```



Podemos compilar as classes e, depois, rodar a classe principal. Obtemos o resultado apresentado na figura 15.2.



```
henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/producer_consumer
h3dema@casa-desktop:producer_consumer$ javac Produtor.java Consumidor.java ExemploProdutorConsumidor.java
h3dema@casa-desktop:producer_consumer$ java ExemploProdutorConsumidor
Produtor 0 - Produziu item# 0
Produtor 0 - Produziu item# 1
Consumidor 1 - Consumiu item# 0
Produtor 0 - Produziu item# 2
Consumidor 0 - Consumiu item# 1
Consumidor 1 - Consumiu item# 2
Produtor 0 - Produziu item# 3
Produtor 0 - Produziu item# 4
Consumidor 0 - Consumiu item# 3
Consumidor 1 - Consumiu item# 4
Produtor 0 - Produziu item# 5
Produtor 0 - Produziu item# 6
Consumidor 0 - Consumiu item# 5
Consumidor 1 - Consumiu item# 6
Produtor 0 - Produziu item# 7
Produtor 0 - Produziu item# 8
Consumidor 0 - Consumiu item# 7
Consumidor 0 - Consumiu item# 8
Produtor 0 - Produziu item# 9
Produtor 0 - Produziu item# 10
Consumidor 0 - Consumiu item# 9
```

Figura 15.2: Execução de ExemploProdutorConsumidor.

## Capítulo 16

# Padrão: Object Pool

O padrão **Object Pool** pode oferecer um ganho significativo de desempenho. Este padrão é mais eficaz em situações em que o custo de inicializar uma instância da classe é elevado, a taxa de instanciação de uma classe é alta e o número de instâncias em utilização em qualquer momento é baixo.

### 16.1 O padrão

Os **Object Pools** (também conhecido como pools de recursos) são usados para gerenciar uma cache de objetos. Um cliente com acesso ao **Object Pool** pode evitar a criação de novos objetos. O cliente solicita ao **Object Pool** objetos que já foram instanciados uma vez. Desta forma vemos que este é um padrão criacional.

Um **Object Pool** poderá ser um pool crescente, ou seja, o pool irá criar os novos objetos se o pool estiver vazio. O pool pode restringir o número de objetos criados, isto é, existe um limite máximo para o número de objetos a serem criados. O pool pode ainda receber objetos criados fora do pool, somente gerenciando a disponibilização destes objetos para os clientes, mediante as instruções de “check in” e “check out”. O **Object Pool** permite que os clientes façam “check-out” dos objetos de gerencia. Quando esses objetos não são mais necessários pelos processos clientes, estes objetos são retornados para o pool, a fim de serem reutilizados.

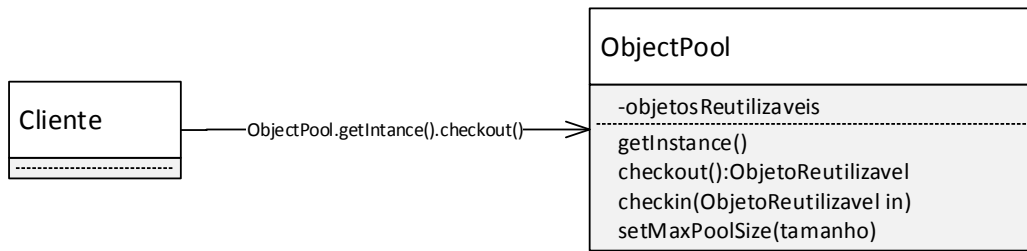
É desejável manter todos os objetos reutilizáveis, que não estão atualmente em uso, no mesmo pool de recursos de modo que eles podem ser gerenciados por uma política coerente. Para conseguir isso, a classe reutilizável **Object Pool** é projetada para ser uma classe **singleton**.

**Object Pools** empregam algumas estratégias para lidar com um pedido quando não há objetos disponíveis:

1. Um **Object Pool** poderá não fornecer o objeto solicitado e retornar um erro para o cliente.
2. Um **Object Pool** poderá atribuir um novo objeto, aumentando o tamanho do pool. Normalmente este tipo de implementação permitem que seja definido um limite máximo.
3. Em um ambiente multithread, Um **Object Pool** pode bloquear o cliente até que outro processo retorne um objeto.

Na figura 16.1 mostramos o diagrama de classes para um **Object Pool**. A ideia básica é que a classe mantenha instâncias de uma classe (que chamamos ObjetoReutilizavel) que possam ser utilizados e reutilizados.

Ao escrever um **Object Pool**, devemos ter cuidado para garantir que o estado dos objetos retornados para o pool é repostos volta a um estado apropriado para o próximo uso do objeto. Se isso não for observado, o objeto poderá estar em um estado inesperado pelo programa cliente.

Figura 16.1: Exemplo de diagrama de classe para **Object Pool**.

Isto pode fazer com que o programa cliente falhe. Cabe ao **Object Pool** redefinir os objetos, não os clientes.

Se o pool é utilizado por vários processos (ou *threads*), ele pode precisar evitar que os processos ou threads paralelos obtenham e tentem utilizar o mesmo objeto em paralelo. Com isto os objetos no pool podem ser imutáveis ou as operações sobre a coleção de objetos deve ser *thread-safe*.

## 16.2 Uma implementação

Vamos fazer uma estrutura de classe capaz de permitir a reserva de salas e equipamentos. A totalidade das salas e equipamentos é mantida por um pool especializado. Note que neste exemplo, não estamos fazendo os pools serem **singletons**. Podemos manter uma relação com os objetos que fizeram checkout, utilizando uma lista de objetos que fizeram “check out”. Desta forma as operações de “check in” e “check out” faríamos a movimentação de uma lista para outra.

Listagem 16.1: Interface para o object pool

```

1 public interface ResourcePool<T> {
2
3     /** permite obter um item do pool */
4     T checkout();
5
6     /** permite obter o item do pool, se ele estiver disponível */
7     T checkout(T v);
8
9     /** verifica se o recurso esta disponível */
10    public boolean disponivel(T v);
11
12    /** Libera o pool */
13    void checkin(T v);
14
15 }
  
```

Listagem 16.2: Implementação genérica para o pool de objetos

```

1 import java.util.List;
2 import java.util.ArrayList;
3
4 class Pool<T> implements ResourcePool<T> {
5
6     /** armazena o conjunto de item que fazem parte do pool */
7     protected List<T> lista = new ArrayList<>();
8     /** lista dos itens que fizeram checkout */
9     private List<T> checkedout = new ArrayList<>();
10
  
```



```
11
12 // inicializa a lista de item
13 public Pool(List<T> lista) {
14     if (null != lista)
15         this.lista.addAll(lista);
16 }
17
18 // escolhe um item da coleção
19 @Override
20 public synchronized T checkout() {
21     if (!lista.isEmpty()) {
22         T v = lista.remove(0); // retira o primeiro da fila
23         checkedout.add(v);     // move para a lista de "checked out"
24         return v;
25     } else {
26         return null;
27     }
28 }
29
30 // verifica se o item esta disponível
31 public boolean disponivel(T v) {
32     return lista.contains(v) && lista.remove(v);
33 }
34
35 // escolhe um item da coleção, se estiver disponível
36 @Override
37 public synchronized T checkout(T v) {
38     if (disponivel(v)) {
39         lista.remove(v); // remove da lista de disponíveis
40         checkedout.add(v); // move para a lista de "checked out"
41         return v;
42     } else {
43         return null;
44     }
45 }
46
47 // adiciona o item na coleção
48 public synchronized void checkin(T v) {
49     checkedout.remove(v); // remove da lista de "checked out"
50     this.lista.add(v);    // coloca como disponível
51 }
52
53 public String toString() {
54     String resultado = "";
55     for (T v : lista)
56         resultado += v + "\n";
57     return resultado;
58 }
59
60 }
```

Listagem 16.3: Implementação para pool de equipamentos

```
1 import java.util.List;
2
3 public class PoolEquipamento extends Pool<Equipamento> {
4
5     // inicializa a lista de funcionários
6     public PoolEquipamento(List<Equipamento> equipamentos) {
7         super(equipamentos);
8     }
9
10    // escolhe uma sala da coleção baseado no tipo, se estiver disponível
11    public Equipamento checkout(TipoEquipamento t) {
12        for(Equipamento e : lista) {
13            if (e.getTipo() == t.toString()) {
```



```

14         return checkout(e);
15     }
16 }
17 return null;
18 }
19 }

```

Listagem 16.4: Classe que implementa os equipamentos

```

1 public class Equipamento {
2
3     private int id;
4     private String nome;
5     private TipoEquipamento tipo;
6
7     public Equipamento(int id, String nome, TipoEquipamento tipo) {
8         this.id = id;
9         this.nome = nome;
10        this.tipo = tipo;
11    }
12
13    public Equipamento(int id, String nome) {
14        this(id, nome, TipoEquipamento.OUTRO);
15    }
16
17    public int getID() { return id; }
18    public String getNome() { return nome; }
19    public String getTipo() { return tipo.toString(); }
20
21    public String toString() { return String.format("[%02d] □ %s", id, nome); }
22
23 }

```

Listagem 16.5: Enumeração com os tipos de equipamentos

```

1 enum TipoEquipamento {COMPUTADOR, PROJETO, OUTRO}

```

Listagem 16.6: Implementação para pool de salas

```

1 import java.util.List;
2 import java.util.ArrayList;
3
4 public class PoolSala extends Pool<Sala> {
5
6     // inicializa a lista de salas
7     public PoolSala(List<Sala> salas) {
8         super(salas);
9     }
10
11    // escolhe uma sala da coleção baseado no tipo, se estiver disponível
12    public Sala checkout(TipoSala t) {
13        for(Sala s : lista) {
14            if (s.getTipo() == t.toString()) {
15                return checkout(s);
16            }
17        }
18        return null;
19    }
20 }

```

Listagem 16.7: Classe que implementa as salas

```

1 public class Sala {
2

```



```
3     private static int id_count = 0;
4
5     private int id;
6     private String nome;
7     private String local;
8     private TipoSala tipo;
9
10    public Sala(String nome, String local, TipoSala tipo) {
11        this.id = id_count++;
12        this.nome = nome;
13        this.local = local;
14        this.tipo = tipo;
15    }
16
17    public Sala(String nome, TipoSala tipo) {
18        this(nome, "INDEFINIDO", tipo);
19    }
20
21    public int getID() { return id; }
22    public String getNome() { return nome; }
23    public String getLocal() { return local; }
24    public String getTipo() { return tipo.toString(); }
25
26    public String toString() { return String.format("[%02d]_Sala_n %s",id,nome);
27    }
```

Listagem 16.8: Enumeração com os tipos de salas

```
1 enum TipoSala { GRANDE, PEQUENA, AUDITORIO }
```

Listagem 16.9: Classe principal de teste

```
1 import java.util.Arrays;
2 import java.util.ArrayList;
3
4 class TesteObjectPool {
5
6     public static void main(String[] args) {
7         /** cria o conjunto de equipamentos */
8         PoolEquipamento equipments = new PoolEquipamento(
9             new ArrayList<Equipamento>(Arrays.asList(
10                 new Equipamento(100, "Projektor_Sony_HDMI", TipoEquipamento.PROJETOR),
11                 new Equipamento(101, "Projektor_Sony_Wi-Fi", TipoEquipamento.PROJETOR),
12                 ,
13                 new Equipamento(102, "Projektor_Sony_WXGA", TipoEquipamento.PROJETOR),
14                 new Equipamento(103, "Projektor_Panasonic", TipoEquipamento.PROJETOR),
15                 new Equipamento(200, "Notebook_Asus_Zenbook", TipoEquipamento.COMPUTADOR),
16                 new Equipamento(201, "Notebook_Asus_Taichi_21", TipoEquipamento.COMPUTADOR),
17                 new Equipamento(300, "Ponteira_laser")
18             )))
19         /** */
20         PoolSala salas = new PoolSala(
21             new ArrayList<Sala>(Arrays.asList(
22                 new Sala("101", TipoSala.AUDITORIO),
23                 new Sala("201", TipoSala.PEQUENA),
24                 new Sala("202", TipoSala.PEQUENA),
25                 new Sala("305", TipoSala.PEQUENA)
26             ))
27         );
28
29         /**
```

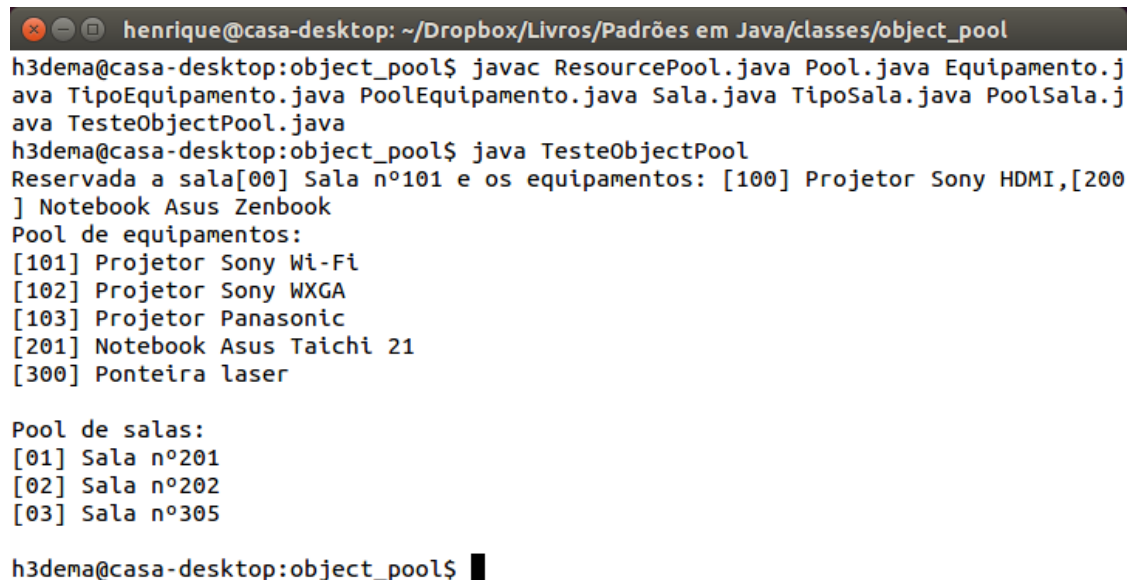


```

30      * faz uma reserva
31      */
32      Sala s1 = salas.checkout();
33      Equipamento e1 = equips.checkout(TipoEquipamento.PROJETOR);
34      Equipamento e2 = equips.checkout(TipoEquipamento.COMPUTADOR);
35      System.out.println("Reservada a sala "+s1+" e os equipamentos: "+e1+", "+e2);
36
37      System.out.println("Pool de equipamentos:");
38      System.out.println(equips);
39      System.out.println("Pool de salas:");
40      System.out.println(salas);
41  }
42
43  }

```

Podemos compilar as classes e rodar a classe principal, obtendo o resultado apresentado na figura 16.2. Criamos os pools nas linhas 8 a 26 da classe principal. Logo depois solicitamos uma sala e equipamentos do pool de equipamentos. Ao final da execução mostramos os pools para conferir se os objetos que fizeram “check out” foram removidos dos pools.



```

henrique@casa-desktop: ~/Dropbox/Livros/Padrões em Java/classes/object_pool
h3dema@casa-desktop:object_pool$ javac ResourcePool.java Pool.java Equipamento.j
ava TipoEquipamento.java PoolEquipamento.java Sala.java TipoSala.java PoolSala.j
ava TesteObjectPool.java
h3dema@casa-desktop:object_pool$ java TesteObjectPool
Reservada a sala[00] Sala nº101 e os equipamentos: [100] Projetor Sony HDMI,[200
] Notebook Asus Zenbook
Pool de equipamentos:
[101] Projetor Sony Wi-Fi
[102] Projetor Sony WXGA
[103] Projetor Panasonic
[201] Notebook Asus Taichi 21
[300] Ponteira laser

Pool de salas:
[01] Sala nº201
[02] Sala nº202
[03] Sala nº305

h3dema@casa-desktop:object_pool$

```

Figura 16.2: Execução de TesteObjectPool.



## Capítulo 17

# Apendice A - Cripto

Apresentamos na listagem 17.1 um código básico para criptografia de strings. A classe possui dois métodos estáticos que podem ser chamados para criptografar ou descriptografar uma string. O resultado é retornado também como uma string. As chamadas são simples.

Para codificar uma string *msg* devemos chamar *Cripto.encripta(chave1, chave2, msg)*, onde *chave1* e *chave2* são duas strings utilizadas para gerar os códigos de criptografia usado *AES*<sup>1</sup>. *chave1* é utilizada para gerar a chave de criptografia (*javax.crypto.spec.SecretKeySpec*). A *chave2* é utilizada para gerar o vetor de inicialização (*javax.crypto.spec.IvParameterSpec*) necessária para este tipo de criptografia. Para descriptografar o texto gerado por *encript* devemos efetuar uma chamada para *Cripto.decripta(chave1, chave2, msg\_encriptada)*.

Listagem 17.1: Cripto.java

```
1  /**
2   * download disponivel em http://commons.apache.org/proper/commons-codec/
3   * download_codec.cgi
4   */
5   import org.apache.commons.codec.binary.Base64;
6
7   import javax.crypto.spec.IvParameterSpec;
8   import javax.crypto.spec.SecretKeySpec;
9   import javax.crypto.Cipher;
10
11  public class Cripto {
12
13      /**
14       * @params chave1, chave2 = chaves de 128 bit, devem ter 16 caracteres
15       * chave1 gera a chave de criptografia
16       * chave2 inicia o vetor
17       */
18
19      private static final String ALGORITMO = "AES";
20      private static final String TRANSFORMACAO = "AES/CBC/PKCS5PADDING";
21      private static final String CODIF_STRING = "UTF-8";
22
23      public static String encripta(String chave1, String chave2, String
24          texto_normal) {
25          try {
26              SecretKeySpec chaveSeg = new SecretKeySpec(chave1.getBytes(CODIF_STRING),
27                  ALGORITMO);
28              IvParameterSpec iv = new IvParameterSpec(chave2.getBytes(CODIF_STRING));
29
30              Cipher cipher = Cipher.getInstance(TRANSFORMACAO);
```

<sup>1</sup>Dê uma olhada em [https://pt.wikipedia.org/wiki/Advanced\\_Encryption\\_Standard](https://pt.wikipedia.org/wiki/Advanced_Encryption_Standard)

```

29         cipher.init(Cipher.ENCRYPT_MODE, chaveSeg, iv); // <<< define para
           encriptar
30
31         byte[] texto_encriptado = cipher.doFinal(texto_normal.getBytes());
32         //System.out.println("texto encriptado:" + Base64.encodeBase64String(
           texto_encriptado));
33         return Base64.encodeBase64String(texto_encriptado);
34     } catch (Exception ex) {
35         ex.printStackTrace();
36     }
37     return null;
38 }
39
40 public static String decripta(String chave1, String chave2, String
           texto_encriptado) {
41     try {
42         SecretKeySpec chaveSeg = new SecretKeySpec(chave1.getBytes(CODIF_STRING),
           ALGORITMO);
43         IvParameterSpec iv = new IvParameterSpec(chave2.getBytes(CODIF_STRING));
44
45         Cipher cipher = Cipher.getInstance(TRANSFORMACAO);
46         cipher.init(Cipher.DECRYPT_MODE, chaveSeg, iv); // <<< define para
           decriptar
47
48         byte[] texto_normal = cipher.doFinal(Base64.decodeBase64(texto_encriptado
           ));
49
50         return new String(texto_normal);
51     } catch (Exception ex) {
52         ex.printStackTrace();
53     }
54     return null;
55 }
56 }

```

Os dois procedimentos são semelhantes. Para criptografar, criamos a chave de segurança usando `javax.crypto.SecretKeySpec`, informando a `chave1` e o algoritmo desejado. No nosso exemplo este é o AES. Criamos o vetor de inicialização chamando o método `javax.crypto.IvParameterSpec` com a `chave2`. Tendo estes dois valores, podemos criar o objeto que efetua a encriptação ou decriptação dos textos. Usamos a classe `javax.crypto.Cipher`, informando o tipo de criptografia desejada. O parâmetro `TRANSFORMACAO` indica a transformação que a instância deverá fazer no formato "algorithm/mode/padding".

Deste ponto em diante, os dois procedimentos divergem. O procedimento de encriptação chama `cipher.init` com o parâmetro `Cipher.ENCRYPT_MODE` que indica que a instância irá realizar a criptografia de texto. Para criptografar uma string chamamos o método `doFinal` a mensagem que desejamos encriptar com um array de bytes. O resultado é também um array de bytes, por isto precisamos da classe `Base64` para codificar este array como uma string. Desta forma, nosso método `encriptar` retorna uma string cujos caracteres representam o texto codificado.

Já no procedimento de decriptação chamamos `cipher.init` com o parâmetro `Cipher.DECRYPT_MODE` que indica que a instância irá realizar a decriptação de um texto. Para decriptografar chamamos o método `doFinal`. Note que o parâmetro do método `decripta` é uma string codificada com BASE64. Desta forma passamos para `doFinal` um array de bytes, gerado pela chamada de `Base64.decodeBase64`. O resultado é também um array de bytes, por isto precisamos novamente da classe `Base64` para codificar este array como uma string. Esta string é o nosso texto decriptado.

Ambos os procedimentos podem ser melhorados verificando no início se as chaves tem tamanho igual a 16 caracteres. Tamanhos maiores que 16 caracteres geram erro, que não está sendo tratado nesta classe simples.

Para que a classe `Cripto` compile é necessário o download do pacote **Apache Commons Co-**



**dec.** Este pacote pode ser localizado em [http://commons.apache.org/proper/commons-codec/download\\_codec.cgi](http://commons.apache.org/proper/commons-codec/download_codec.cgi). Precisamos deste pacote da Apache pois utilizamos a classe `org.apache.commons.codec.binary.Base64`, que fornece codificação e decodificação Base64, como definido pela RFC 2045.

