



K9

TREINAMENTOS

Design Patterns em Java

Design Patterns em Java

14 de junho de 2015

As apostilas atualizadas estão disponíveis em www.k19.com.br

| | |
|---|----------|
| Sumário | i |
| Sobre a K19 | 1 |
| Seguro Treinamento | 2 |
| Termo de Uso | 3 |
| Cursos | 4 |
| 1 Introdução | 5 |
| 1.1 Sistemas Corporativos | 5 |
| 1.2 Orientação a Objetos | 5 |
| 1.3 Padrões de Projeto | 5 |
| 2 Padrões de criação | 7 |
| 2.1 Factory Method | 7 |
| 2.2 Exercícios de Fixação | 11 |
| 2.3 Abstract Factory | 13 |
| 2.4 Abstract Factory + Factory Method | 16 |
| 2.5 Exercícios de Fixação | 17 |
| 2.6 Exercícios Complementares | 20 |
| 2.7 Builder | 20 |
| 2.8 Exercícios de Fixação | 23 |
| 2.9 Prototype | 26 |
| 2.10 Exercícios de Fixação | 28 |
| 2.11 Singleton | 30 |
| 2.12 Exercícios de Fixação | 31 |
| 2.13 Multiton (não GoF) | 32 |
| 2.14 Exercícios de Fixação | 33 |
| 2.15 Object Pool (não GoF) | 35 |
| 2.16 Exercícios de Fixação | 38 |

| | |
|---|-----------|
| 3 Padrões Estruturais | 41 |
| 3.1 Adapter | 41 |
| 3.2 Exercícios de Fixação | 44 |
| 3.3 Bridge | 46 |
| 3.4 Exercícios de Fixação | 49 |
| 3.5 Composite | 51 |
| 3.6 Exercícios de Fixação | 55 |
| 3.7 Decorator | 57 |
| 3.8 Exercícios de Fixação | 61 |
| 3.9 Facade | 63 |
| 3.10 Exercícios de Fixação | 66 |
| 3.11 Front Controller (não GoF) | 68 |
| 3.12 Exercícios de Fixação | 69 |
| 3.13 Flyweight | 70 |
| 3.14 Exercícios de Fixação | 73 |
| 3.15 Proxy | 75 |
| 3.16 Exercícios de Fixação | 78 |
| 4 Padrões Comportamentais | 81 |
| 4.1 Command | 81 |
| 4.2 Exercícios de Fixação | 84 |
| 4.3 Iterator | 86 |
| 4.4 Exercícios de Fixação | 89 |
| 4.5 Mediator | 90 |
| 4.6 Exercícios de Fixação | 93 |
| 4.7 Observer | 96 |
| 4.8 Exercícios de Fixação | 99 |
| 4.9 State | 101 |
| 4.10 Exercícios de Fixação | 103 |
| 4.11 Strategy | 105 |
| 4.12 Exercícios de Fixação | 108 |
| 4.13 Template Method | 109 |
| 4.14 Exercícios de Fixação | 112 |
| 4.15 Visitor | 113 |
| 4.16 Exercícios de Fixação | 117 |



Sobre a K19

A K19 é uma empresa especializada na capacitação de desenvolvedores de software. Sua equipe é composta por profissionais formados em Ciência da Computação pela Universidade de São Paulo (USP) e que possuem vasta experiência em treinamento de profissionais para área de TI.

O principal objetivo da K19 é oferecer treinamentos de máxima qualidade e relacionados às principais tecnologias utilizadas pelas empresas. Através desses treinamentos, seus alunos tornam-se capacitados para atuar no mercado de trabalho.

Visando a máxima qualidade, a K19 mantém as suas apostilas em constante renovação e melhoria, oferece instalações físicas apropriadas para o ensino e seus instrutores estão sempre atualizados didática e tecnicamente.



Seguro Treinamento

Na K19 o aluno faz o curso quantas vezes quiser!

Comprometida com o aprendizado e com a satisfação dos seus alunos, a K19 é a única que possui o Seguro Treinamento. Ao contratar um curso, o aluno poderá refazê-lo quantas vezes desejar mediante a disponibilidade de vagas e pagamento da franquia do Seguro Treinamento.

As vagas não preenchidas até um dia antes do início de uma turma da K19 serão destinadas ao alunos que desejam utilizar o Seguro Treinamento. O valor da franquia para utilizar o Seguro Treinamento é 10% do valor total do curso.



Termo de Uso

Termo de Uso

Todo o conteúdo desta apostila é propriedade da K19 Treinamentos. A apostila pode ser utilizada livremente para estudo pessoal . Além disso, este material didático pode ser utilizado como material de apoio em cursos de ensino superior desde que a instituição correspondente seja reconhecida pelo MEC (Ministério da Educação) e que a K19 seja citada explicitamente como proprietária do material.

É proibida qualquer utilização desse material que não se enquadre nas condições acima sem o prévio consentimento formal, por escrito, da K19 Treinamentos. O uso indevido está sujeito às medidas legais cabíveis.



Conheça os nossos cursos

-  K01 - Lógica de Programação
-  K02 - Desenvolvimento Web com HTML, CSS e JavaScript
-  K03 - SQL e Modelo Relacional
-  K11 - Orientação a Objetos em Java
-  K12 - Desenvolvimento Web com JSF2 e JPA2
-  K21 - Persistência com JPA2 e Hibernate
-  K22 - Desenvolvimento Web Avançado com JFS2, EJB3.1 e CDI
-  K23 - Integração de Sistemas com Webservices, JMS e EJB
-  K41 - Desenvolvimento Mobile com Android
-  K51 - Design Patterns em Java
-  K52 - Desenvolvimento Web com Struts
-  K31 - C# e Orientação a Objetos
-  K32 - Desenvolvimento Web com ASP.NET MVC

www.k19.com.br/cursos

INTRODUÇÃO



Sistemas Corporativos

Dificilmente, uma empresa consegue sobreviver sem auxílio de ferramentas computacionais. Algumas organizações necessitam de ferramentas básicas como editores de texto, planilhas ou geradores de apresentação enquanto outras necessitam de ferramentas específicas (sistemas corporativos) que contemplam todos os processos administrativos da organização.

Em geral, a complexidade no desenvolvimento e na manutenção de um sistema corporativo é alta. Essa complexidade aumenta o custo e o tempo para desenvolvê-lo e mantê-lo.

Técnicas de programação como orientação a objetos, metodologias de gerenciamento como scrum e ferramentas como Java podem diminuir o tempo e o dinheiro gastos na área de TI.



Orientação a Objetos

O paradigma de programação orientado a objetos estabelece princípios fundamentais referentes à organização de um software. Esses princípios podem diminuir consideravelmente o custo no desenvolvimento e na manutenção de sistemas corporativos.

Abaixo, algumas características dos sistemas orientados a objetos:

- Modularidade
- Encapsulamento
- Polimorfismo

Hoje em dia, a orientação a objetos é o modelo de programação mais utilizado na modelagem de sistemas corporativos.



Padrões de Projeto

Apesar de específicos, os sistemas corporativos possuem diversas características semelhantes. Consequentemente, muitos problemas se repetem em contextos distintos.

Suponha que um determinado problema ocorrerá em duzentos sistemas diferentes. Em cada sistema, esse problema pode ser resolvido de uma forma distinta. Então, globalmente, teríamos duzentas soluções para o mesmo problema. Provavelmente, algumas soluções seriam melhores que outras ou até mesmo uma delas melhor do que todas as outras.

Para não perder tempo e dinheiro elaborando soluções diferentes para o mesmo problema, poderíamos escolher uma solução como padrão e adotá-la toda vez que o problema correspondente ocorrer. Além de evitar o retrabalho, facilitaríamos a comunicação dos desenvolvedores e o entendimento técnico do sistema.

Daí surge o conceito de **padrão de projeto** ou **design pattern**. Um padrão de projeto é uma solução consolidada para um problema recorrente no desenvolvimento e manutenção de software orientado a objetos.

A referência mais importante relacionada a padrões de projeto é o livro *Design Patterns: Elements of Reusable Object-Oriented Software* (editora Addison-Wesley, 1995) dos autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Esses quatro autores são conhecidos como “Gang of Four”(GoF). Os diagramas UML apresentados nesta apostila são baseados nos diagramas desse livro.

Padrões GoF

Os padrões definidos no livro *Design Patterns: Elements of Reusable Object-Oriented Software* são denominados padrões GoF. Eles são classificados em três categorias: padrões de **criação**, **estruturais** e **comportamentais**.

PADRÕES DE CRIAÇÃO

Em um sistema orientado a objetos, a criação de certos objetos pode ser uma tarefa extremamente complexa. Podemos destacar dois problemas relacionados a criação de objetos:

- definir qual classe concreta deve ser utilizada para criar o objeto;
- definir como os objetos devem ser criados e como eles se relacionam com outros objetos do sistema.

Seguindo o princípio do encapsulamento, essa complexidade deve ser isolada. A seguir, mostraremos padrões de projetos que podem ser adotados para encapsular a criação de objetos em diversas situações distintas.

Veja abaixo um resumo do objetivo de cada padrão de criação.

Factory Method Encapsular a escolha da classe concreta a ser utilizada na criação de objetos de um determinado tipo.

Abstract Factory Encapsular a escolha das classes concretas a serem utilizadas na criação dos objetos de diversas famílias.

Builder Separar o processo de construção de um objeto de sua representação e permitir a sua criação passo-a-passo. Diferentes tipos de objetos podem ser criados com implementações distintas de cada passo.

Prototype Possibilitar a criação de novos objetos a partir da cópia de objetos existentes.

Singleton Permitir a criação de uma única instância de uma classe e fornecer um modo para recuperá-la.

Multiton Permitir a criação de uma quantidade limitada de instâncias de determinada classe e fornecer um modo para recuperá-las.

Object Pool Possibilitar o reaproveitamento de objetos.



Factory Method

Objetivo: Encapsular a escolha da classe concreta a ser utilizada na criação de objetos de um determinado tipo.

Exemplo prático

Considere um sistema bancário que precisa enviar mensagens aos seus clientes. Por exemplo, após a realização de uma compra com cartão de crédito, uma mensagem contendo informações sobre a compra pode ser enviada ao cliente.

Se esse cliente for uma pessoa física, poderá optar pelo recebimento da mensagem através de email ou SMS. Por outro lado, se for uma pessoa jurídica, poderá também receber a mensagem através de JMS (Java Message Service).

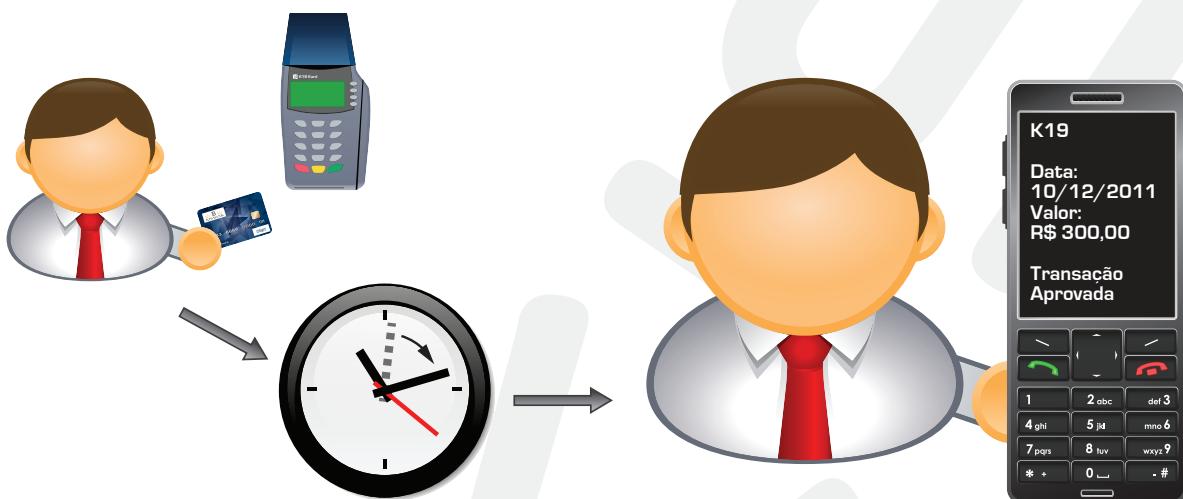


Figura 2.1: Recebimento de mensagem SMS após uma compra

Cada mecanismo de envio será implementado por uma classe. Podemos criar uma interface para padronizar essas classes e obter polimorfismo entre seus objetos.

```
1 public interface Emissor {
2     void envia(String mensagem);
3 }
```

Código Java 2.1: Emissor.java

```
1 public class EmissorSMS implements Emissor {
2     public void envia(String mensagem) {
3         System.out.println("Enviando por SMS a mensagem: ");
4         System.out.println(mensagem);
5     }
6 }
```

Código Java 2.2: EmissorSMS.java

```
1 public class EmissorEmail implements Emissor {
2     public void envia(String mensagem) {
3         System.out.println("Enviando por email a mensagem: ");
4         System.out.println(mensagem);
5     }
6 }
```

Código Java 2.3: EmissorEmail.java

```
1 public class EmissorJMS implements Emissor {
```

```

1 public void envia(String mensagem) {
2     System.out.println("Enviando por JMS a mensagem: ");
3     System.out.println(mensagem);
4 }
5
6 }
```

Código Java 2.4: EmissorJMS.java

Quando for necessário enviar um mensagem, podemos utilizar diretamente esses emissores.

```

1 Emissor emissor = new EmissorSMS();
2 emissor.envia("K19 - Treinamentos");
```

Código Java 2.5: Enviando uma mensagem por SMS

```

1 Emissor emissor = new EmissorEmail();
2 emissor.envia("K19 - Treinamentos");
```

Código Java 2.6: Enviando uma mensagem por email

```

1 Emissor emissor = new EmissorJMS();
2 emissor.envia("K19 - Treinamentos");
```

Código Java 2.7: Enviando uma mensagem por JMS

Utilizando essa abordagem, o código que deseja enviar uma mensagem referencia diretamente as classes que implementam os mecanismos de envio. Para eliminar essa referência direta, podemos adicionar um intermediário entre o código que deseja enviar uma mensagem e as classes que implementam os emissores. Esse intermediário será o responsável pela escolha da classe concreta a ser utilizada para criar o tipo de emissor adequado.

```

1 public class EmissorCreator {
2     public static final int SMS = 0;
3     public static final int EMAIL = 1;
4     public static final int JMS = 2;
5
6     public Emissor create(int tipoDeEmissor) {
7         if(tipoDeEmissor == EmissorCreator.SMS) {
8             return new EmissorSMS();
9         } else if(tipoDeEmissor == EmissorCreator.EMAIL) {
10            return new EmissorEmail();
11        } else if(tipoDeEmissor == EmissorCreator.JMS) {
12            return new EmissorJMS();
13        } else {
14            throw new IllegalArgumentException("Tipo de emissor não suportado");
15        }
16    }
17 }
```

Código Java 2.8: EmissorCreator.java

Especialização

Talvez, o sistema tenha que trabalhar com dois tipos diferentes de envio de mensagens: síncrono e assíncrono. Podemos especializar o criador de emissores, definindo subclasses.

```

1 public class EmissorAssincronoCreator extends EmissorCreator {
2     public Emissor create(int tipoDeEmissor) {
3         if(tipoDeEmissor == EmissorCreator.SMS) {
4             return new EmissorAssincronoSMS();
```

```

5     } else if(tipoDeEmissor == EmissorCreator.EMAIL) {
6         return new EmissorAssincronoEmail();
7     } else if(tipoDeEmissor == EmissorCreator.JMS) {
8         return new EmissorAssincronoJMS();
9     } else {
10        throw new IllegalArgumentException("Tipo de emissor não suportado");
11    }
12}
13}

```

Código Java 2.9: EmissorAssincronoCreator.java

```

1 public class EmissorSincronoCreator extends EmissorCreator {
2     public Emissor create(int tipoDeEmissor) {
3         if(tipoDeEmissor == EmissorCreator.SMS) {
4             return new EmissorSincronoSMS();
5         } else if(tipoDeEmissor == EmissorCreator.EMAIL) {
6             return new EmissorSincronoEmail();
7         } else if(tipoDeEmissor == EmissorCreator.JMS) {
8             return new EmissorSincronoJMS();
9         } else {
10            throw new IllegalArgumentException("Tipo de emissor não suportado");
11        }
12    }
13}

```

Código Java 2.10: EmissorSincronoCreator.java

Agora, para enviar uma mensagem, podemos acionar o criador adequado (`EmissorCreator`, `EmissorAssincronoCreator` ou `EmissorSincronoCreator`) para obter o emissor desejado.

```

1 EmissorCreator creator = new EmissorAssincronoCreator();
2 Emissor emissor = creator.create(EmissorCreator.SMS)
3 emissor.envia("K19 - Treinamentos");

```

Código Java 2.11: Enviando uma mensagem por SMS

```

1 EmissorCreator creator = new EmissorAssincronoCreator();
2 Emissor emissor = creator.create(EmissorCreator.EMAIL)
3 emissor.envia("K19 - Treinamentos");

```

Código Java 2.12: Enviando uma mensagem por email

```

1 EmissorCreator creator = new EmissorSincronoCreator();
2 Emissor emissor = creator.create(EmissorCreator.JMS)
3 emissor.envia("K19 - Treinamentos");

```

Código Java 2.13: Enviando uma mensagem por JMS

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

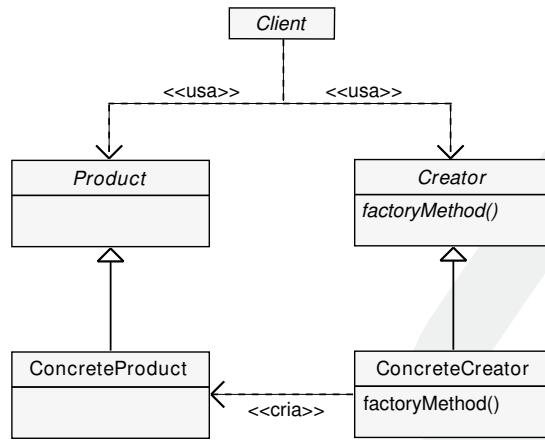


Figura 2.2: UML do padrão Factory Method

Os personagens desse padrão são:

Product (Emissor)

Classe ou interface que define o objeto a ser criado.

ConcreteProduct (EmissorSMS)

Uma implementação particular do tipo de objeto a ser criado.

Creator (EmissorCreator)

Classe ou interface que define a assinatura do método responsável pela criação do produto.
Pode possuir uma implementação padrão do método de criação do produto.

ConcreteCreator (EmissorAssincronoCreator)

Classe que implementa ou sobrescreve o método de criação do produto.



Exercícios de Fixação

- Crie um projeto Java no Eclipse chamado **FactoryMethod**.

- Defina uma interface **Emissor**.

```

1 public interface Emissor {
2     void envia(String mensagem);
3 }
  
```

Código Java 2.14: Emissor.java

- Defina as classes EmissorSMS, EmissorEmail e EmissorJMS que irão implementar a interface **Emissor**.

```

1 public class EmissorSMS implements Emissor {
  
```

```

1 public void envia(String message) {
2     System.out.println("Enviando por SMS a mensagem: ");
3     System.out.println(message);
4 }
5
6 }
```

Código Java 2.15: EmissorSMS.java

```

1 public class EmissorEmail implements Emissor {
2     public void envia(String message) {
3         System.out.println("Enviando por email a mensagem: ");
4         System.out.println(message);
5     }
6 }
```

Código Java 2.16: EmissorEmail.java

```

1 public class EmissorJMS implements Emissor {
2     public void envia(String message) {
3         System.out.println("Enviando por JMS a mensagem: ");
4         System.out.println(message);
5     }
6 }
```

Código Java 2.17: EmissorJMS.java

4 Crie uma classe para testar os emissores.

```

1 public class TestaEmissores {
2     public static void main(String[] args) {
3         Emissor emissor1 = new EmissorSMS();
4         emissor1.envia("K19 Treinamentos");
5
6         Emissor emissor2 = new EmissorEmail();
7         emissor2.envia("K19 Treinamentos");
8
9         Emissor emissor3 = new EmissorJMS();
10        emissor3.envia("K19 Treinamentos");
11    }
12 }
```

Código Java 2.18: TestaEmissores.java

5 Para tornar a classe TestaEmissores menos dependente das classes que implementam os mecanismos de envio, podemos definir uma classe intermediária que será responsável pela criação dos emissores.

```

1 public class EmissorCreator {
2     public static final int SMS = 0;
3     public static final int EMAIL = 1;
4     public static final int JMS = 2;
5
6     public Emissor create(int tipoDeEmissor) {
7         if(tipoDeEmissor == EmissorCreator.SMS) {
8             return new EmissorSMS();
9         } else if (tipoDeEmissor == EmissorCreator.EMAIL) {
10            return new EmissorEmail();
11        } else if (tipoDeEmissor == EmissorCreator.JMS) {
12            return new EmissorJMS();
13        } else {
14            throw new IllegalArgumentException("Tipo de emissor não suportado");
15        }
16    }
17 }
```

```
15 }
16 }
17 }
```

Código Java 2.19: EmissorCreator.java

- 6 Altere a classe TestaEmissores para utilizar a classe EmissorCreator.

```
1 public class TestaEmissores {
2     public static void main(String[] args) {
3         EmissorCreator creator = new EmissorCreator();
4
5         //SMS
6         Emissor emissor1 = creator.create(EmissorCreator.SMS);
7         emissor1.envia("K19 Treinamentos");
8
9         //Email
10        Emissor emissor2 = creator.create(EmissorCreator.EMAIL);
11        emissor2.envia("K19 Treinamentos");
12
13        //JMS
14        Emissor emissor3 = creator.create(EmissorCreator.JMS);
15        emissor3.envia("K19 Treinamentos");
16    }
17 }
```

Código Java 2.20: TestaEmissores.java



Abstract Factory

Objetivo: Encapsular a escolha das classes concretas a serem utilizadas na criação dos objetos de diversas famílias.

Exemplo prático

Estabelecimentos comerciais normalmente oferecem aos clientes diversas opções de pagamento. Por exemplo, clientes podem efetuar pagamentos com dinheiro, cheque, cartões de crédito ou débito, entre outros.

Pagamentos com cartões são realizados por meio de uma máquina de cartão, oferecida e instalada por empresas como Cielo e Redecard. Geralmente, essa máquina é capaz de lidar com cartões de diferentes bandeiras (como Visa e Mastercard).

Nosso objetivo é programar essas máquinas, isto é, desenvolver uma aplicação capaz de se comunicar com as diferentes bandeiras e registrar pagamentos.

No momento do pagamento, a máquina de cartão deve enviar as informações relativas à transação (como valor e senha) para a bandeira correspondente ao cartão utilizado. Além disso, a máquina deve aguardar uma resposta de confirmação ou recusa do pagamento.



Figura 2.3: Máquina de cartão interagindo com a bandeira

Na seção anterior, modelamos um sistema de envio de mensagens utilizando o padrão **Factory Method**. Para desenvolver a aplicação que executará nas máquinas de cartão, devemos implementar algo semelhante que, além de enviar mensagens, também precisa receber.

O nosso sistema será composto por objetos emissores e receptores de mensagens. Como o protocolo de comunicação de cada bandeira é diferente. Como cada bandeira possui o seu próprio protocolo de comunicação, teremos um emissor e um receptor para cada bandeira.

Criaremos fábricas específicas para cada bandeira. Essas fábricas serão responsáveis pela criação dos emissores e receptores. Para padronizá-las, podemos criar uma interface.

```

1 public interface ComunicadorFactory {
2     Emissor createEmissor();
3     Receptor createReceptor();
4 }
```

Código Java 2.21: ComunicadorFactory.java

```

1 public class VisaComunicadorFactory implements ComunicadorFactory {
2     public Emissor createEmissor() {
3         // criando um emissor Visa
4     }
5
6     public Receptor createReceptor() {
7         // criando um receptor Visa
8     }
9 }
```

Código Java 2.22: VisaComunicadorFactory.java

```

1 public class MastercardComunicadorFactory implements ComunicadorFactory {
2     public Emissor createEmissor() {
3         // criando um emissor Mastercard
4     }
5
6     public Receptor createReceptor() {
7         // criando um receptor Mastercard
8     }
9 }
```

Código Java 2.23: MastercardComunicadorFactory.java

O processo de escolha da fábrica é realizado de acordo com a bandeira do cartão utilizado.

```

1 public ComunicadorFactory getComunicadorFactory(Cartao cartao) {
2     String bandeira = cartao.getBandeira();
3     Class clazz = Class.forName(bandeira + "ComunicadorFactory");
4     return (ComunicadorFactory) clazz.newInstance();
5 }
```

Código Java 2.24: Selecionando a fábrica de acordo com as preferências do usuário

Dessa forma, o código da aplicação se torna independente da fábrica utilizada.

```

1 Cartao cartao = ...
2 ComunicadorFactory factory = getComunicadorFactory(cartao);
3
4 String transacao = ...
5 Emissor emissor = factory.getEmissor();
6 emissor.envia(transacao);
7
8 Receptor receptor = factory.getReceptor();
9 String resposta = receptor.recebe();
```

Código Java 2.25: enviando mensagens

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

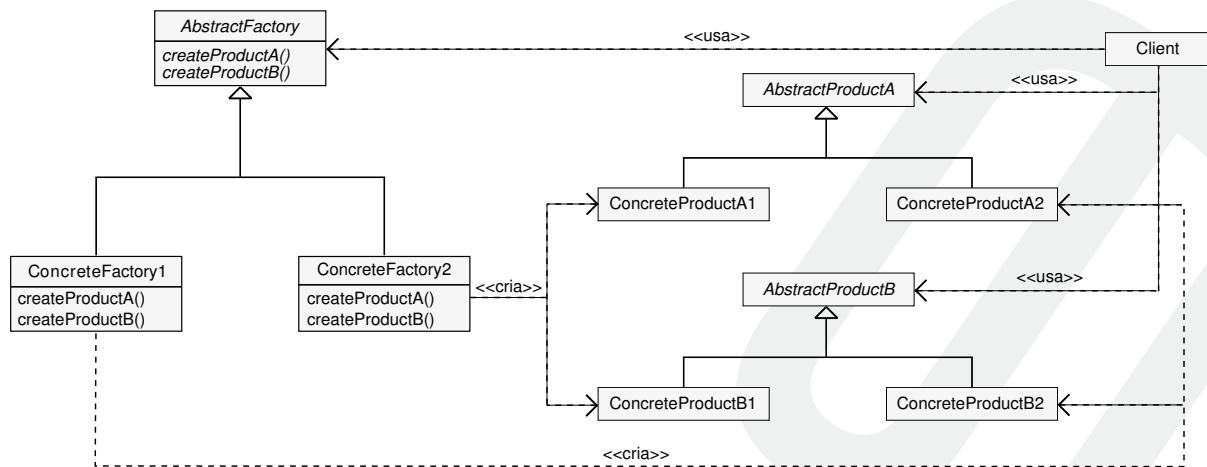


Figura 2.4: UML do padrão Abstract Factory

Os personagens desse padrão são:

AbstractFactory (ComunicadorFactory)

Interface que define as assinaturas dos métodos responsáveis pela criação dos objetos uma família.

ConcreteFactory (VisaComunicadorFactory, MastercardComunicadorFactory)

Classe que implementa os métodos de criação dos objetos de uma família.

AbstractProduct (Emissor, Receptor)

Interface que define um tipo de produto.

ConcreteProduct (EmissorVisa, ReceptorVisa, EmissorMastercard, ReceptorMastercard)

Implementação particular de um tipo de produto.

Client (Aplicação)

Usa apenas as interfaces `AbstractFactory` e `AbstractProduct`.



Abstract Factory + Factory Method

Podemos combinar os dois padrões vistos anteriormente. As implementações da Abstract Factory podem acionar os Factory Methods para criar os emissores e receptores.

```

1 public class VisaComunicadorFactory implements ComunicadorFactory {
2     private EmissorCreator emissorCreator = new EmissorCreator();
3     private ReceptorCreator receptorCreator = new ReceptorCreator();
4
5     public Emissor createEmissor() {
6         return this.emissorCreator.create(EmissorCreator.VISA);
7     }
8
9     public Receptor createReceptor() {
10        return this.receptorCreator.create(ReceptorCreator.VISA);
11    }
12 }
  
```

Código Java 2.26: VisaComunicadorFactory.java

```

1 public class MastercardComunicadorFactory implements ComunicadorFactory {
2     private EmissorCreator emissorCreator = new EmissorCreator();
3     private ReceptorCreator receptorCreator = new ReceptorCreator();
4
5     public Emissor createEmissor() {
6         return this.emissorCreator.create(EmissorCreator.MASTERCARD);
7     }
8
9     public Receptor createReceptor() {
10        return this.receptorCreator.create(ReceptorCreator.MASTERCARD);
11    }
12 }
```

Código Java 2.27: MastercardComunicadorFactory.java

Exercícios de Fixação

- 7 Crie um projeto chamado **AbstractFactory**.

- 8 Defina as interfaces **Receptor** e **Emissor**.

```

1 public interface Emissor {
2     void envia(String mensagem);
3 }
```

Código Java 2.28: Emissor.java

```

1 public interface Receptor {
2     String recebe();
3 }
```

Código Java 2.29: Receptor.java

- 9 Defina as implementações Visa e Mastercard para as interfaces **Emissor** e **Receptor**.

```

1 public class ReceptorVisa implements Receptor {
2     public String recebe() {
3         System.out.println("Recebendo mensagem da Visa.");
4         String mensagem = "Mensagem da Visa";
5         return mensagem;
6     }
7 }
```

Código Java 2.30: ReceptorVisa.java

```

1 public class ReceptorMastercard implements Receptor {
2     public String recebe() {
3         System.out.println("Recebendo mensagem da Mastercard.");
4         String mensagem = "Mensagem da Mastercard";
5         return mensagem;
6     }
7 }
```

7 }

Código Java 2.31: ReceptorMastercard.java

```

1 public class EmissorVisa implements Emissor {
2     public void envia(String mensagem) {
3         System.out.println("Enviando a seguinte mensagem para a Visa:");
4         System.out.println(mensagem);
5     }
6 }
```

Código Java 2.32: EmissorVisa.java

```

1 public class EmissorMastercard implements Emissor {
2     public void envia(String mensagem) {
3         System.out.println("Enviando a seguinte mensagem para a Mastercard:");
4         System.out.println(mensagem);
5     }
6 }
```

Código Java 2.33: EmissorMastercard

10 Defina as fábricas de emissores e receptores.

```

1 public class EmissorCreator {
2     public static final int VISA = 0;
3     public static final int MASTERCARD = 1;
4
5     public Emissor create(int tipoDoEmissor) {
6         if(tipoDoEmissor == EmissorCreator.VISA) {
7             return new EmissorVisa();
8         } else if(tipoDoEmissor == EmissorCreator.MASTERCARD) {
9             return new EmissorMastercard();
10        } else {
11            throw new IllegalArgumentException("Tipo de emissor não suportado");
12        }
13    }
14 }
```

Código Java 2.34: EmissorCreator.java

```

1 public class ReceptorCreator {
2     public static final int VISA = 0;
3     public static final int MASTERCARD = 1;
4
5     public Receptor create(int tipoDoReceptor) {
6         if(tipoDoReceptor == ReceptorCreator.VISA) {
7             return new ReceptorVisa();
8         } else if (tipoDoReceptor == ReceptorCreator.MASTERCARD) {
9             return new ReceptorMastercard();
10        } else {
11            throw new IllegalArgumentException("Tipo de receptor não suportado.");
12        }
13    }
14 }
```

Código Java 2.35: ReceptorCreator.java

11 Crie uma interface **ComunicadorFactory** para padronizar as fábricas de emissores e receptores.

```

1 public interface ComunicadorFactory {
2     Emissor createEmissor();
3     Receptor createReceptor();
4 }
```

Código Java 2.36: ComunicadorFactory.java

- 12 Crie uma fábrica de emissores e receptores Visa.

```

1 public class VisaComunicadorFactory implements ComunicadorFactory {
2     private EmissorCreator emissorCreator = new EmissorCreator();
3     private ReceptorCreator receptorCreator = new ReceptorCreator();
4
5     public Emissor createEmissor() {
6         return emissorCreator.create(EmissorCreator.VISA);
7     }
8
9     public Receptor createReceptor() {
10        return receptorCreator.create(ReceptorCreator.VISA);
11    }
12 }
```

Código Java 2.37: VisaComunicadorFactory.java

- 13 Crie uma fábrica de emissores e receptores Mastercard.

```

1 public class MastercardComunicadorFactory implements ComunicadorFactory {
2     private EmissorCreator emissorCreator = new EmissorCreator();
3     private ReceptorCreator receptorCreator = new ReceptorCreator();
4
5     public Emissor createEmissor() {
6         return emissorCreator.create(EmissorCreator.MASTERCARD);
7     }
8
9     public Receptor createReceptor() {
10        return receptorCreator.create(ReceptorCreator.MASTERCARD);
11    }
12 }
```

Código Java 2.38: MastercardComunicadorFactory.java

- 14 Faça uma classe para testar a fábrica de emissores e receptores Visa.

```

1 public class TestaVisaComunicadorFactory {
2     public static void main(String[] args) {
3         ComunicadorFactory comunicadorFactory = new VisaComunicadorFactory();
4
5         String transacao = "Valor=560;Senha=123";
6         Emissor emissor = comunicadorFactory.createEmissor();
7         emissor.envia(transacao);
8
9         Receptor receptor = comunicadorFactory.createReceptor();
10        String mensagem = receptor.recebe();
11        System.out.println(mensagem);
12    }
13 }
```

Código Java 2.39: TestaVisaComunicadorFactory.java



Exercícios Complementares

- 1 Faça uma classe para testar a fábrica de emissores e receptores da Mastercard.



Builder

Objetivo: Separar o processo de construção de um objeto de sua representação e permitir a sua criação passo-a-passo. Diferentes tipos de objetos podem ser criados com implementações distintas de cada passo.

Exemplo prático

Estamos desenvolvendo um sistema para gerar boletos bancários. No Brasil, a FEBRABAN (Federação Brasileira de Bancos) define regras gerais para os boletos. Contudo, cada instituição bancária define também as suas próprias regras específicas para o formato dos dados dos boletos.

Segundo a FEBRABAN, os elementos principais relacionados a um boleto são:

Sacado: Pessoa ou empresa responsável pelo pagamento do boleto.

Cedente: Pessoa ou empresa que receberá o pagamento do boleto.

Banco: Instituição que receberá o pagamento do sacado e creditará o valor na conta do cedente.

Código de Barras: Representação gráfica das informações do boleto.

Linha Digitável: Representação numérica das informações do boleto.

Nosso Número: Código de identificação do boleto utilizado pela instituição bancária e pelo cedente para controle dos pagamentos.

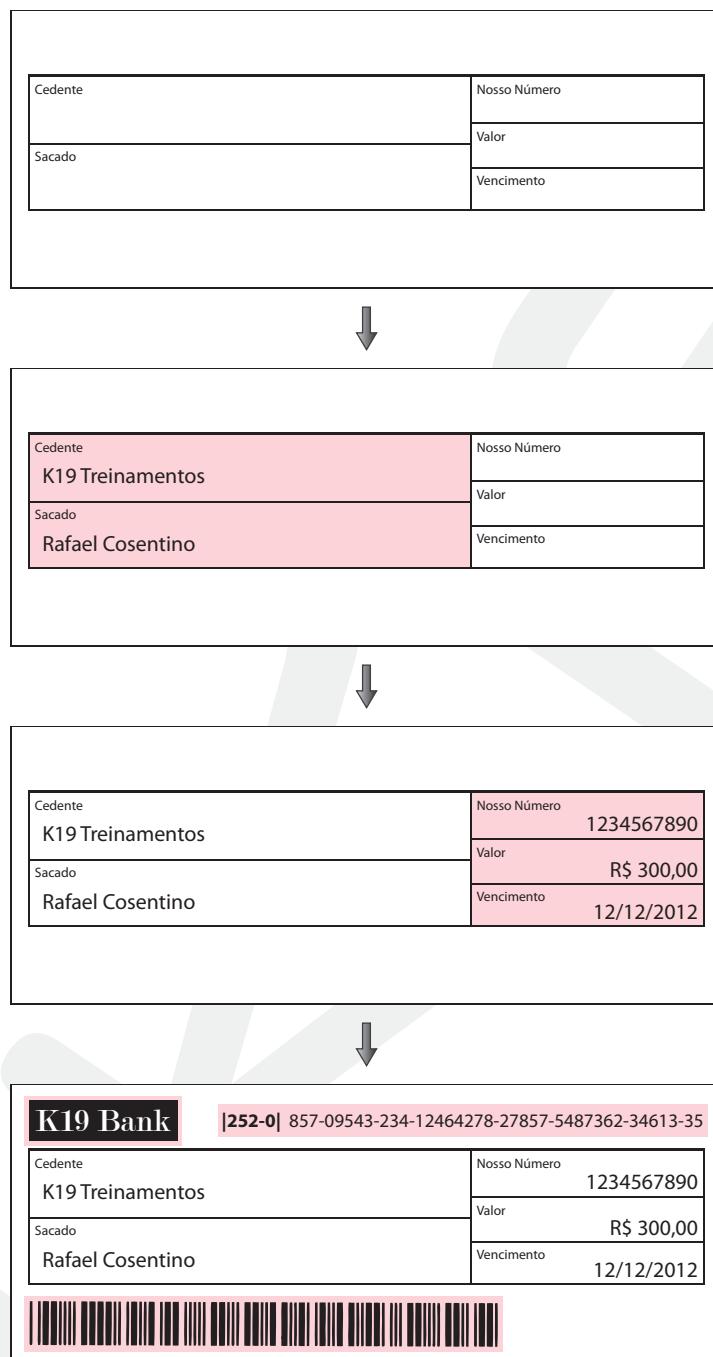


Figura 2.5: Passo-a-passo da criação de um boleto

Para gerar um boleto, devemos definir todas essas informações. Para encapsular a complexidade desse processo, poderíamos definir classes responsáveis pela criação dos boletos de acordo com a instituição bancária. Inclusive, poderíamos padronizar essas classes através de uma interface.

```

1 public interface BoletoBuilder {
2     void buildSacado(String sacado);
3     void buildCedente(String cedente);
4     void buildValor(double valor);
5     void buildVencimento(Calendar vencimento);
6     void buildNossoNumero(int nossoNumero);

```

```

7 void buildCodigoDeBarras();
8 void buildLogotipo();
9
10 Boleto getBoleto();
11 }
```

Código Java 2.40: BoletoBuilder.java

```

1 public class BBBoletoBuilder implements BoletoBuilder {
2     // implementação dos métodos seguindo as regras da FEBRABAN e do BB
3 }
```

Código Java 2.41: BBBoletoBuilder.java

```

1 public class ItauBoletoBuilder implements BoletoBuilder {
2     // implementação dos métodos seguindo as regras da FEBRABAN e do Itaú
3 }
```

Código Java 2.42: ItauBoletoBuilder.java

```

1 public class BradescoBoletoBuilder implements BoletoBuilder {
2     // implementação dos métodos seguindo as regras da FEBRABAN e do Bradesco
3 }
```

Código Java 2.43: BradescoBoletoBuilder.java

Agora, poderíamos obter de alguma fonte externa (usuário, banco de dados, arquivos e etc) as informações necessárias e gerar um boleto utilizando o montador adequado.

```

1 public class GeradorDeBoleto {
2     private BoletoBuilder boletoBuilder;
3
4     public GeradorDeBoleto(BoletoBuilder boletoBuilder) {
5         this.boletoBuilder = boletoBuilder;
6     }
7
8     public Boleto geraBoleto() {
9         String sacado = ...
10        this.boletoBuilder.buildSacado(sacado);
11
12        String cedente = ...
13        this.boletoBuilder.buildCedente(cedente);
14
15        double valor = ...
16        this.boletoBuilder.buildValor(valor);
17
18        Calendar vencimento = ...
19        this.boletoBuilder.buildVencimento(vencimento);
20
21        this.boletoBuilder.buildCodigoDeBarras();
22
23        this.boletoBuilder.buildLogotipo();
24
25        return this.boletoBuilder.getBoleto();
26    }
27 }
```

Código Java 2.44: GeradorDeBoleto.java

Quando for necessário gerar um boleto, devemos escolher uma implementação da interface BoletoBuilder.

```

1 BoletoBuilder boletoBuilder = new BBBoletoBuilder();
2 GeradorDeBoleto geradorDeBoleto = new GeradorDeBoleto(boletoBuilder);
3 Boleto boleto = geradorDeBoleto.geraBoleto();

```

Código Java 2.45: Gerando um boleto

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

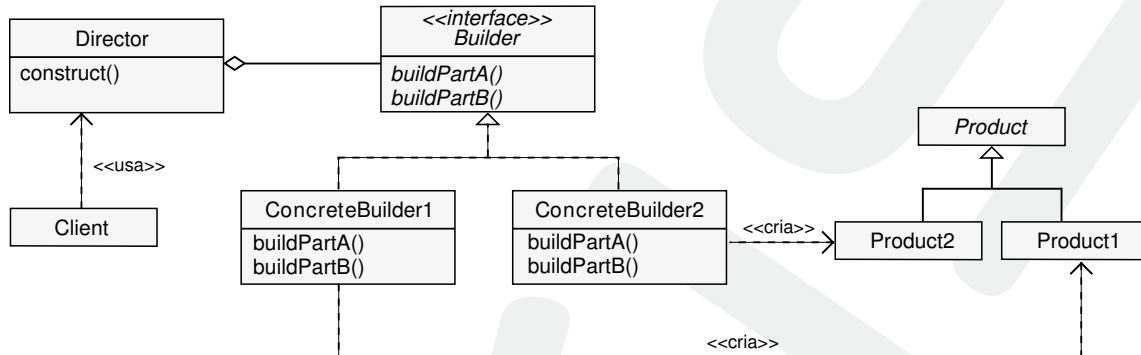


Figura 2.6: UML do padrão Builder

Os personagens desse padrão são:

Product (Boleto)

Define os objetos que devem ser construídos pelos Builders.

Builder (BoletoBuilder)

Interface que define os passos para a criação de um produto.

ConcreteBuilder (BBBoletoBuilder, ItauBoletoBuilder, BradescoBoletoBuilder)

Constrói um produto específico implementando a interface Builder.

Director (GeradorDeBoleto)

Aciona os métodos de um Builder para construir um produto.



Exercícios de Fixação

15 Crie um projeto chamado **Builder**.

16 Defina uma interface para os boletos.

```

1 public interface Boleto {
2     String getSacado();
3     String getCedente();
4     double getValor();
5     Calendar getVencimento();

```

```
6 |     int getNossoNumero();
7 |     String toString();
8 | }
```

Código Java 2.46: Boleto.java

17 Defina a classe **BBBoleto** que implementa a interface **Boleto**.

```
1 | public class BBBoleto implements Boleto {
2 |     private String sacado;
3 |     private String cedente;
4 |     private double valor;
5 |     private Calendar vencimento;
6 |     private int nossoNumero;
7 |
8 |     public BBBoleto(String sacado, String cedente, double valor, Calendar vencimento, ←
9 |         int nossoNumero) {
10 |         this.sacado = sacado;
11 |         this.cedente = cedente;
12 |         this.valor = valor;
13 |         this.vencimento = vencimento;
14 |         this.nossoNumero = nossoNumero;
15 |     }
16 |
17 |     public String getSacado() {
18 |         return this.sacado;
19 |     }
20 |
21 |     public String getCedente() {
22 |         return this.cedente;
23 |     }
24 |
25 |     public double getValor() {
26 |         return this.valor;
27 |     }
28 |
29 |     public Calendar getVencimento() {
30 |         return this.vencimento;
31 |     }
32 |
33 |     public int getNossoNumero() {
34 |         return this.nossoNumero;
35 |     }
36 |
37 |     public String toString() {
38 |         StringBuilder stringBuilder = new StringBuilder();
39 |         stringBuilder.append("Boleto BB");
40 |         stringBuilder.append("\n");
41 |         stringBuilder.append("Sacado: " + this.sacado);
42 |         stringBuilder.append("\n");
43 |
44 |         stringBuilder.append("Cedente: " + this.cedente);
45 |         stringBuilder.append("\n");
46 |
47 |         stringBuilder.append("Valor: " + this.valor);
48 |         stringBuilder.append("\n");
49 |
50 |         stringBuilder.append("Vencimento: " + this.sacado);
51 |         stringBuilder.append("\n");
52 |
53 |         SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");
54 |         String format = simpleDateFormat.format(this.vencimento.getTime());
55 |         stringBuilder.append("Vencimento: " + format);
56 |         stringBuilder.append("\n");
57 |
58 |         stringBuilder.append("Nosso Número: " + this.nossoNumero);
```

```

59     stringBuilder.append("\n");
60
61     return stringBuilder.toString();
62 }
63 }
```

Código Java 2.47: Boleto.java

- 18 Defina uma interface que irá padronizar as classes responsáveis pela criação dos boletos.

```

1 public interface BoletoBuilder {
2     void buildSacado(String sacado);
3     void buildCedente(String cedente);
4     void buildValor(double valor);
5     void buildVencimento(Calendar vencimento);
6     void buildNossoNumero(int nossoNumero);
7
8     Boleto getBoleto();
9 }
```

Código Java 2.48: BoletoBuilder.java

- 19 Defina uma classe responsável pela criação de boletos do banco BB.

```

1 public class BBBoletoBuilder implements BoletoBuilder {
2     private String sacado;
3     private String cedente;
4     private double valor;
5     private Calendar vencimento;
6     private int nossoNumero;
7
8     public void buildSacado(String sacado) {
9         this.sacado = sacado;
10    }
11
12    public void buildCedente(String cedente) {
13        this.cedente = cedente;
14    }
15
16    public void buildValor(double valor) {
17        this.valor = valor;
18    }
19
20    public void buildVencimento(Calendar vencimento) {
21        this.vencimento = vencimento;
22    }
23
24    public void buildNossoNumero(int nossoNumero) {
25        this.nosoNumero = nossoNumero;
26    }
27
28    public Boleto getBoleto() {
29        return new BBBoleto(sacado, cedente, valor, vencimento, nossoNumero);
30    }
31 }
```

Código Java 2.49: BBBoletoBuilder.java

- 20 Faça uma classe para gerar boletos.

```
1 public class GeradorDeBoleto {
```

```

2  private BoletoBuilder boletoBuilder;
3
4  public GeradorDeBoleto(BoletoBuilder boletoBuilder) {
5      this.boletoBuilder = boletoBuilder;
6  }
7
8  public Boleto geraBoleto() {
9
10     this.boletoBuilder.buildSacado("Marcelo Martins");
11
12     this.boletoBuilder.buildCedente("K19 Treinamentos");
13
14     this.boletoBuilder.buildValor(100.54);
15
16     Calendar vencimento = Calendar.getInstance();
17     vencimento.add(Calendar.DATE, 30);
18     this.boletoBuilder.buildVencimento(vencimento);
19
20     this.boletoBuilder.buildNossoNumero(1234);
21
22     Boleto boleto = boletoBuilder.getBoleto();
23
24     return boleto;
25 }
26 }
```

Código Java 2.50: GeradorDeBoleto.java

- 21** Faça uma classe para testar o gerador de boleto.

```

1  public class TestaGeradorDeBoleto {
2  public static void main(String[] args) {
3      BoletoBuilder boletoBuilder = new BBBoletoBuilder();
4      GeradorDeBoleto geradorDeBoleto = new GeradorDeBoleto(boletoBuilder);
5      Boleto boleto = geradorDeBoleto.geraBoleto();
6      System.out.println(boleto);
7  }
8 }
```

Código Java 2.51: TestaGeradorDeBoleto.java



Prototype

Objetivo: Possibilitar a criação de novos objetos a partir da cópia de objetos existentes.

Exemplo prático

Estamos desenvolvendo um sistema de anúncios semelhante ao do Google Adwords. Nesse sistema, os usuários poderão criar campanhas e configurá-las de acordo com as suas necessidades. Uma campanha é composta por diversas informações, entre elas:

- Uma lista de anúncios.
- O valor diário máximo que deve ser gasto pela campanha.
- O valor máximo por exibição de anúncio.
- As datas de início e término.

Nesse tipo de sistema, os usuários geralmente criam campanhas com configurações extremamente parecidas. Dessa forma, seria interessante que o sistema tivesse a capacidade de criar uma campanha a partir de uma outra campanha já criada anteriormente, para que as configurações pudessem ser aproveitadas.

Campanha Anual

[Treinamentos de Java e .NET](#)

www.k19.com.br

Prepare-se para o mercado de trabalho com os treinamentos da K19

Campanha de Verão

[K19 - Padrões de Projeto](#)

www.k19.com.br

Novo treinamento de padrões de projeto da K19

Figura 2.7: Criando um anúncio a partir de um pré-existente

```
1 Campanha nova = velha.criaUmaCopia();
```

Código Java 2.52: Criando uma campanha como cópia de outra

Vários objetos do nosso sistema poderiam ter essa capacidade. Seria interessante definir uma interface para padronizar e marcar os objetos com essas características.

```
1 public interface Prototype<T> {
2     T clone();
3 }
```

Código Java 2.53: Prototype.java

Depois, devemos implementar a interface Prototype nas classes que devem possuir a característica que desejamos.

```
1 public class Campanha implements Prototype<Campanha> {
2     // atributos e métodos
3
4     public Campanha clone() {
5         // lógica para criar uma cópia da campanha this
6     }
7 }
```

Código Java 2.54: Campanha.java

Quando o usuário quiser criar uma campanha com as mesmas configurações de uma campanha já criada, devemos escrever um código semelhante a este:

```
1 Campanha campanha1 = ...
2 Campanha campanha2 = campanha1.clone();
3 campanha2.setNome("K19 - Campanha de Verão");
4 campanha2.getAnuncios().get(0).setTitulo("K19 - Padrões de Projeto");
5 campanha2.getAnuncios().get(0).setTexto("Novo treinamento de Padrões de Projeto da ←
    K19");
```

Código Java 2.55: clonando uma campanha

Organização

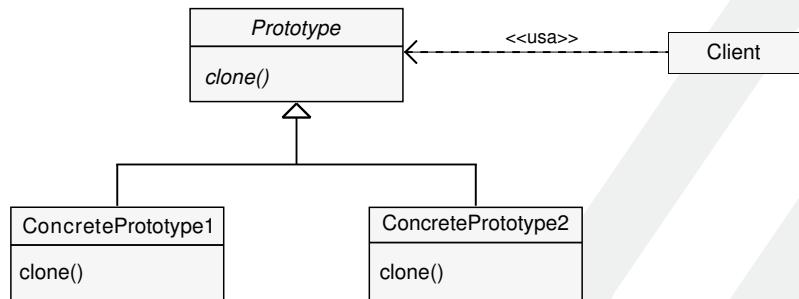


Figura 2.8: UML do padrão Prototype

Os personagens desse padrão são:

Prototype

Abstração dos objetos que possuem a capacidade de se auto copiar.

ConcretePrototype (Campanha)

Classe que define um tipo particular de objeto que pode ser clonado.

Client

Classe que cria novos objetos a partir da interface definida por Prototype.



Exercícios de Fixação

22 Crie um projeto Java chamado **Prototype**.

23 Defina uma interface **Prototype**.

```

1 public interface Prototype<T> {
2     T clone();
3 }
  
```

Código Java 2.56: Prototype.java

24 Defina uma classe **Campanha** que implementa a interface **Prototype**.

```

1 public class Campanha implements Prototype<Campanha> {
2     private String nome;
3     private Calendar vencimento;
4     private Set<String> palavrasChave;
5
6     public Campanha(String nome, Calendar vencimento, Set<String> palavrasChave) {
7         this.nome = nome;
8         this.vencimento = vencimento;
9         this.palavrasChave = palavrasChave;
10    }
11 }
  
```

```

12 public String getNome() {
13     return nome;
14 }
15
16 public Calendar getVencimento() {
17     return vencimento;
18 }
19
20 public Set<String> getPalavrasChave() {
21     return palavrasChave;
22 }
23
24 public Campanha clone() {
25     String nome = "Cópia da Campanha: " + this.nome;
26     Calendar vencimento = (Calendar) this.vencimento.clone();
27     Set<String> palavrasChave = new HashSet<String>(this.palavrasChave);
28     Campanha campanha = new Campanha(nome, vencimento, palavrasChave);
29
30     return campanha;
31 }
32
33 public String toString() {
34     StringBuffer buffer = new StringBuffer();
35     buffer.append("-----");
36     buffer.append("\n");
37     buffer.append("Nome da Campanha: ");
38     buffer.append(this.nome);
39     buffer.append("\n");
40
41     SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");
42     String format = simpleDateFormat.format(this.vencimento.getTime());
43     buffer.append("Vencimento: " + format);
44     buffer.append("\n");
45
46     buffer.append("Palavras-chave: \n");
47     for (String palavraChave : this.palavrasChave) {
48         buffer.append(" - " + palavraChave);
49         buffer.append("\n");
50     }
51     buffer.append("-----");
52     buffer.append("\n");
53
54     return buffer.toString();
55 }
56 }
```

Código Java 2.57: Campanha.java

- 25 Faça uma classe para testar o método **clone**.

```

1 public class TestaPrototype {
2     public static void main(String[] args) {
3         String nome = "K19";
4
5         Calendar vencimento = Calendar.getInstance();
6         vencimento.add(Calendar.DATE, 30);
7
8         Set<String> hashSet = new HashSet<String>();
9
10        hashSet.add("curso");
11        hashSet.add("java");
12        hashSet.add("k19");
13
14        Campanha campanha = new Campanha(nome, vencimento, hashSet);
15        System.out.println(campanha);
16
17        Campanha clone = campanha.clone();
```

```

18     System.out.println(clone);
19 }
20 }
```

Código Java 2.58: TestaPrototype.java



Singleton

Objetivo: Permitir a criação de uma única instância de uma classe e fornecer um modo para recuperá-la.

Exemplo prático

Suponha que estamos desenvolvendo um sistema que possui configurações globais obtidas a partir de um arquivo de propriedades. Essas configurações podem ser armazenadas em um objeto.

```

1 public class Configuracao {
2     private Map<String, String> propriedades = new HashMap<String, String>();
3
4     public Configuracao() {
5         // carrega as propriedades obtidas do arquivo de configuracao
6     }
7
8     public String getPropriedade(String nomeDaPropriedade) {
9         return this.propriedades.get(nomeDaPropriedade);
10    }
11 }
```

Código Java 2.59: Configuracao.java

Não desejamos que existam mais do que um objeto dessa classe ao mesmo tempo no sistema. Para garantir essa restrição, podemos tornar o construtor da classe Configuracao privado e implementar um método estático que controle a criação do único objeto que deve existir.

```

1 public class Configuracao {
2     private static Configuracao instance;
3
4     private Map<String, String> propriedades = new HashMap<String, String>();
5
6     private Configuracao() {
7         // carrega as propriedades obtidas do arquivo de configuracao
8     }
9
10    public static Configuracao getInstance() {
11        if(Configuracao.instance == null) {
12            Configuracao.instance = new Configuracao();
13        }
14        return Configuracao.instance;
15    }
16
17    public String getPropriedade(String nomeDaPropriedade) {
18        return this.propriedades.get(nomeDaPropriedade);
19    }
20 }
```

Código Java 2.60: Configuracao.java

De qualquer ponto do código do nosso sistema, podemos acessar o objeto que contém as configurações da aplicação.

```
1 String timeZone = Configuracao.getInstance().getPropriedade("time-zone");
```

Código Java 2.61: acessando as propriedades

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

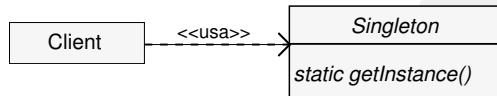


Figura 2.9: UML do padrão Singleton

O personagem desse padrão é:

Singleton (Configuracao)

Classe que permite a criação de uma única instância e fornece um método estático para recuperá-la.



Exercícios de Fixação

26 Crie um projeto Java chamado **Singleton**.

27 Crie a classe **Configuração** e impeça que mais de um objeto exista ao mesmo tempo no sistema.

```

1 public class Configuracao {
2     private Map<String, String> propriedades;
3     private static Configuracao instance;
4
5     private Configuracao() {
6         this.propriedades = new HashMap<String, String>();
7         this.propriedades.put("time-zone", "America/Sao_Paulo");
8         this.propriedades.put("currency-code", "BRL");
9     }
10
11    public static Configuracao getInstance() {
12        if (Configuracao.instance == null) {
13            Configuracao.instance = new Configuracao();
14        }
15        return Configuracao.instance;
16    }
17
18    public String getPropriedade(String nomeDaPropriedade) {
19        return this.propriedades.get(nomeDaPropriedade);
20    }
21 }
```

Código Java 2.62: Configuracao.java

28 Teste a classe **Configuracao**.

```

1 public class TestaSingleton {
2     public static void main(String[] args) {
3         Configuracao configuracao = Configuracao.getInstance();
4         System.out.println(configuracao.getPropriedade("time-zone"));
5         System.out.println(configuracao.getPropriedade("currency-code"));
6     }
7 }
```

Código Java 2.63: TestaSingleton.java



Multiton (não GoF)

Objetivo: Permitir a criação de uma quantidade limitada de instâncias de determinada classe e fornecer um modo para recuperá-las.

Exemplo prático

Estamos desenvolvendo uma aplicação que possuirá diversos temas para interface do usuário. O número de temas é limitado e cada usuário poderá escolher o de sua preferência. Podemos implementar os temas através de uma classe.

```

1 public class Tema {
2     private String nome;
3     private Color corDoFundo;
4     private Color corDaFonte;
5
6     // GETTERS AND SETTERS
7 }
```

Código Java 2.64: Tema.java

Agora, devemos criar os temas que serão disponibilizados para a escolha dos usuários.

```

1 Tema tema1 = new Tema();
2 tema1.setNome("Sky");
3 tema1.setCorDoFundo(Color.BLUE);
4 tema1.setCorDaFonte(Color.BLACK);
5
6 Tema tema2 = new Tema();
7 tema2.setNome("Fire");
8 tema2.setCorDoFundo(Color.RED);
9 tema2.setCorDaFonte(Color.WHITE);
```

Código Java 2.65: Tema.java

Queremos controlar os temas criados de maneira centralizada e acessá-los de qualquer ponto da aplicação garantindo que exista apenas uma instância de cada tema. Podemos expandir a proposta do padrão Singleton para resolver esse problema.

O construtor da classe Tema deve ser privado para que os objetos não sejam criados em qualquer lugar da aplicação. Além disso, essa classe deve disponibilizar um método para que as instâncias possam ser acessadas globalmente.

```

1 public class Tema {
2     private String nome;
3     private Color corDoFundo;
```

```

4  private Color corDaFonte;
5
6  private static Map<String, Tema> temas = new HashMap<String, Tema>();
7
8  public static final String SKY = "Sky";
9  public static final String FIRE = "Fire";
10
11 static {
12     Tema tema1 = new Tema();
13     tema1.setNome(Tema.SKY);
14     tema1.setCorDoFundo(Color.BLUE);
15     tema1.setCorDaFonte(Color.BLACK);
16
17     Tema tema2 = new Tema();
18     tema2.setNome(Tema.FIRE);
19     tema2.setCorDoFundo(Color.RED);
20     tema2.setCorDaFonte(Color.WHITE);
21
22     temas.put(tema1.getNome(), tema1);
23     temas.put(tema2.getNome(), tema2);
24 }
25
26 private Tema() {
27 }
28
29 public static Tema getInstance(String nomeDoTema) {
30     return Tema.temas.get(nomeDoTema);
31 }
32
33 // GETTERS AND SETTERS
34 }
```

Código Java 2.66: Tema.java

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

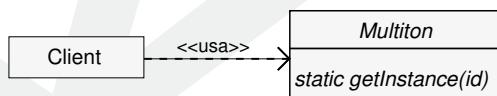


Figura 2.10: UML do padrão Multiton

O personagem desse padrão é:

Multiton (Tema)

Classe que permite a criação de uma quantidade limitada de instâncias e fornece um método estático para recuperá-las.



Exercícios de Fixação

- 29 Crie um projeto chamado **Multiton**.
- 30 Crie a classe **Tema** e pré defina os temas “Fire” e “Sky” utilizando o padrão **Multiton**.

```

1 public class Tema {
2     private String nome;
3     private Color corDoFundo;
4     private Color corDaFonte;
5
6     private static Map<String, Tema> temas = new HashMap<String, Tema>();
7
8     public static final String SKY = "Sky";
9     public static final String FIRE = "Fire";
10
11    static {
12        Tema tema1 = new Tema();
13        tema1.setNome(Tema.SKY);
14        tema1.setCorDoFundo(Color.BLUE);
15        tema1.setCorDaFonte(Color.BLACK);
16
17        Tema tema2 = new Tema();
18        tema2.setNome(Tema.FIRE);
19        tema2.setCorDoFundo(Color.RED);
20        tema2.setCorDaFonte(Color.WHITE);
21
22        temas.put(tema1.getNome(), tema1);
23        temas.put(tema2.getNome(), tema2);
24    }
25
26    private Tema() {
27    }
28
29    public static Tema getInstance(String nomeDoTema) {
30        return Tema.temas.get(nomeDoTema);
31    }
32
33    public String getNome() {
34        return nome;
35    }
36
37    public void setNome(String nome) {
38        this.nome = nome;
39    }
40
41    public Color getCorDoFundo() {
42        return corDoFundo;
43    }
44
45    public void setCorDoFundo(Color corDoFundo) {
46        this.corDoFundo = corDoFundo;
47    }
48
49    public Color getCorDaFonte() {
50        return corDaFonte;
51    }
52
53    public void setCorDaFonte(Color corDaFonte) {
54        this.corDaFonte = corDaFonte;
55    }
56}

```

Código Java 2.67: Tema.java

31 Teste a classe Tema.

```

1 public class TestaTema {
2     public static void main(String[] args) {
3         Tema temaFire = Tema.getInstance(Tema.FIRE);
4         System.out.println("Tema " + temaFire.getNome());
5         System.out.println("Cor Da Fonte : " + temaFire.getCorDaFonte());

```

```
6 System.out.println("Cor Do Fundo : " + temaFire.getCorDoFundo());
7 Tema temaFire2 = Tema.getInstance(Tema.FIRE);
8
9 System.out.println("-----");
10 System.out.println("Comparando as referências...");
11 System.out.println(temaFire == temaFire2);
12 }
13 }
14 }
```

Código Java 2.68: TestaTema.java



Object Pool (não GoF)

Objetivo: Possibilitar o reaproveitamento de objetos.

Considere a seguinte situação. Pretendemos ir a um restaurante sem ter, contudo, efetuado uma reserva. Ao chegar no restaurante, podemos nos deparar com duas situações: (i) há pelo menos uma mesa livre e podemos nos sentar ou (ii) todas as mesas estão ocupadas e precisamos esperar até que uma mesa seja liberada.

Obviamente, a quantidade de mesas do restaurante é limitada. Além disso, o restaurante acomoda um número máximo de pessoas por limitações físicas ou para seguir a legislação.

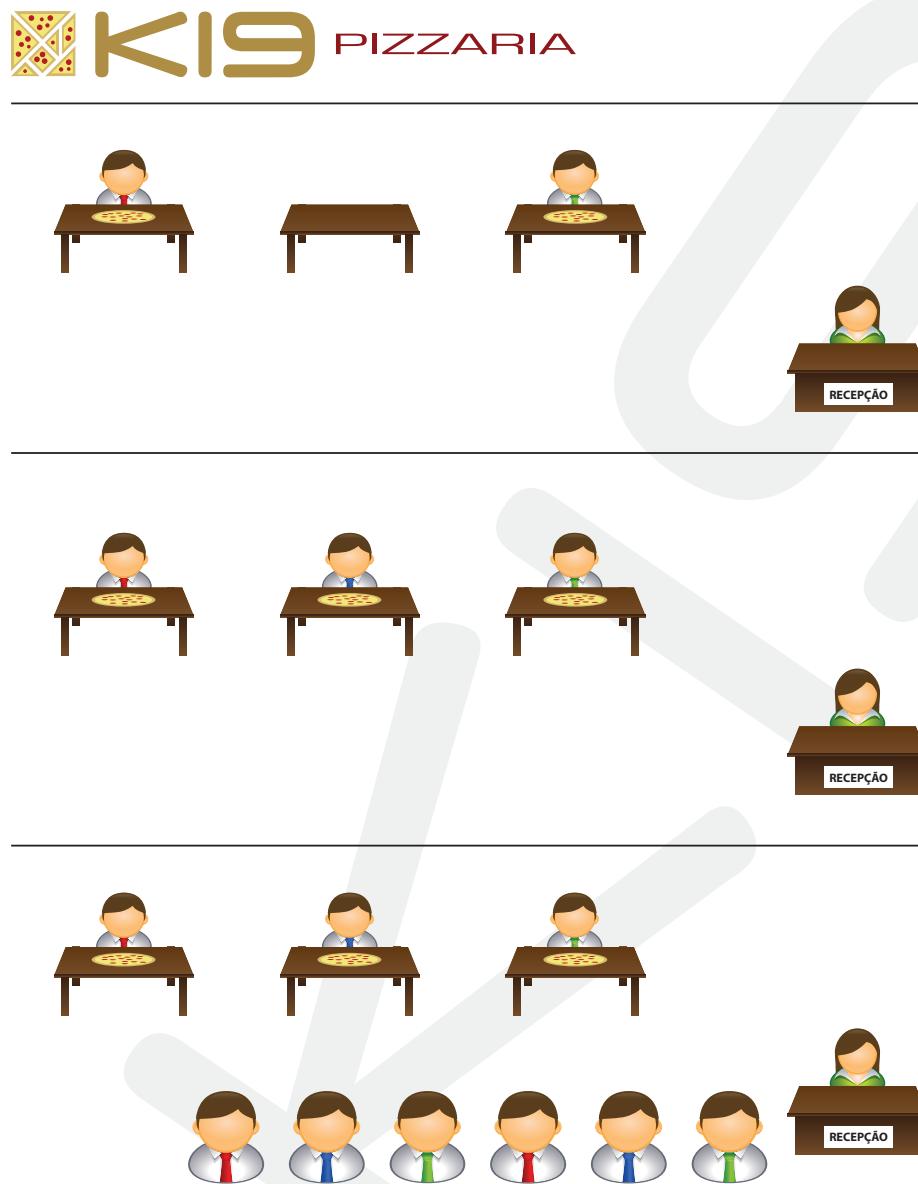


Figura 2.11: Clientes chegando a um restaurante

Essa é uma situação típica em que recursos limitados devem ser reutilizados. O restaurante não adquire novas mesas a medida que clientes chegam ao restaurante e as mesas são reutilizadas por novos clientes assim que são liberadas.

Exemplo prático

Estamos desenvolvendo um sistema para uma empresa com uma quantidade muito grande de projetos. Esse sistema deve controlar os recursos utilizados nos projetos. De maneira genérica, um recurso pode ser um funcionário, uma sala, um computador, um carro, etc.

Podemos implementar classes especializadas no controle de cada tipo de recurso utilizado nos projetos. Além disso, seria interessante padronizar essas classes através de uma interface.

```
1 public interface Pool<T> {
```

```

1   T acquire();
2   void release(T t);
3 }
```

Código Java 2.69: Pool.java

```

1 public class FuncionarioPool implements Pool<Funcionario> {
2     private Collection<Funcionario> funcionarios;
3
4     public FuncionarioPool() {
5         // inicializa a coleção de funcionários
6     }
7
8     public Funcionario acquire() {
9         // escolhe um funcionário da coleção
10    }
11
12    public void release(Funcionario f) {
13        // adiciona o funcionário na coleção
14    }
15 }
```

Código Java 2.70: FuncionarioPool.java

```

1 public class SalaPool implements Pool<Sala> {
2     private Collection<Sala> salas;
3
4     public SalaPool() {
5         // inicializa a coleção de salas
6     }
7
8     public Sala acquire() {
9         // escolhe uma sala da coleção
10    }
11
12    public void release(Sala sala) {
13        // adiciona a sala na coleção
14    }
15 }
```

Código Java 2.71: SalaPool.java

Agora, quando um projeto necessita de um recurso como funcionários ou salas basta utilizar os pools.

```

1 Pool<Sala> poolSalas = ...
2 Pool<Funcionario> poolFuncionario = ...
3
4 // obtendo os recursos
5 Sala sala = poolSalas.acquire();
6 Funcionario funcionario = poolFuncionario.acquire();
7
8 // usando os recursos
9 ...
10
11 // liberando os recursos
12 poolSalas.release(sala);
13 poolFuncionarios.release(funcionario)
```

Código Java 2.72: interagindo com um pool

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

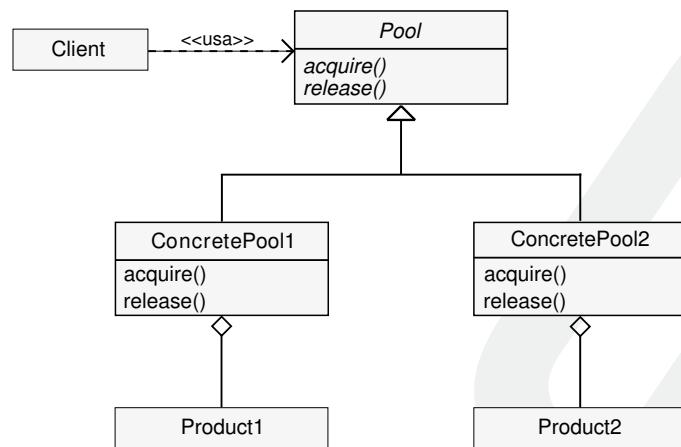


Figura 2.12: UML do padrão Object Pool

Os personagens desse padrão são:

Product (Funcionario, Sala)

Define os objetos gerenciados pelos Pools.

Pool (Pool)

Interface dos objetos que controlam a aquisição e a liberação dos Products.

ConcretePool (SalaPool, FuncionarioPool)

Implementação particular de um Pool que gerencia um Product específico.



Exercícios de Fixação

- 32 Crie um projeto chamado **ObjectPool**.

- 33 Defina as classes **Sala** e **Funcionario**.

```

1 public class Funcionario {
2     private String nome;
3
4     public Funcionario(String nome) {
5         this.nome = nome;
6     }
7
8     public String getNome() {
9         return this.nome;
10    }
11 }

```

Código Java 2.73: Funcionario.java

- 34 Defina a interface **Pool**.

```

1 public interface Pool<T> {
2     T acquire();
3     void release(T t);
4 }
```

Código Java 2.74: Pool.java

- 35 Defina a classe **FuncionarioPool** que implementa a interface **Pool**.

```

1 public class FuncionarioPool implements Pool<Funcionario> {
2     private List<Funcionario> funcionarios;
3
4     public FuncionarioPool() {
5         this.funcionarios = new ArrayList<Funcionario>();
6         this.funcionarios.add(new Funcionario("Marcelo Martins"));
7         this.funcionarios.add(new Funcionario("Rafael Cosentino"));
8         this.funcionarios.add(new Funcionario("Jonas Hirata"));
9     }
10
11    public Funcionario acquire() {
12        if(this.funcionarios.size() > 0) {
13            return this.funcionarios.remove(0);
14        }
15        else {
16            return null;
17        }
18    }
19
20    public void release(Funcionario funcionario) {
21        this.funcionarios.add(funcionario);
22    }
23 }
```

Código Java 2.75: FuncionarioPool.java

- 36 Teste a classe **FuncionarioPool**.

```

1 public class TestaFuncionarioPool {
2     public static void main(String[] args) {
3         Pool<Funcionario> funcionarioPool = new FuncionarioPool();
4         Funcionario funcionario = funcionarioPool.acquire();
5         while (funcionario != null) {
6             System.out.println(funcionario.getNome());
7             funcionario = funcionarioPool.acquire();
8         }
9     }
10 }
```

Código Java 2.76: TestaFuncionarioPool.java



PADRÕES ESTRUTURAIS

As interações entre os objetos de um sistema podem gerar fortes dependências entre esses elementos. Essas dependências aumentam a complexidade das eventuais alterações no funcionamento do sistema. Consequentemente, o custo de manutenção aumenta. Mostraremos alguns padrões de projeto que diminuem o acoplamento entre os objetos de um sistema orientado a objetos.

Veja abaixo um resumo do objetivo de cada padrão estrutural.

Adapter Permitir que um objeto seja substituído por outro que, apesar de realizar a mesma tarefa, possui uma interface diferente.

Bridge Separar uma abstração de sua representação, de forma que ambos possam variar e produzir tipos de objetos diferentes.

Composite Agrupar objetos que fazem parte de uma relação parte-todo de forma a tratá-los sem distinção.

Decorator Adicionar funcionalidade a um objeto dinamicamente.

Facade Prover uma interface simplificada para a utilização de várias interfaces de um subsistema.

Front Controller Centralizar todas as requisições a uma aplicação Web.

Flyweight Compartilhar, de forma eficiente, objetos que são usados em grande quantidade.

Proxy Controlar as chamadas a um objeto através de outro objeto de mesma interface.



Adapter

Objetivo: Permitir que um objeto seja substituído por outro que, apesar de realizar a mesma tarefa, possui uma interface diferente.

No Brasil, mais de dez tipos de tomadas e plugues eram comercializados. Em 2007, um novo padrão para os plugues e tomadas foi definido pela ABNT. Em 2011, tornou-se permitida somente a venda de produtos que seguiam o novo padrão de dois ou três pinos redondos.

No entanto, muitos consumidores ainda não estão preparados para o novo padrão e possuem tomadas incompatíveis principalmente com os plugues de três pinos.

Por outro lado, há também os casos em que a tomada segue o novo padrão mas o plugue do aparelho não o segue. Esse pode ser o caso de aparelhos adquiridos antes da nova lei entrar em vigor ou adquiridos fora do Brasil.

Nesses casos, uma solução é usar um adaptador. Veja a figura abaixo.

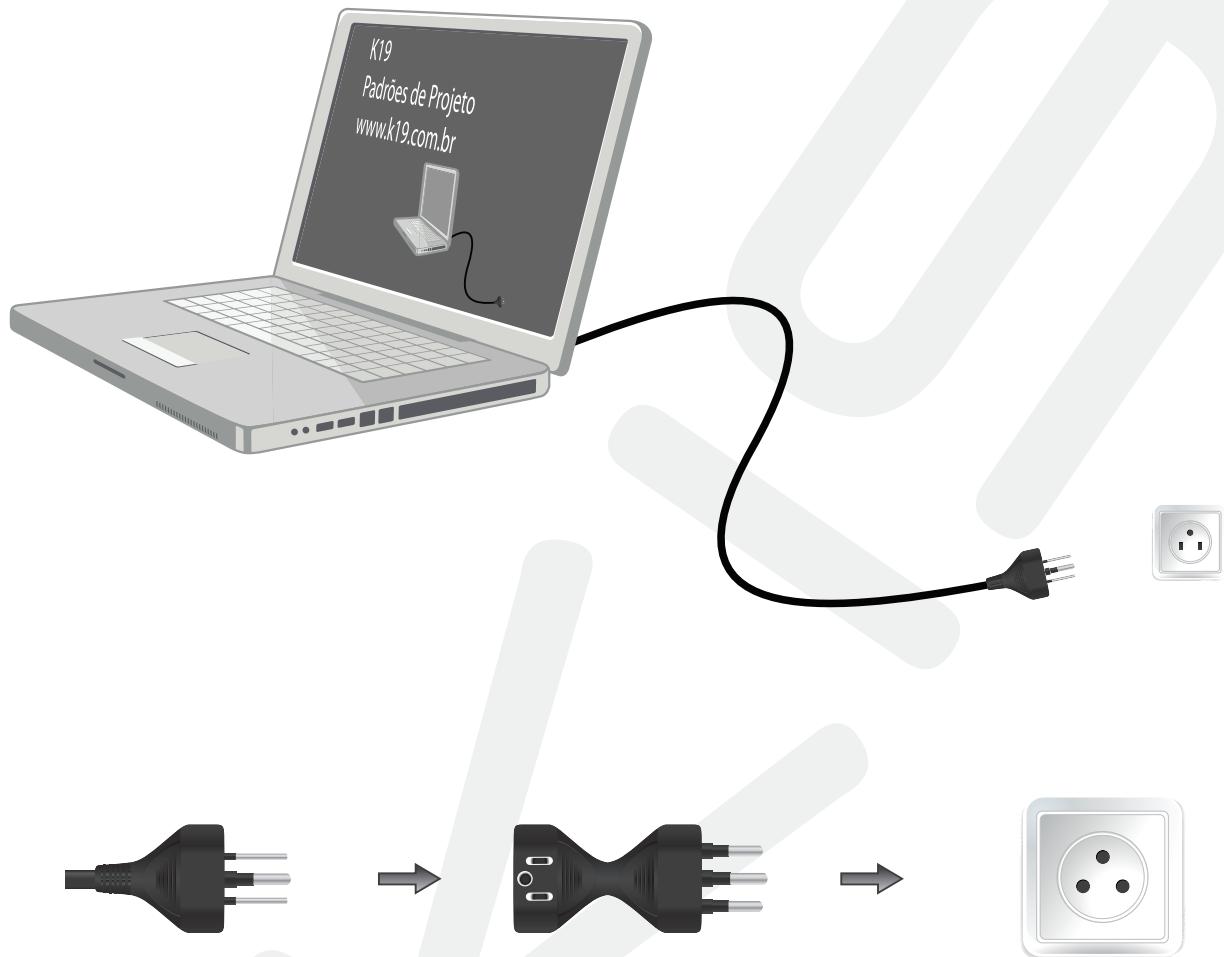


Figura 3.1: Plugue do notebook incompatível com o novo padrão de tomada

Essa é a essência do padrão Adapter.

Exemplo prático

Estamos realizando manutenção no sistema de gerenciamento de uma determinada empresa. O controle de ponto desse sistema possui diversas limitações. Essas limitações causam muitos prejuízos. Principalmente, prejuízos financeiros.

Uma empresa parceira implementou uma biblioteca Java para controlar a entrada e saída dos funcionários. Essa biblioteca não possui as limitações que existem hoje no sistema que estamos realizando manutenção. Os diretores decidiram que a melhor estratégia seria adquirir essa biblioteca e implantá-la no sistema.

Para implantar essa biblioteca, teremos que substituir as classes que atualmente cuidam do controle de ponto pelas classes dessa biblioteca. A complexidade dessa substituição é alta pois os métodos das classes antigas não são compatíveis com os métodos das classes novas. Em outras palavras, as interfaces são diferentes.

Para tentar minimizar o impacto dessa substituição no código do sistema, podemos definir classes intermediárias para adaptar as chamadas às classes da biblioteca que foi adquirida.

Por exemplo, atualmente, a entrada de um funcionário é registrada da seguinte maneira:

```
1 ControleDePonto controleDePonto = new ControleDePonto();
2 Funcionario funcionario = ...
3 controleDePonto.registraEntrada(funcionario);
```

Código Java 3.1: Registrando a entrada de um funcionário - atualmente

Utilizando as classes da nova biblioteca, a entrada de um funcionário deveria ser registrada assim:

```
1 ControleDePontoNovo controleDePontoNovo = ...
2 Funcionario funcionario = ...
3
4 // true indica entrada e false indica saída
5 controleDePontoNovo.registra(funcionario.getCodigo(), true);
```

Código Java 3.2: Registrando a entrada de um funcionário - com a nova biblioteca

Para diminuir o impacto no código do sistema, podemos criar o seguinte adaptador:

```
1 public class ControleDePontoAdapter extends ControleDePonto {
2     private ControleDePontoNovo controleDePontoNovo;
3
4     public void registraEntrada(Funcionario funcionario) {
5         this.controleDePontoNovo.registra(funcionario.getCodigo(), true);
6     }
7 }
```

Código Java 3.3: ControleDePontoAdapter.java

Podemos utilizar o adaptador como se estivéssemos utilizando o controle de ponto antigo.

```
1 ControleDePonto controleDePonto = new ControleDePontoAdapter();
2 Funcionario funcionario = ...
3 controleDePonto.registraEntrada(funcionario);
```

Código Java 3.4: Registrando a entrada de um funcionário - adaptador

Dessa forma, o código do sistema atual deve ser modificado apenas no trecho em que a classe responsável pelo controle de ponto é instanciada.

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

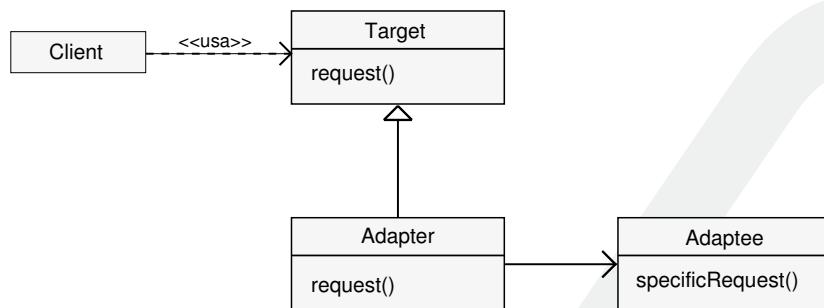


Figura 3.2: UML do padrão Adapter

Os personagens desse padrão são:

Target (ControleDePonto)

Define a interface utilizada pelo Client.

Adaptee (ControleDePontoNovo)

Classe que define o novo objeto a ser utilizado.

Adapter (ControleDePontoAdapter)

Classe que implementa a interface definida pelo Target e adapta as chamadas do Client para o Adaptee.

Client

Interage com os objetos através da interface definida por Target.



Exercícios de Fixação

- Crie um projeto chamado **Adapter**.

- Defina uma classe **Funcionario**.

```

1 public class Funcionario {
2     private String nome;
3
4     public Funcionario(String nome) {
5         this.nome = nome;
6     }
7
8     public String getNome() {
9         return nome;
10    }
11 }
  
```

Código Java 3.5: Funcionario.java

- Defina uma classe **ControleDePonto**.

```

1 public class ControleDePonto {
2     public void registraEntrada(Funcionario f) {
3         Calendar calendar = Calendar.getInstance();
4         SimpleDateFormat simpleDateFormat = new SimpleDateFormat(
5             "dd/MM/yyyy H:m:s");
6         String format = simpleDateFormat.format(calendar.getTime());
7         System.out.println("Entrada: " + f.getNome() + " às " + format);
8     }
9
10    public void registraSaida(Funcionario f) {
11        Calendar calendar = Calendar.getInstance();
12        SimpleDateFormat simpleDateFormat = new SimpleDateFormat(
13            "dd/MM/yyyy H:m:s");
14        String format = simpleDateFormat.format(calendar.getTime());
15        System.out.println("Saída: " + f.getNome() + " às " + format);
16    }
17 }
```

Código Java 3.6: ControleDePonto.java

4 Defina uma classe para testar o controle de ponto.

```

1 public class TesteControleDePonto {
2     public static void main(String[] args) throws InterruptedException {
3         ControleDePonto controleDePonto = new ControleDePonto();
4         Funcionario funcionario = new Funcionario("Marcelo Martins");
5         controleDePonto.registraEntrada(funcionario);
6         Thread.sleep(3000);
7         controleDePonto.registraSaida(funcionario);
8     }
9 }
```

Código Java 3.7: TesteControleDePonto.java

5 Defina uma classe **ControleDePontoNovo**.

```

1 public class ControleDePontoNovo {
2     public void registra(Funcionario f, boolean entrada) {
3         Calendar calendar = Calendar.getInstance();
4         SimpleDateFormat simpleDateFormat = new SimpleDateFormat(
5             "dd/MM/yyyy H:m:s");
6         String format = simpleDateFormat.format(calendar.getTime());
7
8         if (entrada == true) {
9             System.out.println("Entrada: " + f.getNome() + " às " + format);
10        } else {
11            System.out.println("Saída: " + f.getNome() + " às " + format);
12        }
13    }
14 }
```

Código Java 3.8: ControleDePontoNovo.java

6 Defina uma classe adaptador para a nova classe **ControleDePontoNovo**.

```

1 public class ControleDePontoAdapter extends ControleDePonto {
2     private ControleDePontoNovo controleDePontoNovo;
3
4     public ControleDePontoAdapter() {
5         this.controleDePontoNovo = new ControleDePontoNovo();
```

```

6  }
7
8  public void registraEntrada(Funcionario f) {
9      this.controleDePontoNovo.registra(f, true);
10 }
11
12 public void registraSaida(Funcionario f) {
13     this.controleDePontoNovo.registra(f, false);
14 }
15 }
```

Código Java 3.9: ControleDePontoAdapter.java

- 7 Altere a classe de teste do controle de ponto para utilizar o adapter.

```

1 public class TesteControleDePonto {
2     public static void main(String[] args) throws InterruptedException {
3         ControleDePonto controleDePonto = new ControleDePontoAdapter();
4         Funcionario funcionario = new Funcionario("Marcelo Martins");
5         controleDePonto.registraEntrada(funcionario);
6         Thread.sleep(3000);
7         controleDePonto.registraSaida(funcionario);
8     }
9 }
```

Código Java 3.10: TesteControleDePonto.java



Bridge

Objetivo: Separar uma abstração de sua representação, de forma que ambos possam variar e produzir tipos de objetos diferentes.

Hoje em dia, os celulares utilizam cartões SIM (*subscriber identity module*). Entre outros dados, esses cartões armazenam o número do telefone, a chave de autenticação do cliente e um identificador da rede a ser usada na comunicação.

Antes de ser capaz de fazer ligações, o celular precisa passar pelo processo de autenticação. Nesse processo, o celular troca informações com a rede móvel a fim de verificar a identidade do usuário.

Além de armazenar toda informação necessária para essa autenticação, o cartão SIM é quem de fato executa os algoritmos de criptografia usados nesse processo, e não o aparelho celular. Uma vez que a autenticação é feita, o celular torna-se capaz de fazer ligações.

A maioria dos celulares disponíveis hoje em dia é capaz de lidar com diversas operadoras, basta apenas que o cartão SIM do celular seja substituído. Essa é uma das vantagens dessa tecnologia.

Se o seu aparelho celular estiver danificado ou sem bateria, você pode simplesmente remover o cartão SIM do aparelho e colocá-lo em outro. Por outro lado, se você viajar para outro país, por exemplo, você pode levar o seu aparelho celular e adquirir um cartão SIM de uma operadora local.

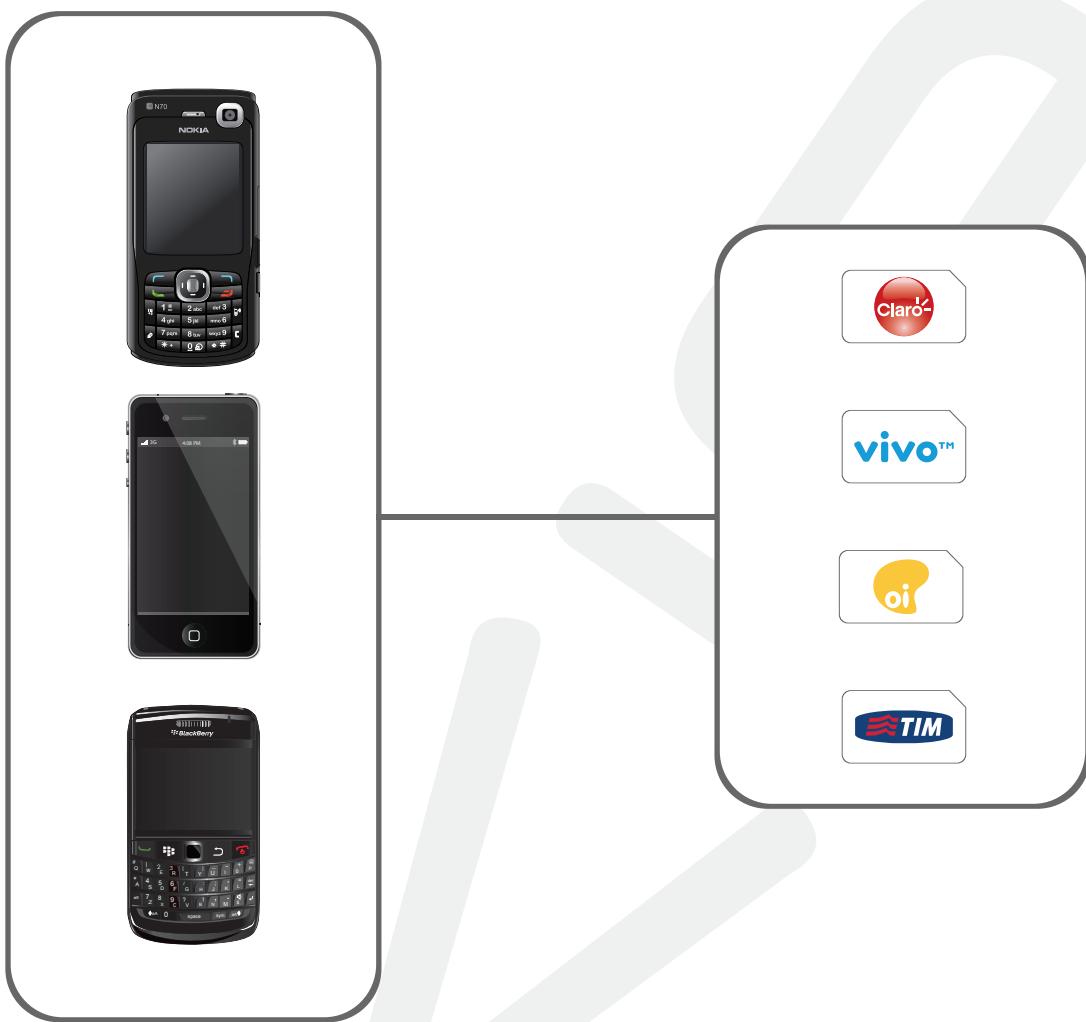


Figura 3.3: Diversas combinações de celular/operadora

A possibilidade de combinar os cartões SIM e os aparelhos celulares de forma independente é a característica principal proposta pelo padrão Bridge.

Exemplo prático

Estamos desenvolvendo um sistema que deve gerar diversos tipos de documentos (recibos, atestados, comunicados, etc) em diversos formatos de arquivos (txt, html, pdf, etc).

Podemos definir uma interface para padronizar os diversos tipos de documentos que o nosso sistema pode suportar.

```

1 public interface Documento {
2     void geraArquivo();
3 }
```

Código Java 3.11: Documento.java

Com a interface criada, podemos definir tipos específicos de documentos.

```

1 public class Recibo implements Documento {
2     private double valor;
3
4     public void geraArquivo() {
5         // cria um arquivo com os dados desse recibo em um formato suportado
6     }
7 }
```

Código Java 3.12: *Recibo.java*

Dessa forma, a lógica dos documentos seria definida nas classes que implementam a interface Documento. Para dividir melhor as responsabilidades, podemos implementar a lógica dos formatos de arquivos que o sistema suportará em outras classes. Inclusive, podemos definir uma interface para padronizar essas classes.

```

1 public interface GeradorDeArquivo {
2     public void gera(String conteudo);
3 }
```

Código Java 3.13: *GeradorDeArquivo.java*

```

1 public class GeradorDeArquivoTXT implements GeradorDeArquivo {
2     public void gera(String conteudo) {
3         // processo para gerar um arquivo txt com o conteúdo do parâmetro
4     }
5 }
```

Código Java 3.14: *GeradorDeArquivoTXT.java*

```

1 public class GeradorDeArquivoHTML implements GeradorDeArquivo {
2     public void gera(String conteudo) {
3         // processo para gerar um arquivo HTML com o conteúdo do parâmetro
4     }
5 }
```

Código Java 3.15: *GeradorDeArquivoHTML.java*

Por fim, os documentos devem ser associados aos geradores de arquivos. Isso pode ser realizado através de construtores nas classes específicas de documentos.

```

1 public class Recibo implements Documento {
2     private double valor;
3
4     private GeradorDeArquivo gerador;
5
6     public Recibo(GeradorDeArquivo gerador) {
7         this.gerador = gerador;
8     }
9
10    public void geraArquivo() {
11        String conteudoDesseRecibo = ...
12        this.gerador.gera(conteudoDesseRecibo)
13    }
14 }
```

Código Java 3.16: *Recibo.java*

Agora, podemos exportar um recibo em txt da seguinte forma:

```

1 GeradorDeArquivoTXT gerador = new GeradorDeArquivoTXT();
2 
```

```

3 Recibo recibo = new Recibo(gerador);
4
5 recibo.geraArquivo();

```

Código Java 3.17: Exportando um recibo em txt

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

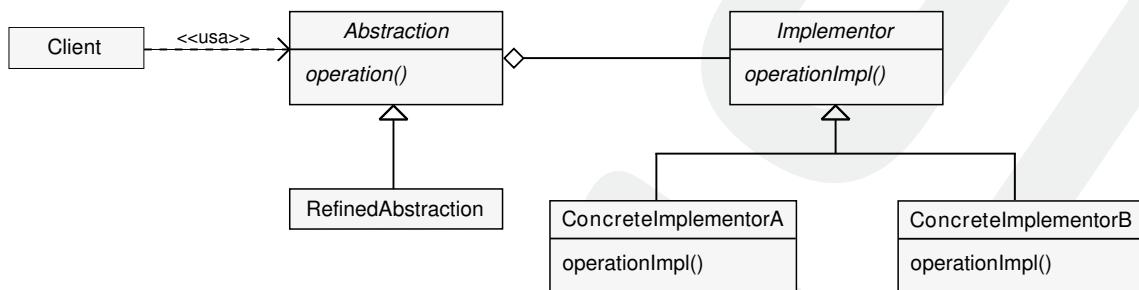


Figura 3.4: UML do padrão Bridge

Os personagens desse padrão são:

Abstraction (Documento)

Define a interface de um determinado tipo de objeto.

RefinedAbstraction (Recibo)

Uma implementação particular do Abstraction que delega a um Implementor a realização de determinadas tarefas.

Implementor (GeradorDeArquivo)

Define a interface dos objetos que serão acionados pelos Abstractions.

ConcreteImplementor (GeradorDeArquivoTXT, GeradorDeArquivoHTML)

uma implementação específica do Implementor

Client

Interage com as Abstractions.



Exercícios de Fixação

- 8 Crie um projeto chamado **Bridge**.

- 9 Crie uma interface **Documento**.

```

1 public interface Documento {
2     void geraArquivo();
3 }

```

Código Java 3.18: Documento.java

- 10 Crie uma classe **Recibo** que gera um arquivo txt.

```

1 public class Recibo implements Documento {
2     private String emissor;
3     private String favorecido;
4     private double valor;
5
6     public Recibo(String emissor, String favorecido, double valor) {
7         this.emissor = emissor;
8         this.favorecido = favorecido;
9         this.valor = valor;
10    }
11
12    public void geraArquivo() {
13        try {
14            PrintStream saida = new PrintStream("recibo.txt");
15            saida.println("Recibo: ");
16            saida.println("Empresa: " + this.emissor);
17            saida.println("Cliente: " + this.favorecido);
18            saida.println("Valor: " + this.valor);
19        } catch (FileNotFoundException e) {
20            e.printStackTrace();
21        }
22    }
23 }
```

Código Java 3.19: Recibo.java

- 11 Teste a classe **Recibo**.

```

1 public class TesteRecibo {
2     public static void main(String[] args) {
3         Recibo recibo = new Recibo("K19 Treinamentos", "Marcelo Martins", 1000);
4         recibo.geraArquivo();
5     }
6 }
```

Código Java 3.20: TesteRecibo.java

- 12 A classe **Recibo** somente gera arquivo no formato txt. Para suportar mais formatos, podemos definir a lógica de gerar arquivos em diversos formatos em outras classes. Vamos padronizar estas classes através de uma interface.

```

1 public interface GeradorDeArquivo {
2     public void gera(String conteudo);
3 }
```

Código Java 3.21: GeradorDeArquivo.java

- 13 Crie uma classe **GeradorDeArquivoTXT** que implementa a interface **GeradorDeArquivo**.

```

1 public class GeradorDeArquivoTXT implements GeradorDeArquivo {
2     public void gera(String conteudo) {
```

```

3   try {
4     PrintStream saida = new PrintStream("arquivo.txt");
5     saida.println(conteudo);
6     saida.close();
7   } catch (FileNotFoundException e) {
8     e.printStackTrace();
9   }
10 }
11 }
```

Código Java 3.22: GeradorDeArquivoTXT.java

- 14 Associe os geradores de arquivos aos documentos. Altere a classe **Recibo** para receber um gerador de arquivo como parâmetro no construtor.

```

1 public class Recibo implements Documento {
2   private String emissor;
3   private String favorecido;
4   private double valor;
5   private GeradorDeArquivo geradorDeArquivo;
6
7   public Recibo(String emissor, String favorecido, double valor,
8     GeradorDeArquivo geradorDeArquivo) {
9     this.emissor = emissor;
10    this.favorecido = favorecido;
11    this.valor = valor;
12    this.geradorDeArquivo = geradorDeArquivo;
13  }
14
15  public void geraArquivo() {
16    StringBuffer buffer = new StringBuffer();
17    buffer.append("Recibo: ");
18    buffer.append("\n");
19    buffer.append("Empresa: " + this.emissor);
20    buffer.append("\n");
21    buffer.append("Cliente: " + this.favorecido);
22    buffer.append("\n");
23    buffer.append("Valor: " + this.valor);
24    buffer.append("\n");
25    this.geradorDeArquivo.gera(buffer.toString());
26  }
27 }
```

Código Java 3.23: Recibo.java

- 15 Teste a classe a classe **Recibo** e associe com o gerador de arquivo txt.

```

1 public class TesteRecibo {
2   public static void main(String[] args) {
3     GeradorDeArquivo geradorDeArquivoTXT = new GeradorDeArquivoTXT();
4     Recibo recibo = new Recibo("K19 Treinamentos", "Marcelo Martins", 1000,
5       geradorDeArquivoTXT);
6     recibo.geraArquivo();
7   }
8 }
```

Código Java 3.24: TesteRecibo.java



Composite

Objetivo: Agrupar objetos que fazem parte de uma relação parte-todo de forma a tratá-los sem distinção.

Exemplo prático

Suponha que estamos desenvolvendo um sistema para calcular um caminho entre quaisquer dois pontos do mundo. Um caminho pode ser percorrido de diversas maneiras: à pé, de carro, de ônibus, de trem, de avião, de navio, etc.

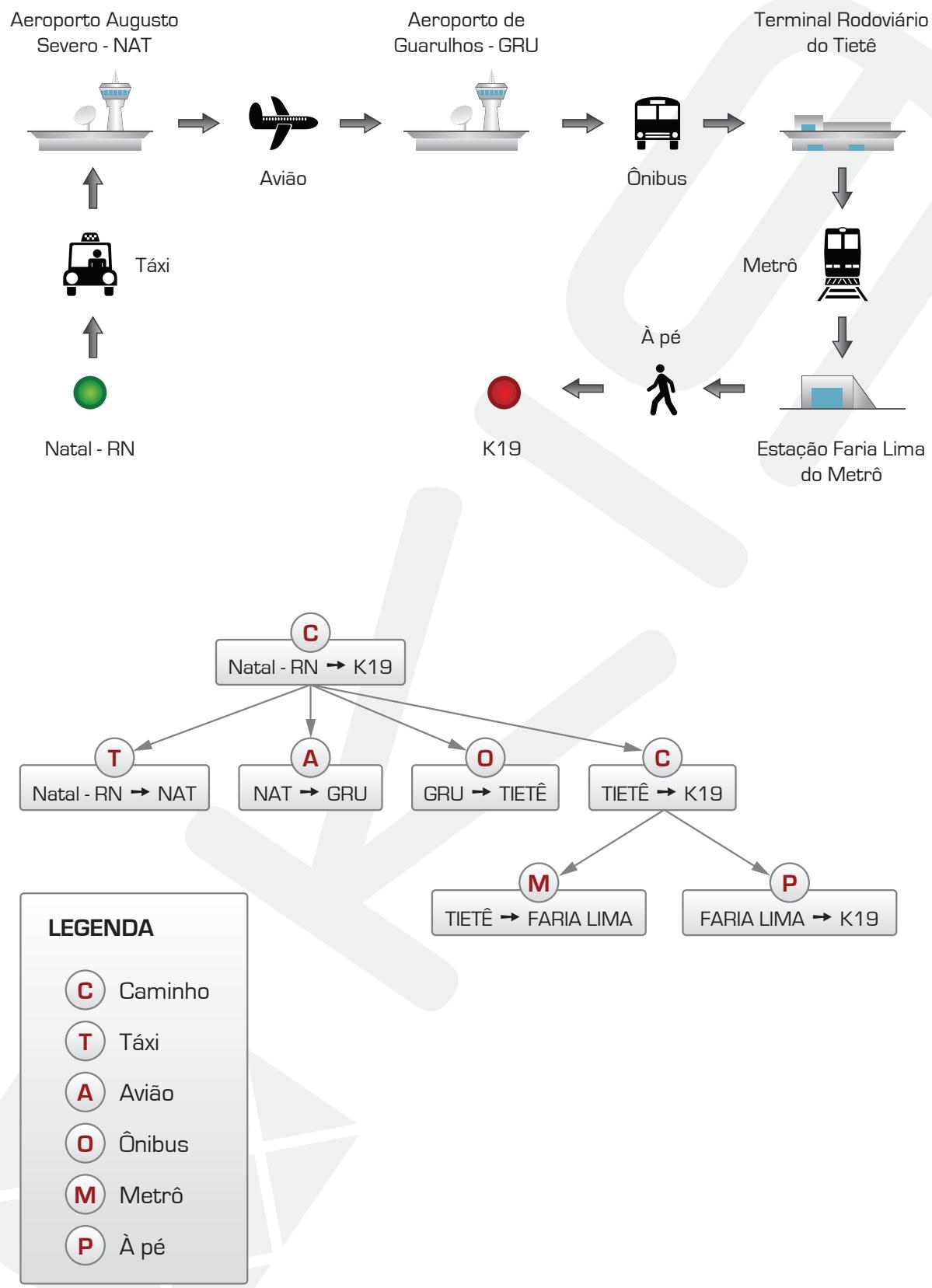


Figura 3.5: Viagem de Natal-RN para São Paulo

O sistema deve apresentar graficamente para os usuários as rotas que forem calculadas. Cada tipo de trecho deve ser apresentado de uma maneira específica. Por exemplo, se o trecho for de caminhada então deve aparecer na impressão da rota a ilustração de uma pessoa andando.

Cada tipo de trecho pode ser implementado por uma classe e seria interessante definir uma interface para padronizá-las.

```
1 public interface Trecho {
2     void imprime();
3 }
```

Código Java 3.25: Trecho.java

```
1 public class TrechoAndando implements Trecho {
2     // atributos e métodos
3
4     public void imprime() {
5         // imprime na tela as informações desse trecho andando.
6     }
7 }
```

Código Java 3.26: TrechoAndando.java

```
1 public class TrechoDeCarro implements Trecho {
2     // atributos e métodos
3
4     public void imprime() {
5         // imprime na tela as informações desse trecho de carro.
6     }
7 }
```

Código Java 3.27: TrechoDeCarro.java

O próprio caminho entre dois pontos pode ser considerado um trecho. Basicamente, um caminho é um trecho composto por outros trechos.

```
1 public class Caminho implements Trecho {
2     private List<Trecho> trechos = new ArrayList<Trecho>();
3
4     public void adiciona(Trecho trecho) {
5         this.trechos.add(trecho);
6     }
7
8     public void remove(Trecho trecho) {
9         this.trechos.remove(trecho);
10    }
11
12    public void imprime() {
13        // imprime na tela as informações desse caminho.
14    }
15 }
```

Código Java 3.28: Caminho.java

O processo de construção de um caminho envolveria a criação ou a recuperação de trechos de quaisquer tipos para compor um trecho maior.

```
1 Trecho trecho1 = ...
2 Trecho trecho2 = ...
3 Trecho trecho3 = ...
4
```

```

5 | Caminho caminho1 = new Caminho();
6 | caminho1.adiciona(trecho1);
7 | caminho1.adiciona(trecho2);
8 |
9 | Caminho caminho2 = new Caminho();
10 | caminho2.adiciona(caminho1);
11 | caminho2.adiciona(trecho3);

```

Código Java 3.29: Criando um caminho

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

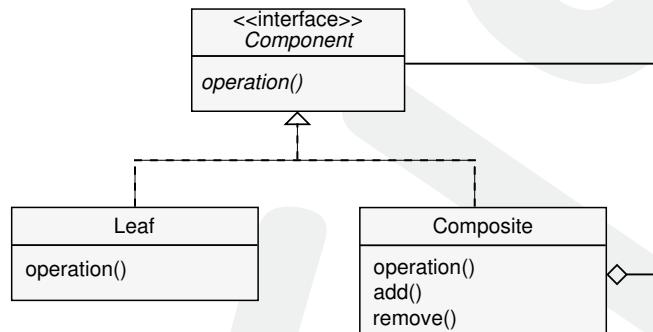


Figura 3.6: UML do padrão Composite

Os personagens desse padrão são:

Component (Trecho)

Interface que define os elementos da composição.

Composite (Caminho)

Define os Components que são formados por outros Components.

Leaf (TrechoAndando, TrechoDeCarro)

Define os elementos básicos da composição, isto é, aqueles que não são formados por outros Components.



Exercícios de Fixação

16 Crie um projeto chamado **Composite**.

17 Defina uma interface **Trecho**.

```

1 | public interface Trecho {
2 |     void imprime();
3 | }

```

Código Java 3.30: Trecho.java

- 18** Defina as classes que implementam a interface **Trecho**.

```

1 public class TrechoAndando implements Trecho {
2     private String direcao;
3     private double distancia;
4
5     public TrechoAndando(String direcao, double distancia) {
6         this.direcao = direcao;
7         this.distancia = distancia;
8     }
9
10    public void imprime() {
11        System.out.println("Vá Andando: ");
12        System.out.println(this.direcao);
13        System.out.println("A distância percorrida será de: " + this.distancia + " metros");
14    }
15 }
```

Código Java 3.31: *TrechoAndando.java*

```

1 public class TrechoDeCarro implements Trecho {
2     private String direcao;
3     private double distancia;
4
5     public TrechoDeCarro(String direcao, double distancia) {
6         this.direcao = direcao;
7         this.distancia = distancia;
8     }
9
10    public void imprime() {
11        System.out.println("Vá de carro:");
12        System.out.println(this.direcao);
13        System.out.println("A distância percorrida será de: " + this.distancia + " metros");
14    }
15 }
```

Código Java 3.32: *TrechoDeCarro.java*

- 19** Defina uma classe **Caminho** que é um **Trecho** e será composto por um ou mais trechos.

```

1 public class Caminho implements Trecho {
2     private List<Trecho> trechos;
3
4     public Caminho() {
5         this.trechos = new ArrayList<Trecho>();
6     }
7
8     public void adiciona(Trecho trecho) {
9         this.trechos.add(trecho);
10    }
11
12    public void remove(Trecho trecho) {
13        this.trechos.remove(trecho);
14    }
15
16    public void imprime() {
17        for (Trecho trecho : this.trechos) {
18            trecho.imprime();
19        }
20    }
21 }
```

Código Java 3.33: *Caminho.java*

- 20 Crie uma classe para testar e criar um caminho que é composto por mais de um trecho.

```

1 public class TestaCaminho {
2     public static void main(String[] args) {
3         Trecho trecho1 = new TrechoAndando(
4             "Vá até o cruzamento da Av. Rebouças com a Av.Brigadeiro Faria Lima",
5             500);
6         Trecho trecho2 = new TrechoDeCarro(
7             "Vá até o cruzamento da Av. Brigadeiro Faria Lima com a Av.Cidade Jardim",
8             1500);
9         Trecho trecho3 = new TrechoDeCarro(
10            "Vire a direita na Marginal Pinheiros", 500);
11
12         Caminho caminho1 = new Caminho();
13         caminho1.adiciona(trecho1);
14         caminho1.adiciona(trecho2);
15
16         System.out.println("Caminho 1 : ");
17         caminho1.imprime();
18
19         Caminho caminho2 = new Caminho();
20         caminho2.adiciona(caminho1);
21         caminho2.adiciona(trecho3);
22         System.out.println("-----");
23         System.out.println("Caminho 2: ");
24         caminho2.imprime();
25     }
26 }
```

Código Java 3.34: TesteCaminho.java



Decorator

Objetivo: Adicionar funcionalidade a um objeto dinamicamente.

Considere um mapa de ruas digital, como o Google Maps, por exemplo. É natural que o mapa contenha os nomes das ruas. Contudo, o mapa também poderia exibir diversas outras informações (como, por exemplo, relevo, indicadores de estabelecimentos de comércio e serviço), bem como oferecer opções para encontrar caminhos entre dois pontos e traçar rotas.

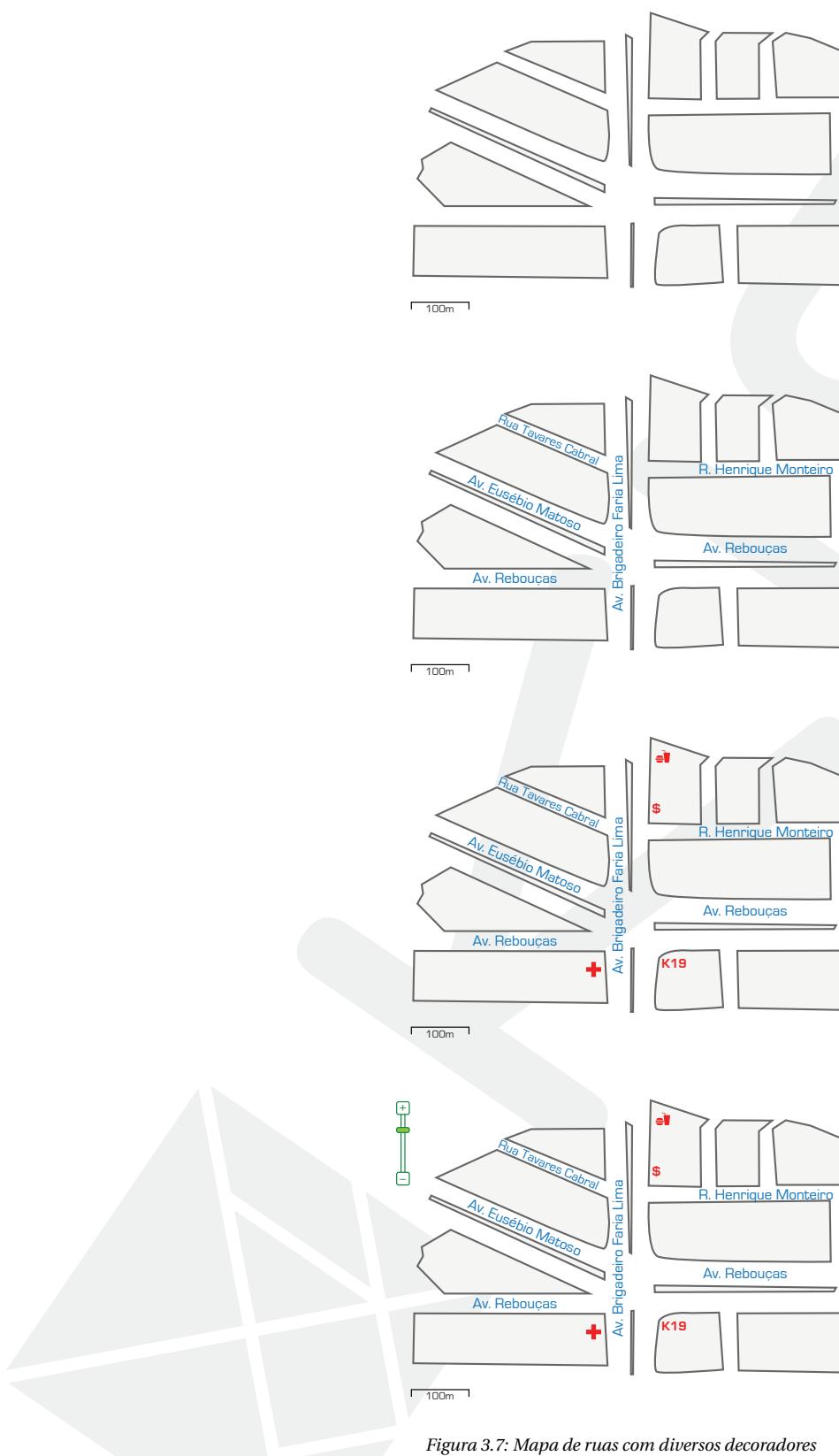


Figura 3.7: Mapa de ruas com diversos decoradores

Essas opções adicionais são o que chamamos de decorações.

Exemplo prático

Como exemplo prático do padrão Factory Method, consideramos um sistema de envio de mensagens. Nesse exemplo, definimos uma interface para padronizar os emissores.

```
1 public interface Emissor {
2     void envia(String mensagem);
3 }
```

Código Java 3.35: Emissor.java

Um possível emissor, poderia ser implementado mais ou menos assim:

```
1 public class EmissorBasico implements Emissor {
2     public void envia(String mensagem) {
3         System.out.println("Enviando uma mensagem: ");
4         System.out.println(mensagem);
5     }
6 }
```

Código Java 3.36: EmissorBasico.java

Agora, suponha que estejamos interessados em adicionar algumas funcionalidades no processo de envio de mensagem. Tais funcionalidades incluem criptografia e compressão das mensagens.

Para não alterar as classes que definem os emissores, cada funcionalidade adicional (decoração) será implementada por um novo objeto (decorador).

Quando queremos enviar uma mensagem, não podemos chamar diretamente os emissores, pois as funcionalidades adicionais não serão executadas. Portanto, devemos entregar a mensagem a um decorador, que executará a tarefa para a qual foi concebido. O decorador, por sua vez, terá também a responsabilidade de repassar a mensagem a um emissor para que ela seja enviada. Dessa forma, todo decorador deve possuir um emissor.

O código atual utiliza a interface dos emissores para enviar mensagens. Para não afetar esse código, os decoradores devem seguir a mesma interface dos emissores. Assim, quem envia uma mensagem através de um emissor, não sabe se este emissor é um decorador. Note que, dessa forma, decoradores podem ser encadeados.

Vamos então definir uma classe abstrata para representar os decoradores de emissores. Nesta classe, definiremos que todos os decoradores possuirão um emissor.

```
1 public abstract class EmissorDecorator implements Emissor {
2     private Emissor emissor;
3
4     public EmissorDecorator(Emissor emissor) {
5         this.emissor = emissor;
6     }
7
8     public abstract void envia(String mensagem);
9
10    public Emissor getEmissor() {
11        return this.emissor;
12    }
13 }
```

Código Java 3.37: EmissorDecorator.java

Agora, podemos implementar alguns decoradores.

```

1 public class EmissorDecoratorComCriptografia extends EmissorDecorator {
2     public EmissorDecoratorComCriptografia(Emissor emissor) {
3         super(emissor);
4     }
5
6     public void envia(String mensagem) {
7         System.out.println("Enviando mensagem criptografada: ");
8         this.getEmissor().envia(cryptografa(mensagem));
9     }
10
11    private String cryptografa(String mensagem) {
12        String mensagemCriptografada = ...
13        return mensagemCriptografada;
14    }
15 }
```

Código Java 3.38: EmissorDecoratorComCriptografia.java

```

1 public class EmissorDecoratorComCompressao extends EmissorDecorator {
2
3     public EmissorDecoratorComCompressao(Emissor emissor) {
4         super(emissor);
5     }
6
7     void envia(String mensagem) {
8         System.out.println("Enviando mensagem comprimida: ");
9         this.getEmissor().envia(comprime(mensagem));
10    }
11
12    private String compressa(String mensagem) {
13        String mensagemComprimida = ...
14        return mensagemComprimida;
15    }
16 }
```

Código Java 3.39: EmissorDecoratorComCompressao.java

Os decoradores poderiam então ser usados da seguinte forma:

```

1 String mensagem = ...
2
3 // Um emissor que apenas envia a mensagem
4 Emissor emissor = new EmissorBasico();
5 emissor.envia(mensagem);
6
7 // Emissor que envia mensagens criptografadas
8 emissor = new EmissorComCriptografia(new EmissorBasico());
9 emissor.envia(mensagem);
10
11 // Emissor que envia mensagens criptografadas e comprimidas
12 emissor = new EmissorComCriptografia(new EmissorComCompressao(new EmissorBasico()));
13 emissor.envia(mensagem);
```

Código Java 3.40: Emissores com diferentes funcionalidades inseridas dinamicamente

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

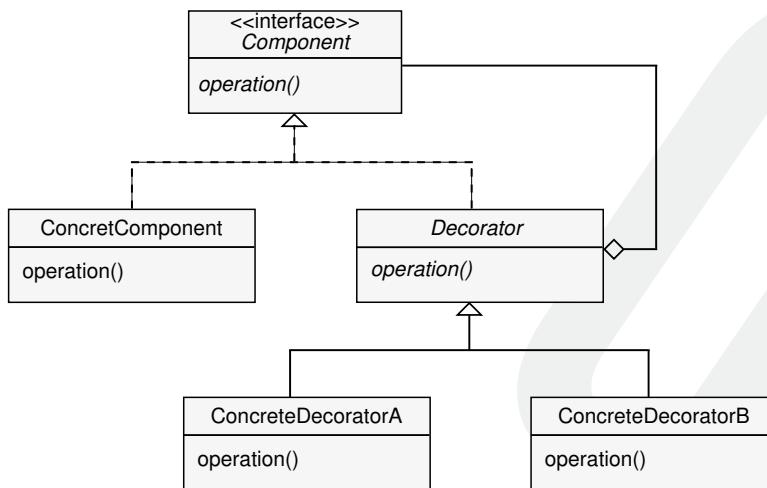


Figura 3.8: UML do padrão Decorator

Os personagens desse padrão são:

Component (Emissor)

Define a interface de objetos que possuem determinada tarefa.

ConcreteComponent (EmissorBasico)

Implementação particular do Component.

Decorator (EmissorDecorator)

Classe abstrata que mantém uma referência para um Component e será utilizada para padronizar os objetos decoradores.

ConcreteDecorator (EmissorDecoratorComCriptografia, EmissorDecoratorComCompressao)

Implementação de um Decorator.



Exercícios de Fixação

21 Crie um projeto chamado **Decorator**.

22 Defina uma interface **Emissor**.

```

1 public interface Emissor {
2     void envia(String mensagem);
3 }
  
```

Código Java 3.41: *Emissor.java*

23 Crie a classe **EmissorBasico**.

```

1 public class EmissorBasico implements Emissor {
2     public void envia(String mensagem) {
3         System.out.println("Enviando uma mensagem: ");
4         System.out.println(mensagem);
5     }
6 }
```

Código Java 3.42: EmissorBasico.java

- 24** Crie uma classe **EmissorDecorator** para modelar um decorador de emissores.

```

1 public abstract class EmissorDecorator implements Emissor {
2     private Emissor emissor;
3
4     public EmissorDecorator(Emissor emissor) {
5         this.emissor = emissor;
6     }
7
8     public abstract void envia(String mensagem);
9
10    public Emissor getEmissor() {
11        return this.emissor;
12    }
13 }
```

Código Java 3.43: EmissorDecorator.java

- 25** Crie um decorador que envia mensagens criptografadas e outro que envia mensagens comprimidas.

```

1 public class EmissorDecoratorComCriptografia extends EmissorDecorator {
2
3     public EmissorDecoratorComCriptografia(Emissor emissor) {
4         super(emissor);
5     }
6
7     void envia(String mensagem) {
8         System.out.println("Enviando mensagem criptografada: ");
9         this.getEmissor().envia(cryptograph(mensagem));
10    }
11
12    private String cryptograph(String mensagem) {
13        String mensagemCriptografada = new StringBuilder(mensagem).reverse().toString();
14        return mensagemCriptografada;
15    }
16 }
```

Código Java 3.44: EmissorDecoratorComCriptografia.java

```

1 public class EmissorDecoratorComCompressao extends EmissorDecorator {
2
3     public EmissorDecoratorComCompressao(Emissor emissor) {
4         super(emissor);
5     }
6
7     void envia(String mensagem) {
8         System.out.println("Enviando mensagem comprimida: ");
9         String mensagemComprimida;
10        try {
11            mensagemComprimida = compress(mensagem);
12        } catch (IOException e) {
13            mensagemComprimida = mensagem;
```

```

14     }
15     this.getEmissor().envia(mensagemComprimida);
16 }
17
18 private String comprime(String mensagem) throws IOException {
19     ByteArrayOutputStream out = new ByteArrayOutputStream();
20     DeflaterOutputStream dout = new DeflaterOutputStream(out, new Deflater());
21     dout.write(mensagem.getBytes());
22     dout.close();
23     return new String(out.toByteArray());
24 }
25 }
```

Código Java 3.45: EmissorDecoratorComCompressao.java

26 Teste os decoradores.

```

1 public class TesteEmissorDecorator {
2
3     public static void main(String[] args) {
4         String mensagem = "";
5
6         Emissor emissorCript = new EmissorComCriptografia(new EmissorBasico());
7         emissorCript.envia(mensagem);
8
9         Emissor emissorCompr = new EmissorComCompressao(new EmissorBasico());
10        emissorCompr.envia(mensagem);
11
12        Emissor emissorCriptCompr = new EmissorComCriptografia(new EmissorComCompressao(←
13            new EmissorBasico()));
14        emissorCriptCompr.envia(mensagem);
15    }
16 }
```

Código Java 3.46: TesteEmissorDecorator.java



Facade

Objetivo: Prover uma interface simplificada para a utilização de várias interfaces de um subsistema.

Considere uma pessoa planejando suas próximas férias de verão. Ela poderia comprar a passagem aérea, reservar o hotel e agendar passeios, tudo por conta própria.

Entretanto, tudo isso poderia ser muito trabalhoso. Em particular, ela precisaria pesquisar preços, comparar opções, reservar o hotel de acordo com as datas de chegada e saída de seu voo, etc.

É nesse ponto que entram as agências de viagens. Elas podem facilitar essa tarefa e trazer comodidade àquele que deseja viajar, atuando como um intermediário entre o cliente e as companhias aéreas, hotéis e empresas de passeio.

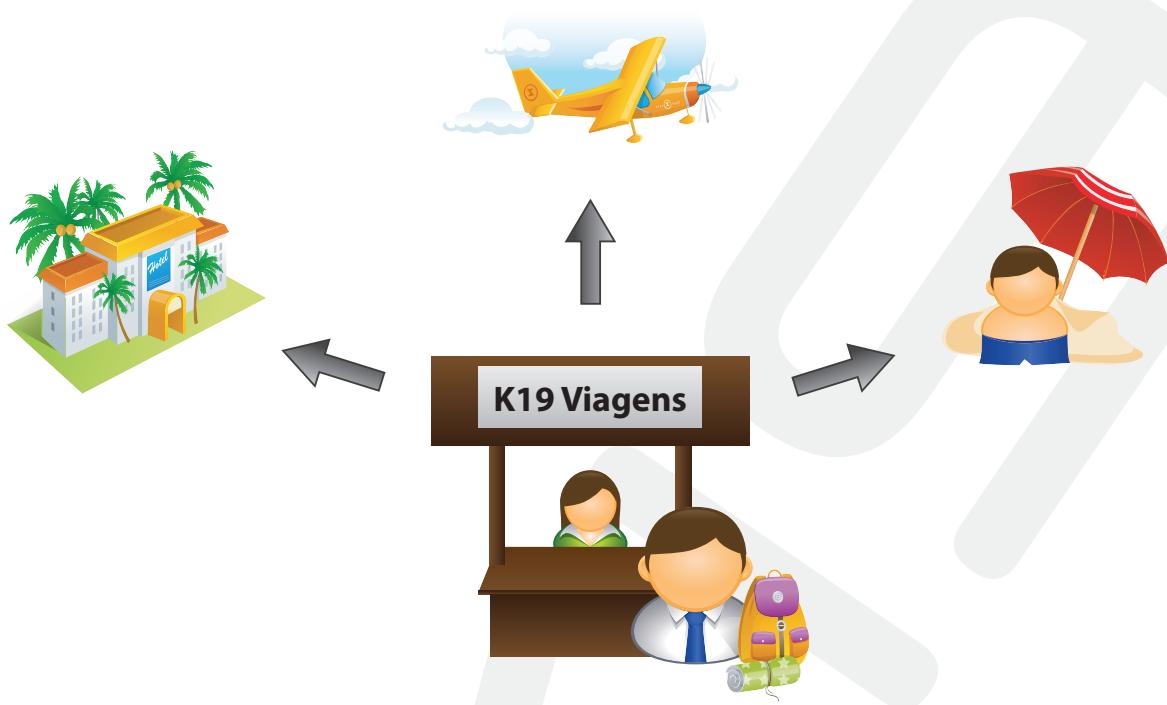


Figura 3.9: Pessoa comprando um pacote oferecido por uma agência de viagens

A ideia do padrão Facade é a mesma da agência de viagens, ou seja, simplificar a interação de um cliente com diversos sistemas.

Exemplo prático

Estamos melhorando um sistema que realiza todos os procedimentos que devem ser realizados após o registro de um pedido. Quando um pedido é realizado, o módulo que gerencia o estoque deve ser avisado para que o produto seja encaminhado ao endereço de entrega. O módulo financeiro deve ser avisado para que o processo de faturamento seja realizado. O módulo de pós venda também deve ser avisado para que contatos futuros sejam realizados com o cliente com o intuito de verificar a satisfação do mesmo com o produto obtido.

O sistema já está funcionando e realiza todos os processos decorrentes da realização de um novo pedido. Mas, queremos simplificar essa lógica encapsulando as chamadas aos módulos de estoque, financeiro e de pós venda.

```

1 Pedido p = ...
2 estoque.enviaProduto(p.getProduto(), p.getEnderecoDeEntrega(), p.getNotaFiscal());
3 financeiro.fatura(p.getCliente(), p.getNotaFiscal());
4 posVenda.agendaContato(p.getCliente(), p.getProduto());

```

Código Java 3.47: disparando os processos decorrentes de um novo pedido

A ideia aqui é criar uma classe que encapsula todos os processos que envolvem o acesso aos módulos de estoque, financeiro e de pós venda.

```

1 public class PedidoFacade {
2     private Estoque estoque;

```

```

3  private Financeiro financeiro;
4
5  private PosVenda posVenda;
6
7
8  public PedidoFacade(Estoque estoque, Financeiro financeiro, PosVenda posVenda) {
9      this.estoque = estoque;
10     this.financeiro = financeiro;
11     this.posVenda = posVenda;
12 }
13
14 public void registraPedido(Pedido p) {
15     this.estoque.enviaProduto(p.getProduto(), p.getEnderecoDeEntrega(), p.getNotaFiscal());
16     this.financeiro.fatura(p.getCliente(), p.getNotaFiscal());
17     this.posVenda.agendaContato(p.getCliente(), p.getProduto());
18 }
19
20 public void cancelaPedido(Pedido p) {
21     // logica para cancelar pedidos
22 }
23 }
```

Código Java 3.48: PedidoFacade.java

Agora, quando um pedido é realizado, não é mais necessário acessar diretamente diversos módulos diferentes, diminuindo assim a complexidade das operações.

```

1 Pedido p = ...
2 PedidoFacade pedidoFacade = ...
3 pedidoFacade.registraPedido(p);
```

Código Java 3.49: Registrando um pedido

Da mesma forma, se um pedido for cancelado, devemos acionar a fachada.

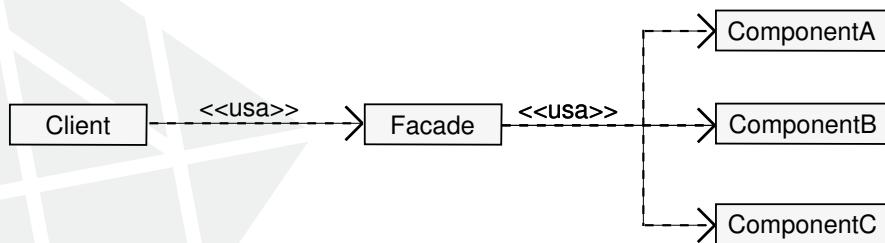
```

1 Pedido p = ...
2 PedidoFacade pedidoFacade = ...
3 pedidoFacade.cancelaPedido(p);
```

Código Java 3.50: Cancelando um pedido

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

*Figura 3.10: UML do padrão Facade*

Os personagens desse padrão são:

Facade (PedidoFacade) Classe intermediária que simplifica o acesso aos Component.

Client

Classe que usa os Component de forma indireta através do Facade.

Component (Estoque, Financeiro, PosVenda)

Classes que compõem o subsistema.

**Exercícios de Fixação**

- 27** Crie um projeto chamado **Facade**.

- 28** Defina as classes **Pedido**, **Estoque**, **Financeiro** e **PosVenda**.

```

1 public class Pedido {
2     private String produto;
3     private String cliente;
4     private String endereçoDeEntrega;
5
6     public Pedido(String produto, String cliente, String endereçoDeEntrega) {
7         this.produto = produto;
8         this.cliente = cliente;
9         this.endereçoDeEntrega = endereçoDeEntrega;
10    }
11
12    public String getProduto() {
13        return produto;
14    }
15
16    public String getCliente() {
17        return cliente;
18    }
19
20    public String getEndereçoDeEntrega() {
21        return endereçoDeEntrega;
22    }
23
24 }
```

Código Java 3.51: Pedido.java

```

1 import java.text.SimpleDateFormat;
2 import java.util.Calendar;
3
4 public class Estoque {
5     public void enviaProduto(String produto, String endereçoDeEntrega) {
6         Calendar calendar = Calendar.getInstance();
7         calendar.add(Calendar.DATE, 2);
8         SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");
9         String format = simpleDateFormat.format(calendar.getTime());
10
11         System.out.println("O produto " + produto
12             + " será entregue no endereço " + endereçoDeEntrega
13             + " até as 18h do dia " + format);
14     }
15 }
```

Código Java 3.52: Estoque.java

```
1 public class Financeiro {
```

```

1 public void fatura(String cliente, String produto) {
2     System.out.println("Fatura:");
3     System.out.println("Cliente: " + cliente);
4     System.out.println("Produto: " + produto);
5 }
6
7 }
```

Código Java 3.53: Financeiro.java

```

1 public class PosVenda {
2     public void agendaContato(String cliente, String produto) {
3         Calendar calendar = Calendar.getInstance();
4         calendar.add(Calendar.DATE, 30);
5         SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");
6         String format = simpleDateFormat.format(calendar.getTime());
7
8         System.out.println("Entrar em contato com " + cliente
9             + " sobre o produto " + produto + " no dia " + format);
10    }
11 }
```

Código Java 3.54: PosVenda.java

- 29** Crie uma classe **PedidoFacade** que encapsula o acesso aos módulos de estoque, financeiro e pós venda.

```

1 public class PedidoFacade {
2     private Estoque estoque;
3     private Financeiro financeiro;
4     private PosVenda posVenda;
5
6     public PedidoFacade(Estoque estoque, Financeiro financeiro, PosVenda posVenda) {
7         this.estoque = estoque;
8         this.financeiro = financeiro;
9         this.posVenda = posVenda;
10    }
11
12    public void registraPedido(Pedido p) {
13        this.estoque.enviaProduto(p.getProduto(), p.getEnderecoDeEntrega(), p.getNotaFiscal());
14        this.financeiro.fatura(p.getCliente(), p.getNotaFiscal());
15        this.posVenda.agendaContato(p.getCliente(), p.getProduto());
16    }
17 }
```

Código Java 3.55: PedidoFacade.java

- 30** Teste a classe **PedidoFacade**.

```

1 public class TestePedidoFacade {
2     public static void main(String[] args) {
3         Estoque estoque = new Estoque();
4         Financeiro financeiro = new Financeiro();
5         PosVenda posVenda = new PosVenda();
6         PedidoFacade facade = new PedidoFacade(estoque, financeiro, posVenda);
7         Pedido pedido = new Pedido("Notebook", "Rafael Cosentino",
8             "Av Brigadeiro Faria Lima, 1571, São Paulo, SP");
9         facade.registraPedido(pedido);
10    }
11 }
```

Código Java 3.56: TestePedidoFacade.java



Front Controller (não GoF)

Objetivo: Centralizar todas as requisições a uma aplicação Web.

Exemplo prático

Estamos desenvolvendo um framework MVC para o desenvolvimento de aplicações web em Java. O processamento de uma requisição HTTP segue os seguintes passos:

- Converter os dados enviados pelo navegador para os tipos de Java.
- Validar os valores convertidos.
- Acionar o processamento das regras de negócio.
- Acionar a montagem da tela de resposta.
- Enviar a tela para o usuário.

O nosso framework deve interceptar todas as requisições HTTP para realizar esses passos. Para interceptar as requisições HTTP, podemos implementar uma Servlet.

```

1 @WebServlet("/*")
2 public class FrontController extends HttpServlet {
3     public void service(HttpServletRequest req, HttpServletResponse res) {
4         // passo1
5         // passo2
6         // passo3
7         // passo4
8     }
9 }
```

Código Java 3.57: *FrontController.java*

As tarefas que devem ser realizadas em todas as requisições podem ser implementadas nessa Servlet. Por outro lado, as tarefas específicas devem ser delegadas para outra parte do framework ou para a aplicação.

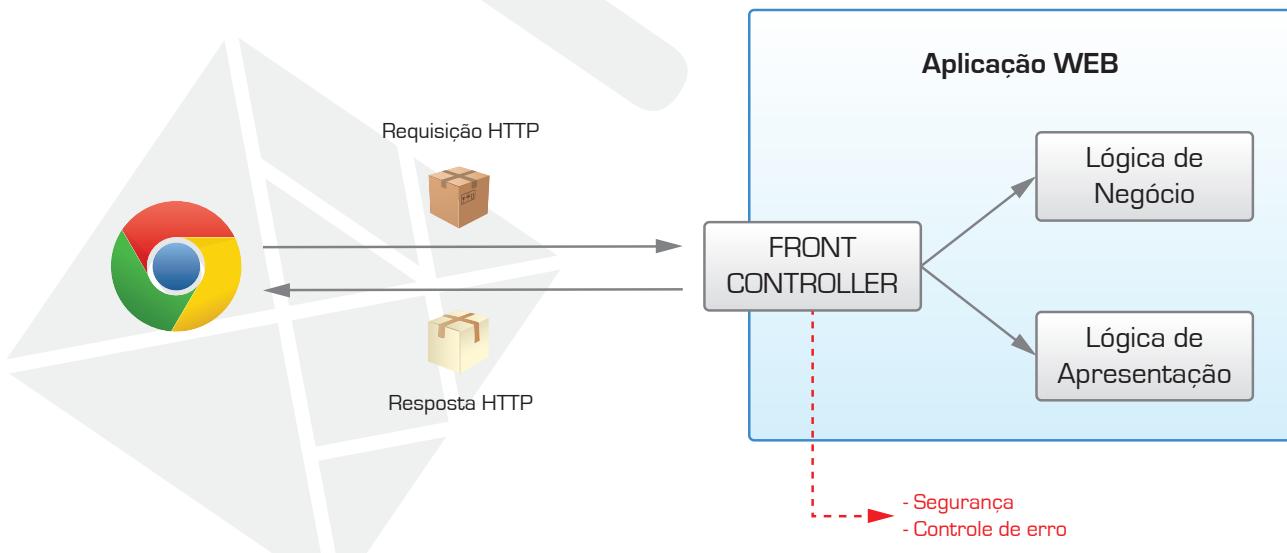


Figura 3.11: *Front Controller*



Exercícios de Fixação

- 31 Crie um projeto web chamado **FrontController**.
- 32 Configure um web container através da view **servers** e implante o projeto **FrontController** no web container configurado.
- 33 Adicione a seguinte implementação do FrontController no nosso framework.

```

1 package controllers;
2
3 import java.io.IOException;
4 import java.lang.reflect.Method;
5
6 import javax.servlet.RequestDispatcher;
7 import javax.servlet.ServletException;
8 import javax.servlet.annotation.WebServlet;
9 import javax.servlet.http.HttpServlet;
10 import javax.servlet.http.HttpServletRequest;
11 import javax.servlet.http.HttpServletResponse;
12
13 @WebServlet("*.k19")
14 public class FrontController extends HttpServlet {
15     private static final long serialVersionUID = 1L;
16
17     public void service(HttpServletRequest req, HttpServletResponse res)
18         throws ServletException, IOException {
19         String[] split = req.getRequestURI().split("/");
20
21         String controllerName = split[2];
22         String actionPerformed = split[3].split("\\.")[0];
23
24         System.out.println(controllerName);
25         System.out.println(actionPerformed);
26
27     try {
28
29         Class<?> controllerClass = Class.forName("controllers."
30             + controllerName);
31         Method method = controllerClass.getDeclaredMethod(actionPerformed);
32
33         Object controller = controllerClass.newInstance();
34         method.invoke(controller);
35
36         RequestDispatcher dispatcher = req.getRequestDispatcher("/" + controllerName
37             + "/" + actionPerformed + ".jsp");
38
39         dispatcher.forward(req, res);
40     } catch (Exception e) {
41         e.printStackTrace();
42     }
43 }
44 }
```

Código Java 3.58: *FrontController.java*

- 34 Adicione um controlador nos padrões do nosso framework.

```

1 package controllers;
2
3 public class Teste {
4     public void teste(){
5         System.out.println("Teste.teste()");
6     }
7 }
```

Código Java 3.59: Teste.java

- 35** Adicione a pasta **Teste** que contém o arquivo **teste.jsp** na pasta **WebContent**.

```

1 <html>
2     <head>
3         <title>Teste - teste</title>
4     </head>
5     <body>
6         <h1>controller: Teste</h1>
7         <h1>action: teste</h1>
8     </body>
9 </html>
```

Código JSP 3.1: teste.jsp

- 36** Inicialize o web container e accese a página:

<http://localhost:8080/FrontController/Teste/teste.k19>



Flyweight

Objetivo: Compartilhar, de forma eficiente, objetos que são usados em grande quantidade.

Exemplo prático

Estamos desenvolvendo uma aplicação para gerenciar milhares de apresentações com slides. Essa aplicação disponibilizará um conjunto de temas que podem ser aplicados individualmente em cada slide de uma apresentação.

Em geral, o conteúdo (título e texto) de cada slide é único. Portanto, não seria possível compartilhar de maneira eficiente o conteúdo de slides diferentes.

Por outro lado, como vários slides podem utilizar o mesmo tema, eles poderiam compartilhar as informações relativas à formatação da fonte, cor de fundo, layout, e etc. Consequentemente, a quantidade de memória utilizada seria drasticamente reduzida.

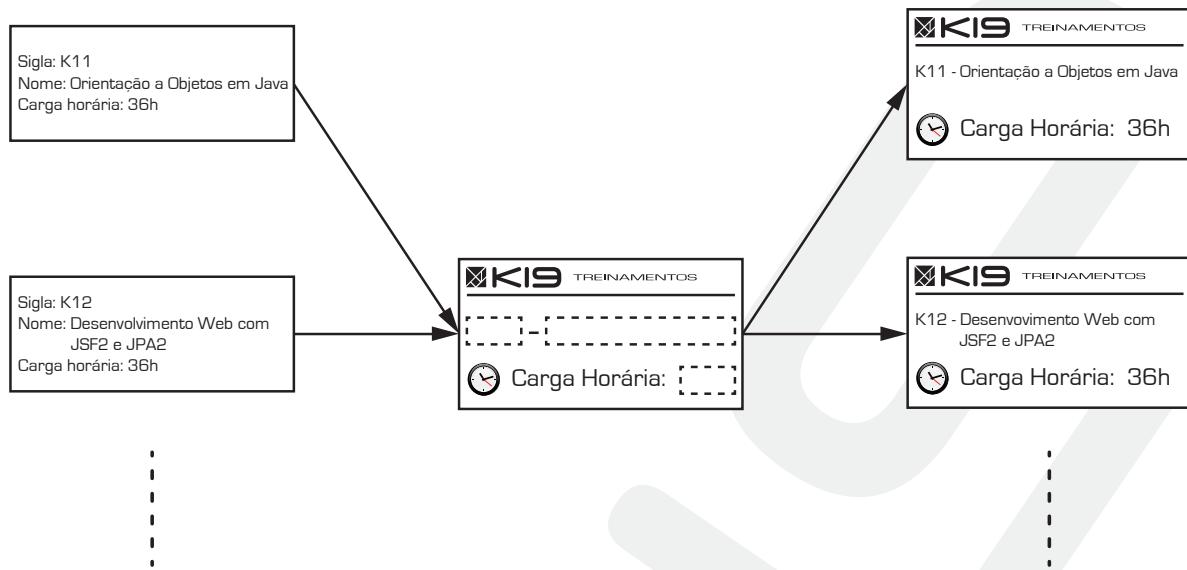


Figura 3.12: O mesmo tema sendo aplicado a diferentes slides

Podemos definir uma interface para padronizar o funcionamento dos temas.

```
1 public interface TemaFlyweight {
2     void imprime(String titulo, String texto);
3 }
```

Código Java 3.60: TemaFlyweight.java

A interface TemaFlyweight define um método que recebe o conteúdo dos slides e deve aplicar a formatação correspondente ao tema. Com a interface definida, podemos implementar alguns temas específicos.

```
1 public class TemaHifen implements TemaFlyweight {
2     public void imprime(String titulo, String texto) {
3         // implementação
4     }
5 }
```

Código Java 3.61: TemaHifen.java

```
1 public class TemaAsterisco implements TemaFlyweight {
2     public void imprime(String titulo, String texto) {
3         // implementação
4     }
5 }
```

Código Java 3.62: TemaAsterisco.java

```
1 public class TemaK19 implements TemaFlyweight {
2     public void imprime(String titulo, String texto) {
3         // implementação
4     }
5 }
```

Código Java 3.63: TemaK19.java

Para controlar a criação e acesso dos objetos que definem os temas, podemos implementar a seguinte classe.

```

1 public class TemaFlyweightFactory {
2     private static Map<Class<? extends TemaFlyweight>, TemaFlyweight> temas = new ←
3         HashMap<Class<? extends TemaFlyweight>, TemaFlyweight>();
4     public static final Class<TemaAsterisco> ASTERISCO = TemaAsterisco.class;
5     public static final Class<TemaHifen> HIFEN = TemaHifen.class;
6     public static final Class<TemaK19> K19 = TemaK19.class;
7
8     public static TemaFlyweight getTema(Class<? extends TemaFlyweight> clazz) {
9         if (!temas.containsKey(clazz)) {
10             try {
11                 temas.put(clazz, clazz.newInstance());
12             } catch (Exception e) {
13                 e.printStackTrace();
14             }
15         }
16         return temas.get(clazz);
17     }

```

Código Java 3.64: TemaFlyweightFactory.java

Por fim, deveríamos associar os slides aos temas. Isso poderia ser feito através do construtor da classe que define os slides.

```

1 public class Slide {
2     private TemaFlyweight tema;
3
4     public Slide(TemaFlyweight tema) {
5         this.tema = tema;
6     }
7 }

```

Código Java 3.65: Slide.java

```

1 TemaFlyweight tema = TemaFlyweightFactory.getTema(TemaFlyweightFactory.K19);
2 Slide slide = new Slide(tema);

```

Código Java 3.66: Slide.java

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

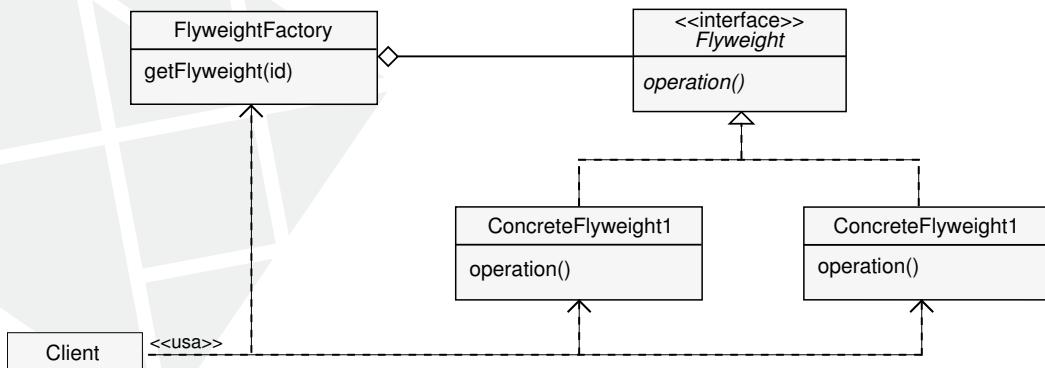


Figura 3.13: UML do padrão Flyweight

Os personagens desse padrão são:

Flyweight (TemaFlyweight)

Interface que define os objetos que serão compartilhados.

ConcreteFlyweight (TemaHifen, TemaAsterisco, TemaK19)

Tipo específico de Flyweight.

FlyweightFactory (TemaFlyweightFactory)

Classe que controla a criação e recuperação de Flyweights.

Client

Utiliza FlyweightFactory para recuperar os Flyweights.



Exercícios de Fixação

37 Crie um projeto chamado **Flyweight**.

38 Defina a interface **TemaFlyweight**.

```
1 public interface TemaFlyweight {
2     void imprime(String titulo, String texto);
3 }
```

Código Java 3.67: TemaFlyweight.java

39 Agora, implemente alguns temas.

```
1 public class TemaHifen implements TemaFlyweight {
2     public void imprime(String titulo, String texto) {
3         System.out.println("----- " + titulo + " -----");
4         System.out.println(texto);
5         char[] rodape = new char[22 + titulo.length()];
6         Arrays.fill(rodape, '-');
7         System.out.println(rodape);
8     }
9 }
```

Código Java 3.68: TemaHifen.java

```
1 public class TemaAsterisco implements TemaFlyweight {
2     public void imprime(String titulo, String texto) {
3         System.out.println("***** " + titulo + " *****");
4         System.out.println(texto);
5         char[] rodape = new char[22 + titulo.length()];
6         Arrays.fill(rodape, '*');
7         System.out.println(rodape);
8     }
9 }
```

Código Java 3.69: TemaAsterisco.java

```

1 public class TemaK19 implements TemaFlyweight {
2     public void imprime(String titulo, String texto) {
3         System.out
4             .println("##### " + titulo.toUpperCase() + " #####");
5         System.out.println(texto);
6         char[] rodapeE = new char[(int) Math.floor((6 + titulo.length()) / 2.0)];
7         char[] rodapeD = new char[(int) Math.ceil((6 + titulo.length()) / 2.0)];
8         Arrays.fill(rodapeE, '#');
9         Arrays.fill(rodapeD, '#');
10        System.out.println(new String(rodapeE) + " www.k19.com.br "
11                           + new String(rodapeD));
12    }
13 }

```

Código Java 3.70: TemaK19.java

- 40** Defina uma classe para controlar a criação e recuperação dos temas.

```

1 public class TemaFlyweightFactory {
2     private static Map<Class<? extends TemaFlyweight>, TemaFlyweight> temas = new ←
3         HashMap<Class<? extends TemaFlyweight>, TemaFlyweight>();
4     public static final Class<TemaAsterisco> ASTERISCO = TemaAsterisco.class;
5     public static final Class<TemaHifen> HIFEN = TemaHifen.class;
6     public static final Class<TemaK19> K19 = TemaK19.class;
7
8     public static TemaFlyweight getTema(Class<? extends TemaFlyweight> clazz) {
9         if (!temas.containsKey(clazz)) {
10             try {
11                 temas.put(clazz, clazz.newInstance());
12             } catch (Exception e) {
13                 e.printStackTrace();
14             }
15         }
16         return temas.get(clazz);
17     }
18 }

```

Código Java 3.71: TemaFlyweightFactory.java

- 41** Defina uma classe para representar um slide.

```

1 public class Slide {
2     private TemaFlyweight tema;
3     private String titulo;
4     private String texto;
5
6     public Slide(TemaFlyweight tema, String titulo, String texto) {
7         this.tema = tema;
8         this.titulo = titulo;
9         this.texto = texto;
10    }
11
12    public void imprime() {
13        this.tema.imprime(titulo, texto);
14    }
15 }

```

Código Java 3.72: Slide.java

- 42** Crie uma classe para modelar uma apresentação.

```

1 public class Apresentacao {
2     private List<Slide> slides = new ArrayList<Slide>();
3
4     public void adicionaSlide(Slide slide) {
5         slides.add(slide);
6     }
7
8     public void imprime() {
9         for (Slide slide : this.slides) {
10             slide.imprime();
11             System.out.println();
12         }
13     }
14 }
```

Código Java 3.73: Apresentacao.java

43 Teste a utilização dos temas.

```

1 public class TestaTemas {
2     public static void main(String[] args) {
3         Apresentacao a = new Apresentacao();
4         a.adicionaSlide(new Slide(TemaFlyweightFactory
5             .getTema(TemaFlyweightFactory.K19),
6             "K11 - Orientação a Objetos em Java",
7             "Com este curso você vai obter uma base\n"
8             + "sólida de conhecimentos de Java\n"
9             + "e de Orientação a Objetos."));
10        a.adicionaSlide(new Slide(TemaFlyweightFactory
11            .getTema(TemaFlyweightFactory.ASTERISCO),
12            "K12 - Desenvolvimento Web com JSF2 e JPA2",
13            "Depois deste curso, você estará apto a\n"
14            + "desenvolver aplicações Web com\n"
15            + "os padrões da plataforma Java.");
16        a.adicionaSlide(new Slide(TemaFlyweightFactory
17            .getTema(TemaFlyweightFactory.HIFEN),
18            "K21 - Persistência com JPA2 e Hibernate",
19            "Neste curso de Java Avançado, abordamos de\n"
20            + "maneira profunda os recursos de persistência\n"
21            + "do JPA2 e do Hibernate.");
22
23        a.imprime();
24    }
25 }
```

Código Java 3.74: TestaTemas.java



Proxy

Objetivo: Controlar as chamadas a um objeto através de outro objeto de mesma interface.

Se uma queda de energia ocorrer enquanto estamos desenvolvendo um trabalho no computador, este trabalho pode ser perdido. Para evitar situações como essa, podemos utilizar um *no-break*. Após uma interrupção no fornecimento de energia, este aparelho mantém o computador ligado por tempo suficiente para que possamos salvar o nosso trabalho.

Para isso, o computador deve estar conectado ao *no-break* e não à tomada. O *no-break*, por sua vez, deve estar conectado à tomada.

Não queremos alterar o plugue do computador, portanto é interessante que o *no-break* possua o mesmo encaixe que a tomada.

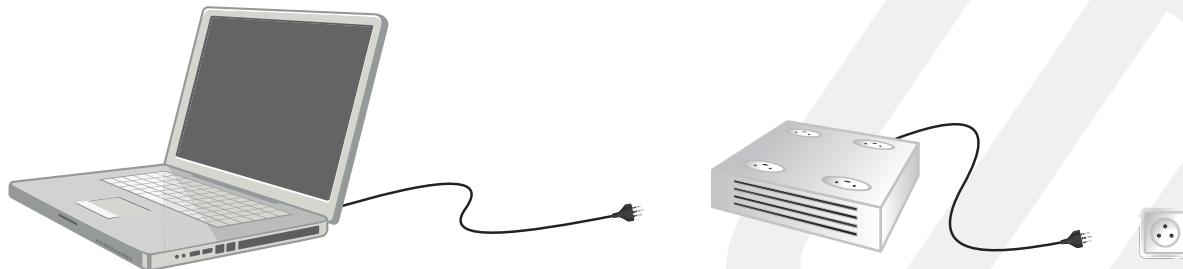


Figura 3.14: Computador conectado ao no-break

Essa é a mesma ideia do padrão Proxy. Esse padrão define um intermediário para controlar o acesso a um determinado objeto, podendo adicionar funcionalidades que esse objeto não possui.

Exemplo prático

Estamos desenvolvendo uma aplicação bancária que deve registrar todas as operações realizadas pelos objetos que representam as contas do banco. O registro das operações pode ser utilizado posteriormente em uma auditoria.

Para manter o sistema mais coeso, não queremos implementar o registro das operações dentro dos objetos que representam as contas. A ideia é implementar essa lógica em objetos intermediários. Para preservar o modo de utilização das contas, podemos manter a interface nesses objetos intermediários.

Podemos definir uma interface para padronizar os métodos dos objetos que representam as contas e os objetos intermediários responsáveis pelo registro das operações.

```

1 public interface Conta {
2     void deposita(double valor);
3     void saca(double valor);
4     double getSaldo();
5 }
```

Código Java 3.75: Conta.java

Podemos implementar vários tipos específicos de contas. Por exemplo:

```

1 public class ContaPadrao implements Conta {
2     private double saldo;
3
4     public void deposita(double valor) {
5         this.saldo += valor;
6     }
7
8     public void saca(double valor) {
9         this.saldo -= valor;
10    }
11
12    public double getSaldo() {
13        return this.saldo;
14    }
15 }
```

Código Java 3.76: ContaPadrão.java

Implementando a mesma interface das contas, podemos definir os objetos intermediários.

```

1 public class ContaProxy implements Conta {
2     private Conta conta;
3
4     public ContaProxy(Conta conta) {
5         this.conta = conta;
6     }
7
8     public void deposita(double valor) {
9         // registra a operação
10        this.conta.deposita(valor);
11    }
12
13    public void saca(double valor) {
14        // registra a operação
15        this.conta.saca(valor);
16    }
17
18    public double getSaldo() {
19        // registra a operação
20        return this.conta.getSaldo();
21    }
22 }
```

Código Java 3.77: ContaProxy.java

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

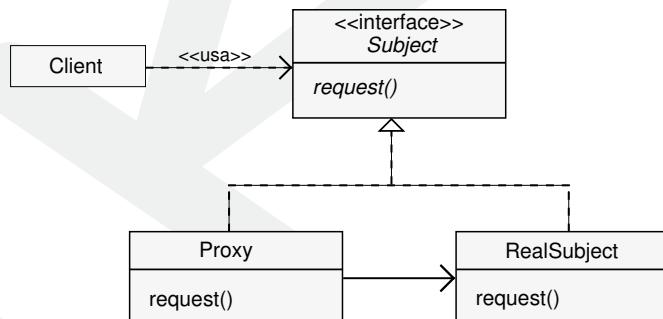


Figura 3.15: UML do padrão Proxy

Os personagens desse padrão são:

Subject (Conta)

Interface que padroniza RealSubject e Proxy.

RealSubject (ContaPadrão)

Define um tipo de objeto do domínio da aplicação.

Proxy (ContaProxy)

Define os objetos que controlam o acesso aos RealSubjects.

Client

Cliente que usa o RealSubject por meio do Proxy.



Exercícios de Fixação

- 44 Crie um projeto chamado **Proxy**.

- 45 Crie uma interface **Conta**.

```

1 public interface Conta {
2     void deposita(double valor);
3     void saca(double valor);
4     double getSaldo();
5 }
```

Código Java 3.78: Conta.java

- 46 Defina a classe **ContaPadrao** que implementa a interface **Conta**.

```

1 public class ContaPadrao implements Conta {
2     private double saldo;
3
4     public void deposita(double valor) {
5         this.saldo += valor;
6     }
7
8     public void saca(double valor) {
9         this.saldo -= valor;
10    }
11
12    public double getSaldo() {
13        return this.saldo;
14    }
15 }
```

Código Java 3.79: ContaPadrao.java

- 47 Crie uma classe intermediária **ContaProxy** que fará os registros das operações.

```

1 public class ContaProxy implements Conta{
2     private Conta conta;
3     public ContaProxy(Conta conta) {
4         this.conta = conta;
5     }
6
7     public void deposita(double valor) {
8         System.out.println("Efetuando o depósito de R$ "+valor+"...");
9         this.conta.deposita(valor);
10        System.out.println("Depósito de R$ "+valor+" efetuado...");
11    }
12
13    public void saca(double valor) {
14        System.out.println("Efetuando o saque de R$ "+valor);
15        this.conta.saca(valor);
16        System.out.println("Saque de R$ "+valor+" efetuado.");
17 }
```

```
17 }
18 }
19
20 public double getSaldo() {
21     System.out.println("Verificando o saldo...");
22     return this.conta.getSaldo();
23 }
24 }
```

Código Java 3.80: ContaProxy.java

48 Teste a classe **ContaProxy**.

```
1 public class TesteProxy {
2     public static void main(String[] args) {
3         Conta contaPadrao = new ContaPadrao();
4         Conta contaProxy = new ContaProxy(contaPadrao);
5         contaProxy.deposita(100);
6         contaProxy.saca(59);
7         System.out.println("Saldo: " + contaProxy.getSaldo());
8     }
9 }
```

Código Java 3.81: TesteProxy.java



PADRÕES COMPORTAMENTAIS

Veja abaixo um resumo do objetivo de cada padrão comportamental.

Command Controlar as chamadas a um determinado componente, modelando cada requisição como um objeto. Permitir que as operações possam ser desfeitas, enfileiradas ou registradas.

Iterator Fornecer um modo eficiente para percorrer sequencialmente os elementos de uma coleção, sem que a estrutura interna da coleção seja exposta.

Mediator Diminuir a quantidade de “ligações” entre objetos introduzindo um mediador, através do qual toda comunicação entre os objetos será realizada.

Observer Definir um mecanismo eficiente para reagir às alterações realizadas em determinados objetos.

State Alterar o comportamento de um determinado objeto de acordo com o estado no qual ele se encontra.

Strategy Permitir de maneira simples a variação dos algoritmos utilizados na resolução de um determinado problema.

Template Method Definir a ordem na qual determinados passos devem ser realizados na resolução de um problema e permitir que esses passos possam ser realizados de formas diferentes de acordo com a situação.

Visitor Permitir atualizações específicas em uma coleção de objetos de acordo com o tipo particular de cada objeto atualizado.



Command

Objetivo: Controlar as chamadas a um determinado componente, modelando cada requisição como um objeto. Permitir que as operações possam ser desfeitas, enfileiradas ou registradas.

Exemplo prático

Estamos desenvolvendo um aplicativo para gerenciar playlists de música. Os usuários poderão selecionar suas músicas favoritas e definir a ordem na qual elas devem ser reproduzidas. Um playlist é basicamente uma sequência de músicas. Contudo, o aplicativo pode adicionar, entre as músicas de um playlist, comandos para aumentar ou diminuir o volume de reprodução.

Vamos utilizar uma biblioteca de áudio para desenvolver esse aplicativo. Através dessa biblioteca, podemos tocar músicas e controlar o volume das saídas de áudio.

```

1 public class Player {
2     public void play(File file) {
3         // implementação
4     }
5
6     public void increaseVolume(int levels) {
7         // implementação
8     }
9
10    public void decreaseVolume(int levels) {
11        // implementação
12    }
13 }
```

Código Java 4.1: Player.java (classe da biblioteca)

Os três métodos da classe Player são “bloqueantes”. Por exemplo, o método play() só termina quando a música que está sendo reproduzida terminar.

Devemos controlar os comandos enviados ao player da biblioteca para poder implementar o nosso aplicativo. Vamos definir uma interface para padronizar os comandos e criar classes para modelar os comandos.

```

1 public interface Comando {
2     void executa();
3 }
```

Código Java 4.2: Comando.java

```

1 public class TocaMusicaComando implements Comando {
2     private Player player;
3     private File file;
4
5     public TocaMusicaComando(Player player, File file) {
6         this.player = player;
7         this.file = file;
8     }
9
10    public void executa() {
11        this.player.play(this.file);
12    }
13 }
```

Código Java 4.3: TocaMusicaComando.java

```

1 public class AumentaVolumeComando implements Comando {
2     private Player player;
3     private int levels;
4
5     public AumentaVolumeComando(Player player, int levels) {
6         this.player = player;
7         this.levels = levels;
8     }
9
10    public void executa() {
11        this.player.increaseVolume(this.levels);
12    }
13 }
```

Código Java 4.4: AumentaVolumeComando.java

```

1 public class DiminuiVolumeComando implements Comando {
2     private Player player;
```

```

3  private int levels;
4
5  public DiminuiVolumeComando(Player player, int levels) {
6      this.player = player;
7      this.levels = levels;
8  }
9
10 public void executa() {
11     this.player.decreaseVolume(this.levels);
12 }
13 }
```

Código Java 4.5: DiminuiVolumeComando.java

Agora, devemos implementar uma classe que dispara os comandos na ordem definida pelo usuário.

```

1 public class ListaDeComandos {
2     private List<Comando> comandos = new ArrayList<Comando>();
3
4     public void adiciona(Comando comando) {
5         this.comandos.add(comando);
6     }
7
8     public void executa(){
9         for(Comando comando : this.comandos) {
10             comando.executa();
11         }
12     }
13 }
```

Código Java 4.6: ListaDeComandos.java

Por fim, teríamos que criar os comandos associados a um player e depois adicioná-los em um playlist.

```

1 Player player = new Player();
2 ListaDeComandos listaDeComandos = new ListaDeComandos();
3
4 listaDeComandos.adiciona(new TocaMusicaComando(player, new File("musica1.mp3")));
5 listaDeComandos.adiciona(new AumentaVolumeComando(player, 3));
6 listaDeComandos.adiciona(new TocaMusicaComando(player, new File("musica2.mp3")));
7 listaDeComandos.adiciona(new DiminuiVolumeComando(player, 3));
8 listaDeComandos.adiciona(new TocaMusicaComando(player, new File("musica3.mp3")));
9
10 listaDeComandos.executa();
```

Código Java 4.7: exemplo de uso

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

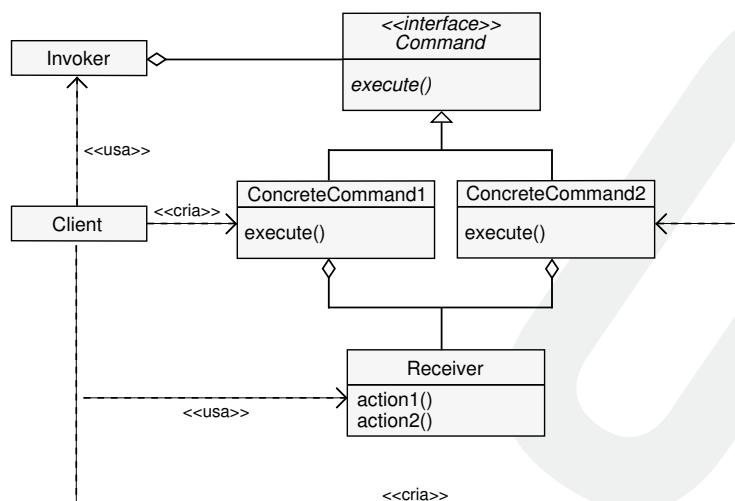


Figura 4.1: UML do padrão Command

Os personagens desse padrão são:

Command (Comando)

Define uma interface para a execução dos métodos do Receiver.

ConcreteCommand (TocaMusicaComando, AumentaVolumeComando, DiminuiVolumeComando)

Classe que implementa Command e modela uma operação específica do Receiver.

Invoker (ListaDeComandos)

Classe que armazena os Commands que devem ser executados.

Receiver (Player)

Define os objetos que terão as chamadas aos seus métodos controladas.

Client

Instancia os Commands associando-os ao Receiver e armazena-os no Invoker.



Exercícios de Fixação

1 Crie um projeto chamado **Command**.

2 Crie uma classe Player.

```

1 public class Player {
2     public void play(String filename) throws InterruptedException {
3         System.out.println("Tocando o arquivo "+filename);
4         long duracao = (long) (Math.random()*2000);
5         System.out.println("Duração (s) :" + duracao/1000.0);
6         Thread.sleep(duracao);
7         System.out.println("Fim");
8     }
9 }
```

```

10 public void increaseVolume(int levels) {
11     System.out.println("Diminuindo o volume em " + levels);
12 }
13
14 public void decreaseVolume(int levels) {
15     System.out.println("Aumentando o volume em " + levels);
16 }
17 }
```

Código Java 4.8: Player.java

- 3 Defina uma interface para padronizar os comandos enviados ao player.

```

1 public interface Comando {
2     void executa(Player player);
3 }
```

Código Java 4.9: Comando.java

- 4 Defina as classes que modelarão os comandos enviados ao player.

```

1 public class TocaMusicaComando implements Comando {
2     private Player player;
3     private String file;
4
5     public TocaMusicaComando(Player player, String file) {
6         this.player = player;
7         this.file = file;
8     }
9
10    public void executa() {
11        try {
12            this.player.play(this.file);
13        } catch (InterruptedException e) {
14            e.printStackTrace();
15        }
16    }
17 }
```

Código Java 4.10: TocaMusicaComando.java

```

1 public class AumentaVolumeComando implements Comando {
2     private Player player;
3     private int levels;
4
5     public AumentaVolumeComando(Player player, int levels) {
6         this.player = player;
7         this.levels = levels;
8     }
9
10    public void executa() {
11        this.player.increaseVolume(this.levels);
12    }
13 }
```

Código Java 4.11: AumentaVolumeComando.java

```

1 public class DiminuiVolumeComando implements Comando {
2     private Player player;
3     private int levels;
4
5     public DiminuiVolumeComando(Player player, int levels) {
```

```

6     this.player = player;
7     this.levels = levels;
8 }
9
10 public void executa() {
11     this.player.decreaseVolume(this.levels);
12 }
13 }
```

Código Java 4.12: DiminuiVolumeComando.java

- 5 Defina uma classe ListaDeComandos que conterá a lista de comandos na ordem definida pelo usuário.

```

1 public class ListaDeComandos {
2     private List<Comando> comandos = new ArrayList<Comando>();
3
4     public void adiciona(Comando comando) {
5         this.comandos.add(comando);
6     }
7
8     public void executa() {
9         for (Comando comando : this.comandos) {
10             comando.executa();
11         }
12     }
13 }
```

Código Java 4.13: ListaDeComandos.java

- 6 Faça uma classe para testar a ListaDeComandos.

```

1 public class TestaListaDeComandos {
2     public static void main(String[] args) {
3         Player player = new Player();
4         ListaDeComandos listaDeComandos = new ListaDeComandos();
5
6         listaDeComandos.adiciona(new TocaMusicaComando(player, "musica1.mp3"));
7         listaDeComandos.adiciona(new AumentaVolumeComando(player, 3));
8         listaDeComandos.adiciona(new TocaMusicaComando(player, "musica2.mp3"));
9         listaDeComandos.adiciona(new DiminuiVolumeComando(player, 3));
10        listaDeComandos.adiciona(new TocaMusicaComando(player, "musica3.mp3"));
11
12        listaDeComandos.executa();
13    }
14 }
```

Código Java 4.14: TestaListaComandos.java



Iterator

Objetivo: Fornecer um modo eficiente para percorrer sequencialmente os elementos de uma coleção, sem que a estrutura interna da coleção seja exposta.

Considere um turista que pretende visitar os principais pontos turísticos de determinada cidade. Infelizmente, ele não pôde planejar seus passeios e não sabe quais pontos turísticos existem nessa cidade. Além disso, seu período de permanência na cidade é curto.

Para tornar sua experiência mais agradável, o turista pode solicitar o serviço de um guia turístico. Guias conhecem muito bem a cidade onde atuam e sabem definir um bom roteiro para a visitação dos pontos turísticos.

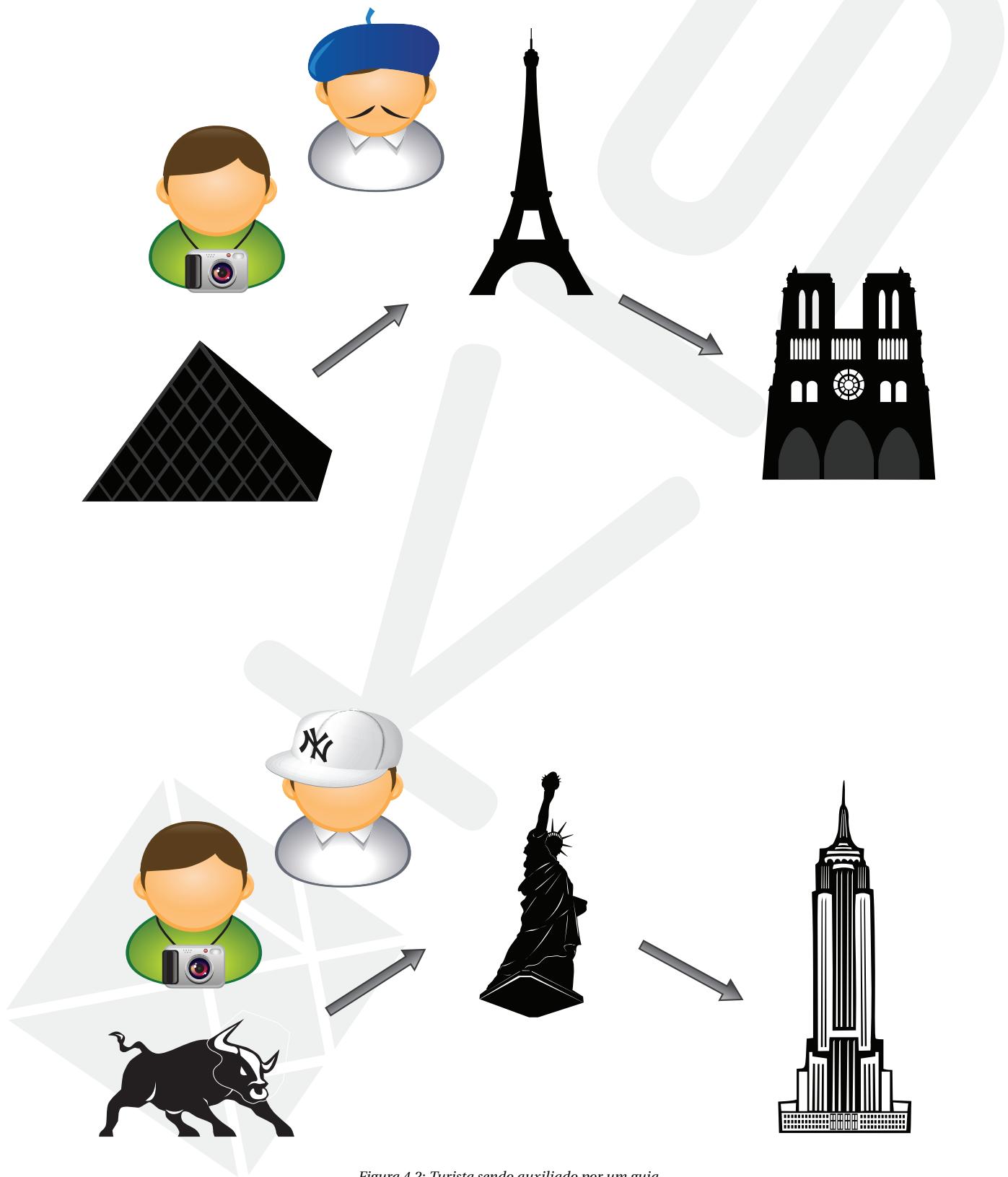


Figura 4.2: Turista sendo auxiliado por um guia

A ideia do padrão Iterator é fornecer um meio eficiente de se percorrer todos os elementos de uma determinada coleção.

Exemplo prático

A maior parte das aplicações necessitam de estruturas de dados para organizar as informações armazenadas na memória. Cada estrutura estabelece um determinado conjunto de restrições relacionadas aos dados e disponibilizam operações eficientes para que possamos manipular as informações.

Em geral, as plataformas de desenvolvimento de aplicações oferecem, através de bibliotecas, diversas estruturas de dados para facilitar o trabalho dos programadores. Por exemplo, na plataforma Java, as classes `ArrayList`, `LinkedList`, `Vector`, `HashSet` e `TreeSet` são exemplos de implementações disponíveis.

Muitas vezes, é necessário percorrer todos os elementos de uma estrutura de dados para realizar uma determinada tarefa. A organização interna de cada estrutura é complexa e deve estar encapsulada nas classes que as implementam. Dessa forma, em geral, o desenvolvedor não teria as condições necessárias para percorrer os elementos da estrutura de dados utilizada.

Podemos encapsular o processo de visitar cada elemento de uma estrutura de dados dentro dela mesma. Essa abordagem é interessante pois cada estrutura conhece a sua organização interna podendo, dessa forma, implementar esse processo da maneira mais eficiente.

Na plataforma Java, as estruturas que possuem a capacidade de ter os seus elementos percorridos implementam uma interface que abstrai a ideia desse processo.

```
1 public interface Iterable<E> {  
2     Iterator<E> iterator();  
3 }
```

Código Java 4.15: *Iterable.java*

Toda estrutura de dados que implementa a interface `Iterable` deve produzir objetos da interface `Iterator`. Os objetos da interface `Iterator` funcionam como guias que indicam o melhor “caminho” para visitar os elementos de uma determinada estrutura.

```
1 Iterator<String> guia = estrutura.iterator();  
2  
3 while(guia.hasNext()) {  
4     String elemento = guia.next();  
5 }
```

Código Java 4.16: Utilizando um *Iterator*

Do ponto de vista do programador, a complexidade para percorrer os elementos de uma estrutura de dados está “escondida” dentro do iterator.

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

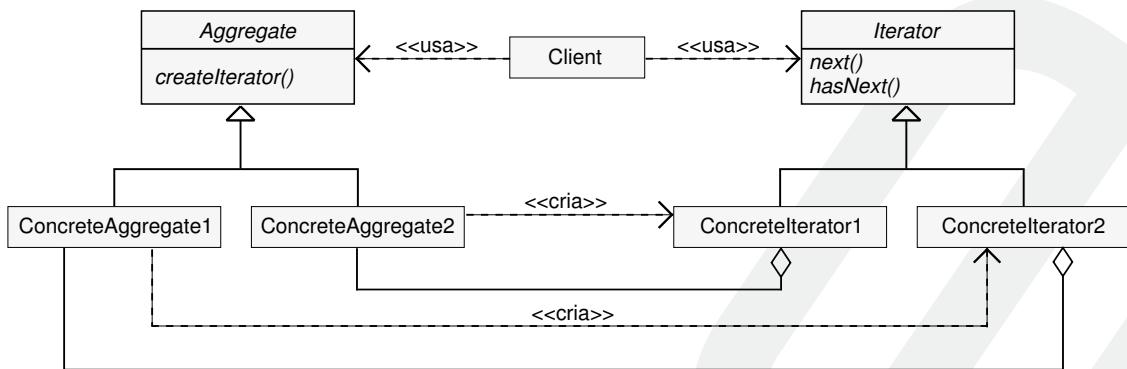


Figura 4.3: UML do padrão Iterator

Os personagens desse padrão são:

Iterator (Iterator)

Define a interface dos objetos que encapsulam toda a complexidade para percorrer os elementos do Aggregate.

ConcreteIterator

Implementação da interface Iterator para um tipo específico de Aggregate.

Aggregate (Iterable)

Define a interface das coleções de objetos que podem ter seus elementos percorridos através de um Iterator.

ConcreteAggregate (ArrayList, LinkedList, Vector, HashSet, TreeSet)

Estrutura de dados que implementa o Aggregate.



Exercícios de Fixação

7 Crie um projeto chamado **Iterator**.

8 Defina uma classe **ListaDeNomes** que implementa a interface Iterable e mantém a lista de nomes num array.

```

1 public class ListaDeNomes implements Iterable<String> {
2     private String[] nomes;
3     private int length;
4
5     public ListaDeNomes(String[] nomes) {
6         this.nomes = nomes;
7         this.length = this.nomes.length;
8     }
9
10    public Iterator<String> iterator() {
11        return this.new ListaDeNomesIterator();
12    }
13
14    private class ListaDeNomesIterator implements Iterator<String> {
  
```

```

15     private int i = 0;
16
17     public boolean hasNext() {
18         return (this.i) < ListaDeNomes.this.length;
19     }
20
21     public String next() {
22         return ListaDeNomes.this.nomes[i++];
23     }
24
25     public void remove() {
26         ListaDeNomes.this.nomes[i] = null;
27
28         for (int j = i; (j + 1) < ListaDeNomes.this.length; j++) {
29             ListaDeNomes.this.nomes[j] = ListaDeNomes.this.nomes[j + 1];
30         }
31         ListaDeNomes.this.length--;
32     }
33 }
34 }
```

Código Java 4.17: *ListaDeNomes.java*

9 Teste a classe *ListaDeNomes*.

```

1 public class TestaIterator {
2     public static void main(String[] args) {
3         String[] nomes = new String[4];
4         nomes[0] = "Rafael Cosentino";
5         nomes[1] = "Marcelo Martins";
6         nomes[2] = "Jonas Hirata";
7         nomes[3] = "Solange Domingues";
8
9         ListaDeNomes listaDeNomes = new ListaDeNomes(nomes);
10        Iterator<String> iterator = listaDeNomes.iterator();
11        iterator.hasNext();
12        iterator.remove();
13
14        while (iterator.hasNext()) {
15            String nome = iterator.next();
16            System.out.println(nome);
17        }
18
19        System.out.println("-----");
20        System.out.println("Testando o foreach");
21        for (String nome : listaDeNomes) {
22            System.out.println(nome);
23        }
24    }
25 }
```

Código Java 4.18: *TestaIterator*

Mediator

Objetivo: Diminuir a quantidade de “ligações” entre objetos introduzindo um mediador, através do qual toda comunicação entre os objetos será realizada.

Considere a rede de telefonia fixa. Ela permite que ligações telefônicas sejam realizadas entre quaisquer dois aparelhos dessa rede. Obviamente, os aparelhos não estão diretamente conectados entre si dois a dois. Tente imaginar a quantidade de fios que teria de estar conectada a cada aparelho.

Os telefones estão conectados a uma central que distribui as ligações de acordo com o número discado, criando uma ligação temporária entre os dois aparelhos.

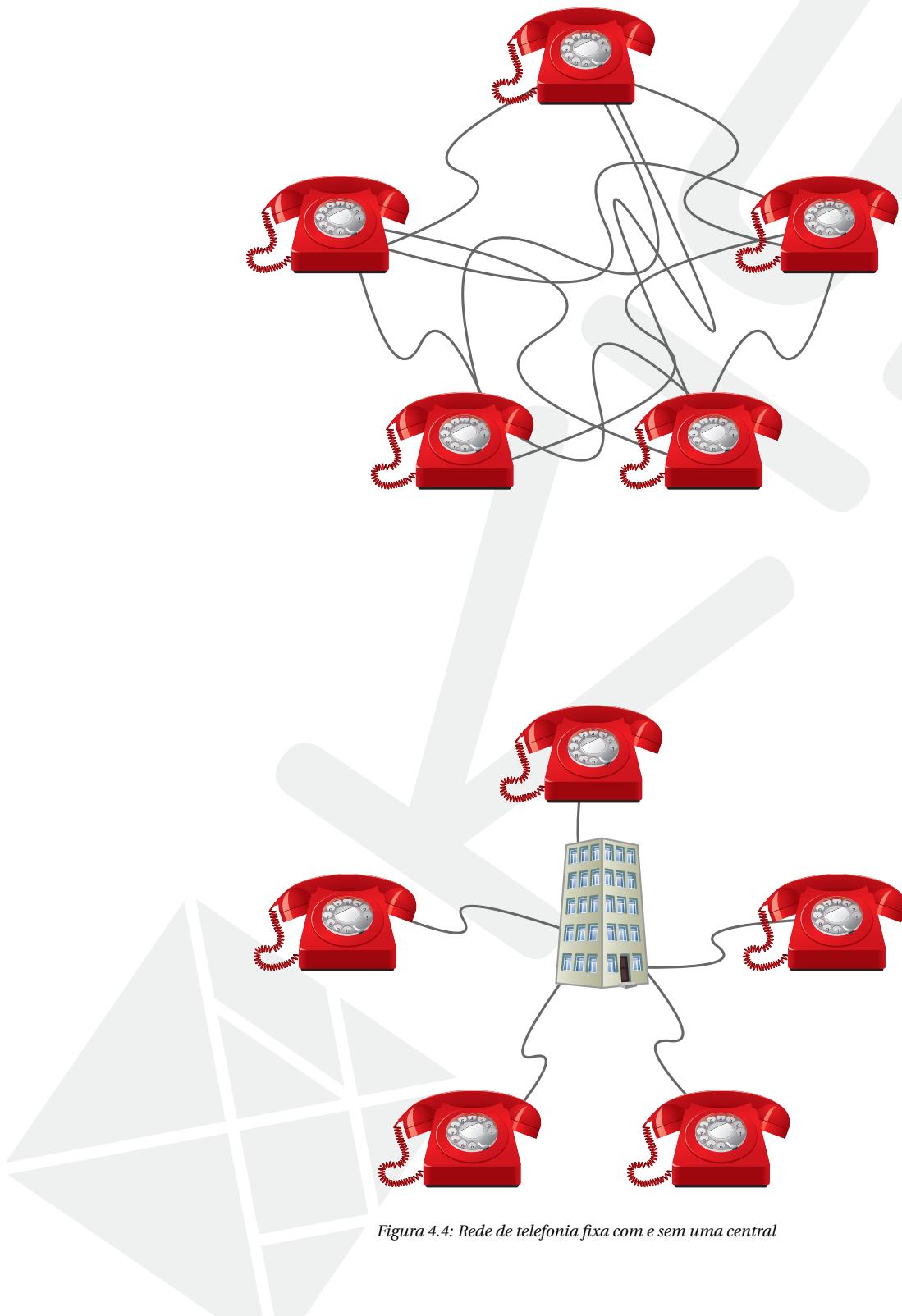


Figura 4.4: Rede de telefonia fixa com e sem uma central

A ideia do padrão Mediator é semelhante à ideia da central telefônica. Eliminar conexões excessivas entre elementos por meio da introdução de um intermediário único.

Exemplo prático

Estamos desenvolvendo um sistema para controlar o fluxo de táxis e passageiros em um aeroporto. Os táxis disponíveis ficam a espera de passageiros em uma fila organizada pela ordem de chegada. Da mesma forma, os passageiros que desejam utilizar um táxi ficam em uma fila de espera que também é organizada pela ordem de chegada.

Criaremos uma classe para modelar os passageiros, outra para os táxis e uma terceira para mediar a comunicação entre esses objetos.

```
1 public class Passageiro {
2     // implementação
3 }
```

Código Java 4.19: Passageiro.java

```
1 public class Taxi {
2     // implementação
3 }
```

Código Java 4.20: Taxi.java

```
1 public class CentralDeTaxi {
2     // implementação
3 }
```

Código Java 4.21: CentralDeTaxi.java

A central de táxi deve manter uma fila com os táxis disponíveis e outra com os passageiros em espera.

```
1 public class CentralDeTaxi {
2     private List<Taxi> taxisLivres = new ArrayList<Taxi>();
3     private List<Passageiro> passageirosEmEspera = new ArrayList<Passageiro>();
4 }
```

Código Java 4.22: CentralDeTaxi.java

Além disso, a central de táxi deve disponibilizar um método que será acionado por um táxi quando esse fica disponível, e outro que será chamado por um passageiro quando esse entrar na fila de espera.

```
1 public class CentralDeTaxi {
2     private List<Taxi> taxisLivres = new ArrayList<Taxi>();
3     private List<Passageiro> passageirosEmEspera = new ArrayList<Passageiro>();
4
5     public void adicionaTaxiDisponivel(Taxi taxi) {
6         // implementação
7     }
8
9     public void pedeTaxi(Passageiro passageiro) {
10        // implementação
11    }
12 }
```

Código Java 4.23: CentralDeTaxi.java

Para poder se comunicar com a central de táxi, tanto os táxis quanto os passageiros devem ter uma referência para a central. Essas ligações podem ser estabelecidas através dos construtores das

classes Passageiro e Taxi.

```

1 public class Passageiro {
2     private CentralDeTaxi central;
3     public Passageiro(CentralDeTaxi central) {
4         this.central = central;
5     }
6 }
```

Código Java 4.24: Passageiro.java

```

1 public class Taxi {
2     private CentralDeTaxi central;
3     public Taxi(CentralDeTaxi central) {
4         this.central = central;
5     }
6 }
```

Código Java 4.25: Taxi.java

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

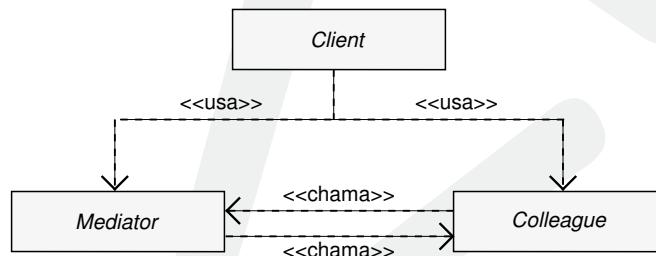


Figura 4.5: UML do padrão Mediator

Os personagens desse padrão são:

Mediator

Interface que padroniza as operações que serão chamadas pelos Colleagues.

ConcreateMediator (CentralDeTaxi)

Implementação particular do Mediator, que coordena a interação entre os Colleagues.

Colleague

Possível interface para padronizar os ConcreateColleagues.

ConcreateColleague (Taxi, Passageiro)

Classes que interagem entre si por meio do Mediator.



Exercícios de Fixação

- 10 Crie um projeto chamado **Mediator**.

- 11 Defina as classes Passageiro, Taxi e CentralDeTaxi.

```

1 public class Passageiro {
2     private CentralDeTaxi central;
3     public Passageiro(CentralDeTaxi central) {
4         this.central = central;
5     }
6 }
```

Código Java 4.26: Passageiro.java

```

1 public class Taxi {
2     private CentralDeTaxi central;
3     public Taxi(CentralDeTaxi central) {
4         this.central = central;
5     }
6 }
```

Código Java 4.27: Taxi.java

```

1 public class CentralDeTaxi {
2     private List<Taxi> taxisLivres = new ArrayList<Taxi>();
3     private List<Passageiro> passageirosEmEspera = new ArrayList<Passageiro>();
4
5     public void adicionaTaxiDisponivel(Taxi taxi) {
6         // implementação
7     }
8
9     public void pedeTaxi(Passageiro passageiro) {
10        // implementação
11    }
12 }
```

Código Java 4.28: CentralDeTaxi.java

- 12 Adicione à classe Passageiro um método para simular a chegada de passageiros. Adicione também um atributo para identificar o passageiro.

```

1 public class Passageiro implements Runnable {
2     private String nome;
3     private CentralDeTaxi central;
4
5     public Passageiro(String nome, CentralDeTaxi central) {
6         this.nome = nome;
7         this.central = central;
8     }
9
10    public String getNome() {
11        return nome;
12    }
13
14    public void run() {
15        for (int i = 0; i < 5; i++) {
16            this.central.pedeTaxi(this);
17        }
18    }
19 }
```

Código Java 4.29: Passageiro.java

- 13 Adicione à classe Taxi um método para simular uma corrida de táxi. Adicione também um

atributo para identificar o táxi.

```

1 public class Taxi {
2     private CentralDeTaxi central;
3     private int id;
4     private static int contador = 0;
5
6     public Taxi(CentralDeTaxi central) {
7         this.central = central;
8         this.id = Taxi.contador++;
9     }
10
11    public int getId() {
12        return id;
13    }
14
15    public void atende() {
16        try {
17            Thread.sleep((long) (Math.random() * 3000.0));
18        } catch (InterruptedException e) {
19            e.printStackTrace();
20        }
21        this.central.adicionaTaxiDisponivel(this);
22    }
23 }
```

Código Java 4.30: Taxi.java

- 14 Implemente o método adicionaTaxiDisponivel() da classe CentralDeTaxi. Esse método deve receber um Taxi como parâmetro e adicioná-lo à fila de táxis livres.

```

1 public class CentralDeTaxi {
2     private List<Taxi> taxisLivres = new ArrayList<Taxi>();
3     private List<Passageiro> passageirosEmEspera = new ArrayList<Passageiro>();
4
5     public synchronized void adicionaTaxiDisponivel(Taxi taxi) {
6         System.out.println("Taxi " + taxi.getId() + " voltou pra fila");
7         taxisLivres.add(taxi);
8         this.notifyAll();
9     }
10 }
```

Código Java 4.31: CentralDeTaxi.java

- 15 Implemente o método pedeTaxi() da classe CentralDeTaxi. Esse método deve receber um Passageiro como parâmetro e adicioná-lo à fila de passageiros.

```

1 public class CentralDeTaxi {
2     private List<Taxi> taxisLivres = new ArrayList<Taxi>();
3     private List<Passageiro> passageirosEmEspera = new ArrayList<Passageiro>();
4
5     public synchronized void adicionaTaxiDisponivel(Taxi taxi) {
6         System.out.println("Taxi " + taxi.getId() + " voltou pra fila");
7         taxisLivres.add(taxi);
8         this.notifyAll();
9     }
10
11    public void pedeTaxi(Passageiro passageiro) {
12        Taxi taxi = this.esperaTaxi(passageiro);
13        System.out.println("Taxi " + taxi.getId() + " levando " + passageiro.getNome());
14        taxi.atende();
15    }
16 }
```

```

17 private Taxi esperaTaxi(Passageiro passageiro) {
18     this.passageirosEmEspera.add(passageiro);
19     synchronized (this) {
20         while (this.taxisLivres.isEmpty())
21             || !this.passageirosEmEspera.get(0).equals(passageiro)) {
22             try {
23                 this.wait();
24             } catch (InterruptedException e) {
25                 e.printStackTrace();
26             }
27         }
28         this.passageirosEmEspera.remove(0);
29         return this.taxisLivres.remove(0);
30     }
31 }
32 }
```

Código Java 4.32: CentralDeTaxi.java

- 16** Crie uma classe para simular o sistema.

```

1 public class TestaCentralDeTaxi {
2     public static void main(String[] args) {
3         CentralDeTaxi central = new CentralDeTaxi();
4
5         Passageiro p1 = new Passageiro("Rafael Cosentino", central);
6         Passageiro p2 = new Passageiro("Marcelo Martins", central);
7         Passageiro p3 = new Passageiro("Jonas Hirata", central);
8
9         Taxi t1 = new Taxi(central);
10        central.adicionaTaxiDisponivel(t1);
11
12        Taxi t2 = new Taxi(central);
13        central.adicionaTaxiDisponivel(t2);
14
15        new Thread(p1).start();
16        new Thread(p2).start();
17        new Thread(p3).start();
18    }
19 }
```

Código Java 4.33: TestaCentralDeTaxi.java



Observer

Objetivo: Definir um mecanismo eficiente para reagir às alterações realizadas em determinados objetos.

Considere um supermercado em que os caixas possuem uma fila única. Cada caixa é identificado por um número, que é exibido em uma placa à sua frente. Esse número fica visível à todos os clientes da fila.

Também visível aos clientes da fila, existe um painel que exibe o número de identificação do próximo caixa disponível (se houver algum).

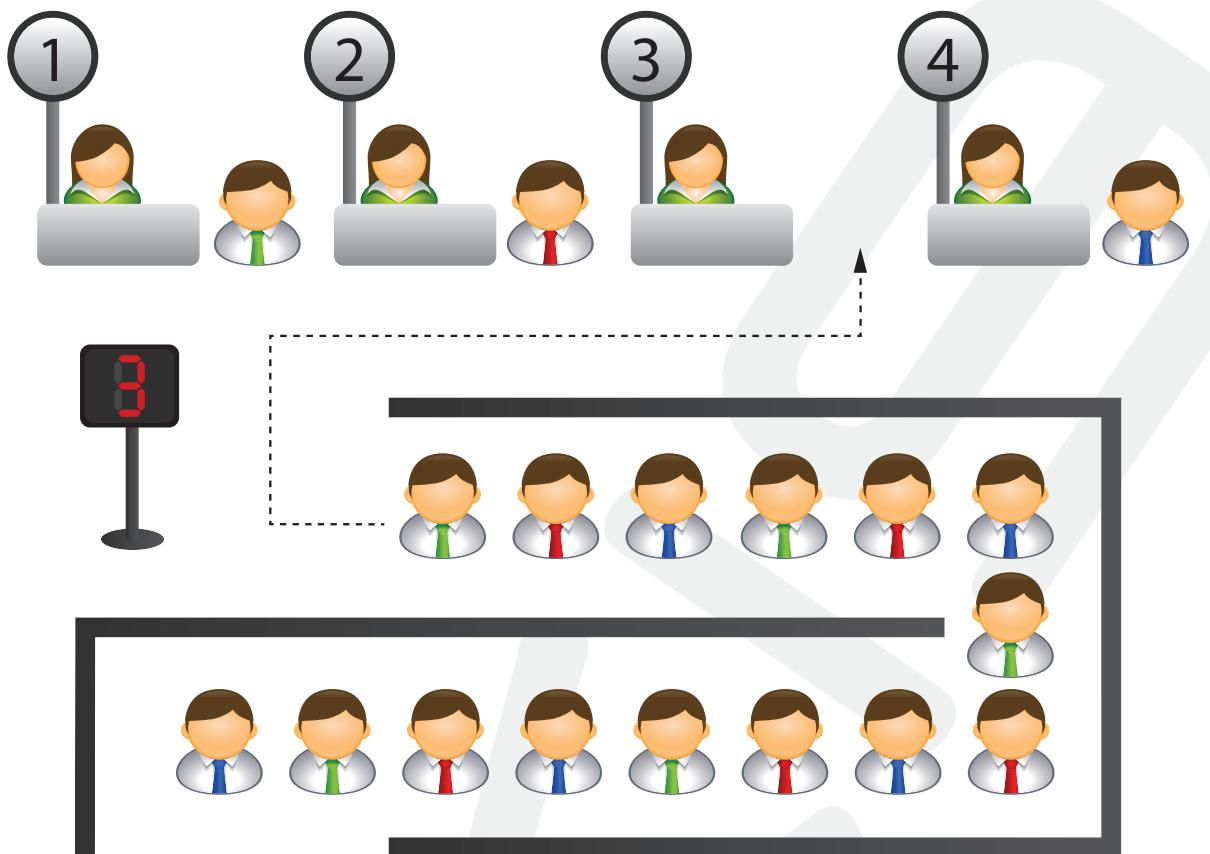


Figura 4.6: Fila dos caixas de um supermercado e o seu painel

O painel precisa saber sobre as mudanças no estado (livre ou ocupado) de cada caixa para atualizar o seu mostrador.

Uma possibilidade é fazer o painel consultar, periodicamente, o estado de cada caixa. Contudo, essa pode ser uma abordagem ineficiente, principalmente quando o estado dos caixas não é alterado frequentemente.

Uma outra possibilidade é fazer com que cada caixa avisasse ao painel sobre uma mudança em seu estado.

Nessa abordagem, no momento em que um caixa fica disponível ou ocupado, ele poderia passar o seu número de identificação para o painel, para informá-lo sobre a sua mudança de estado. O painel, por sua vez, atualizaria o número exibido quando necessário.

A ideia fundamental do padrão Observer é atribuir aos objetos que tem seus estados alterados a tarefa de notificar os objetos interessados nessas mudanças. Em nosso exemplo, os caixas notificam o painel sobre alterações em seus estados.

Exemplo prático

Estamos desenvolvendo um sistema para o mercado financeiro. Esse sistema deve manter todos os interessados em uma determinada ação informados do valor da mesma. Toda alteração no valor de uma ação deve ser informada aos seus respectivos interessados.

Podemos ter vários tipos de interessados. Por exemplo, uma pessoa física, uma empresa, um órgão público, entre outros. Para manter uma padronização na modelagem, definiremos uma interface para os interessados.

```
1 public interface AcaoObserver {
2     void notificaAlteracao(Acao acao);
3 }
```

Código Java 4.34: AcaoObserver.java

Agora, podemos implementar os interessados de maneira concreta.

```
1 public class Corretora implements AcaoObserver {
2
3     public void notificaAlteracao(Acao acao) {
4         // implementacao
5     }
6 }
```

Código Java 4.35: Corretora.java

Por outro lado, a classe Acao deve aceitar interessados (observadores) e avisá-los quando o seu valor for alterado.

```
1 public class Acao {
2
3     private double valor;
4     private Set<AcaoObserver> interessados = new HashSet<AcaoObserver>();
5
6     public void registraInteressado(AcaoObserver interessado) {
7         this.interessados.add(interesseado);
8     }
9
10    public void cancelaInteresse(AcaoObserver interessado) {
11        this.interessados.remove(interesseado);
12    }
13
14    public double getValor() {
15        return this.valor;
16    }
17
18    public void setValor(double valor) {
19        this.valor = valor;
20        for(AcaoObserver interessado : this.interessados) {
21            interessado.notificaAlteracao(this);
22        }
23    }
24 }
```

Código Java 4.36: Acao.java

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

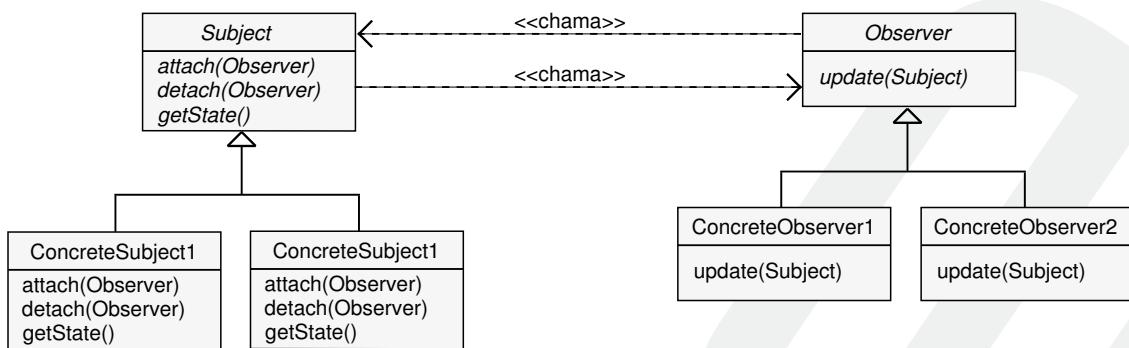


Figura 4.7: UML do padrão Observer

Os personagens desse padrão são:

Observer (AcaoObserver)

Interface dos objetos interessados no estado dos Subjects.

ConcreteObserver (Corretora)

Implementação particular de um Observer.

Subject

Interface usada para padronizar os objetos que serão observados.

ConcreteSubject (Acao)

Implementação de um Subject.



Exercícios de Fixação

17 Crie um projeto chamado **Observer**.

18 Defina a classe Acao.

```

1 public class Acao {
2     private String codigo;
3     private double valor;
4
5     public Acao(String codigo, double valor) {
6         this.codigo = codigo;
7         this.valor = valor;
8     }
9
10    public double getValor() {
11        return valor;
12    }
13
14    public void setValor(double valor) {
15        this.valor = valor;
16    }
17
18    public String getCodigo() {
19        return codigo;
  
```

```
20 } }
```

Código Java 4.37: Acao.java

- 19 Para notificar os interessados sobre as alterações nos valores da ação, devemos registrar os interessados e notificá-los. Para padronizar a notificação dos interessados, criemos a interface AcaoObserver.

```
1 public interface AcaoObserver {
2     void notificaAlteracao(Acao acao);
3 }
```

Código Java 4.38: AcaoObserver.java

- 20 Altere a classe Acao para registrar os interessados e notificá-los sobre a alteração no valor da ação.

```
1 public class Acao {
2     private String codigo;
3     private double valor;
4
5     private Set<AcaoObserver> interessados = new HashSet<AcaoObserver>();
6
7     public Acao(String codigo, double valor) {
8         this.codigo = codigo;
9         this.valor = valor;
10    }
11
12    public void registraInteressado(AcaoObserver interessado) {
13        this.interessados.add(intere
14    }
15
16    public void cancelaInteresse(AcaoObserver interessado) {
17        this.interessados.remove(intere
18    }
19
20    public double getValor() {
21        return valor;
22    }
23
24    public void setValor(double valor) {
25        this.valor = valor;
26        for (AcaoObserver interessado : this.interessados) {
27            interessado.notificaAlteracao(this);
28        }
29    }
30
31    public String getCodigo() {
32        return codigo;
33    }
34 }
```

Código Java 4.39: Acao.java

- 21 Defina a classe Corretora e implemente a interface AcaoObserver para que as corretoras sejam notificadas sobre as alterações nos valores das ações.

```
1 public class Corretora implements AcaoObserver {
2     private String nome;
3 }
```

```

4  public Corretora(String nome) {
5      this.nome = nome;
6  }
7
8  public void notificaAlteracao(Acao acao) {
9      System.out.println("Corretora " + this.nome + " sendo notificada:");
10     System.out.println("A ação " + acao.getCodigo()
11         + " teve o seu valor alterado para " + acao.getValor());
12 }
13 }
```

Código Java 4.40: Corretora.java

- 22 Faça uma classe para testar as classes Corretora e Acao.

```

1  public class TestaObserver {
2      public static void main(String[] args) {
3          Acao acao = new Acao("VALE3", 45.27);
4
5          Corretora corretora1 = new Corretora("Corretora1");
6          Corretora corretora2 = new Corretora("Corretora2");
7
8          acao.registraInteressado(corretora1);
9          acao.registraInteressado(corretora2);
10
11         acao.setValor(50);
12     }
13 }
```

Código Java 4.41: TestaObserver.java

State

Objetivo: Alterar o comportamento de um determinado objeto de acordo com o estado no qual ele se encontra.

Exemplo prático

Estamos trabalhando em uma empresa que vende taxímetros para o mundo todo. Temos que implementar a lógica para calcular o valor das corridas de acordo com a bandeira selecionada no aparelho. O taxímetro pode ser configurado com diversas bandeiras.

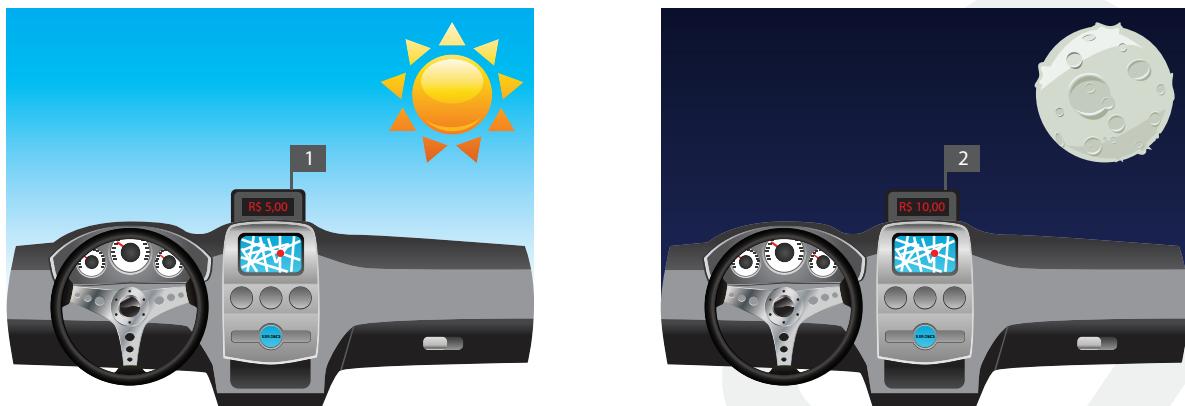


Figura 4.8: Taxímetro e suas bandeiras

Vamos implementar uma interface para padronizar os métodos das bandeiras do taxímetro.

```
1 public interface Bandeira {
2     double calculaValorDaCorrida(double tempo, double distancia);
3 }
```

Código Java 4.42: Bandeira.java

Agora, podemos implementar o taxímetro propriamente.

```
1 public class Taximetro {
2     private Bandeira bandeira;
3
4     public Taximetro(Bandeira bandeira) {
5         this.bandeira = bandeira;
6     }
7
8     public void setBandeira(Bandeira bandeira) {
9         this.bandeira = bandeira;
10    }
11
12    public double calculaValorDaCorrida(double tempo, double distancia) {
13        return this.bandeira.calculaValorDaCorrida(tempo, distancia);
14    }
15 }
```

Código Java 4.43: Taximetro.java

Depois, é necessário definir as bandeiras do taxímetro de acordo com as necessidades locais, pois as regras de cálculo das corridas são diferentes para cada região.

```
1 public class Bandeira1 implements Bandeira{
2     public double calculaValorDaCorrida(double tempo, double distancia) {
3         return 5.0 + tempo * 1.5 + distancia * 1.7;
4     }
5 }
```

Código Java 4.44: Bandeira1.java

```
1 public class Bandeira2 implements Bandeira {
2     public double calculaValorDaCorrida(double tempo, double distancia) {
3         return 10.0 + tempo * 3.0 + distancia * 4.0;
4     }
5 }
```

5 }

Código Java 4.45: Bandeira2.java

Por fim, o taxímetro deve ser utilizado da seguinte forma:

```

1 Bandeira b1 = new Bandeira1();
2 Taximetro taximetro = new Taximetro(b1);
3
4 double valor1 = taximetro.calculaValorDaCorrida(10,20);
5 System.out.println("Valor da corrida em bandeira 1: " + valor1);
6
7 Bandeira b2 = new Bandeira2();
8 taximetro.setBandeira(b2);
9
10 double valor2 = taximetro.calculaValorDaCorrida(5,30);
11 System.out.println("Valor da corrida em bandeira 2: " + valor2);

```

Código Java 4.46: Utilizando o taxímetro

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

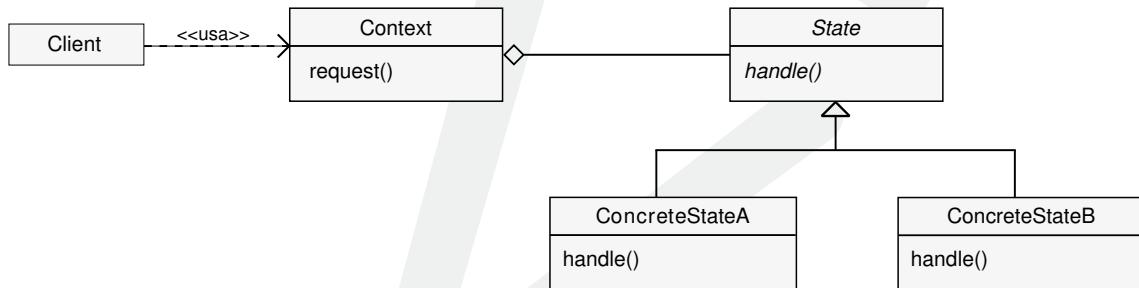


Figura 4.9: UML do padrão State

Os personagens desse padrão são:

State (Bandeira)

Interface para padronizar os estados do Context.

ConcreteState (Bandeira1, Bandeira2)

Implementação particular de um State.

Context (Taxímetro)

Mantém uma referência para um State que define o estado atual.



Exercícios de Fixação

- 23 Crie um projeto chamado **State**.

- 24** Defina uma interface para padronizar os métodos das bandeiras.

```
1 public interface Bandeira {
2     double calculaValorDaCorrida(double tempo, double distancia);
3 }
```

Código Java 4.47: Bandeira.java

- 25** Defina a classe Taxímetro.

```
1 public class Taxímetro {
2     private Bandeira bandeira;
3
4     public Taxímetro(Bandeira bandeira) {
5         this.bandeira = bandeira;
6     }
7
8     public void setBandeira(Bandeira bandeira) {
9         this.bandeira = bandeira;
10    }
11
12    public void calculaValorDaCorrida(double tempo, double distancia) {
13        this.bandeira.calculaValorDaCorrida(tempo, distancia);
14    }
15 }
```

Código Java 4.48: Taxímetro.java

- 26** Implemente algumas classes que modelam a bandeira de um taxímetro.

```
1 public class Bandeira1 implements Bandeira{
2     public double calculaValorDaCorrida(double tempo, double distancia) {
3         return 5.0 + tempo * 1.5 + distancia * 1.7;
4     }
5 }
```

Código Java 4.49: Bandeira1.java

```
1 public class Bandeira2 implements Bandeira {
2     public double calculaValorDaCorrida(double tempo, double distancia) {
3         return 10.0 + tempo * 3.0 + distancia * 4.0;
4     }
5 }
```

Código Java 4.50: Bandeira2.java

- 27** Teste a classe Taxímetro.

```
1 public class TestaTaxímetro {
2     public static void main(String[] args) {
3         Bandeira b1 = new Bandeira1();
4         Bandeira b2 = new Bandeira2();
5
6         Taxímetro taxímetro = new Taxímetro(b1);
7
8         double valor1 = taxímetro.calculaValorDaCorrida(10, 20);
9         System.out.println("Valor da corrida em bandeira 1: " + valor1);
10
11        taxímetro.setBandeira(b2);
```

```
12 |     double valor2 = taximetro.calculaValorDaCorrida(5, 30);
13 |     System.out.println("Valor da corrida em bandeira 2: " + valor2);
14 |
15 | }
16 }
```

Código Java 4.51: TestaTaximetro.java



Strategy

Objetivo: Permitir de maneira simples a variação dos algoritmos utilizados na resolução de um determinado problema.

Quando desejamos percorrer um trajeto entre dois pontos numa cidade, precisamos decidir qual caminho devemos seguir. Podem haver diversos caminhos entre dois pontos, o que pode tornar a escolha de um bom caminho um pouco difícil.

Existem algumas aplicações que podem ajudar nessa tarefa, como é o caso do Google Maps, por exemplo. Normalmente, essas aplicações permitem que o usuário escolha a forma (à pé, de bicicleta, de carro, ou usando o transporte público) que deseja percorrer o caminho. Essa escolha afeta os passos para se chegar ao destino final.

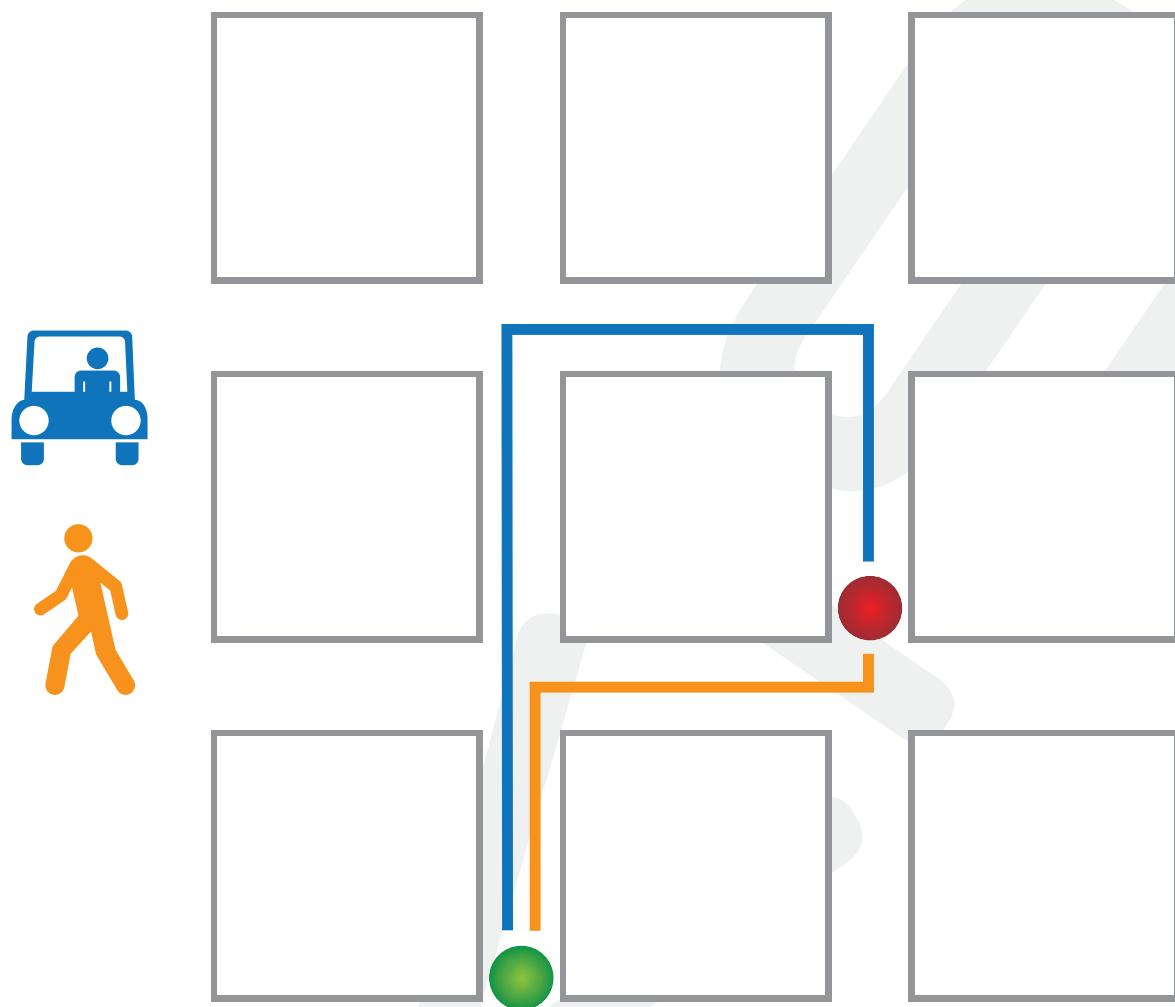


Figura 4.10: Diferentes modos de se chegar a um determinado destino

O padrão Strategy propõe uma solução que pode ser adotada nesse cenário. A ideia fundamental desse padrão é possibilitar facilmente a variação do algoritmo a ser utilizado na resolução de um problema. Em nosso exemplo, diferentes algoritmos são usados para se encontrar uma rota entre dois pontos, dependendo do modo como o usuário deseja percorrer o caminho.

Exemplo prático

Uma tarefa muito comum no desenvolvimento de uma aplicação é ordenar uma lista de elementos. Em Ciência da Computação, foram desenvolvidos diversos algoritmos de ordenação. Podemos escolher o algoritmo mais apropriado de acordo com a situação.

De qualquer forma, todos os algoritmos de ordenação devem produzir o mesmo resultado, podendo variar no consumo de memória e no tempo gasto para realizar a ordenação.

Podemos definir uma interface para padronizar as diversas implementações dos algoritmos de ordenação.

```

1 public interface Sorter {
2     <T extends Comparable<? super T>> List<T> sort(List<T> list);

```

3 }

Código Java 4.52: Sorter.java

Agora, podemos criar uma classe para cada algoritmo que vamos implementar.

```
1 public class InsertionSorter implements Sorter {
2     public <T extends Comparable<? super T>> List<T> sort(List<T> list) {
3         // implementação
4     }
5 }
```

Código Java 4.53: InsertionSorter.java

```
1 public class BubbleSorter implements Sorter {
2     public <T extends Comparable<? super T>> List<T> sort(List<T> list) {
3         // implementação
4     }
5 }
```

Código Java 4.54: BubbleSorter.java

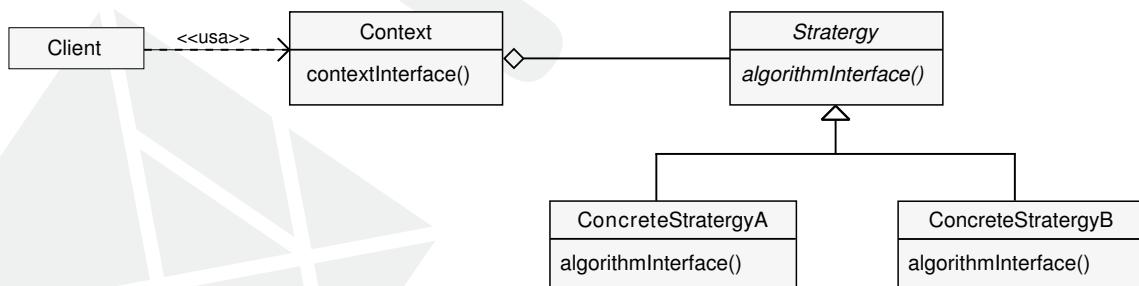
Agora podemos escolher qual algoritmo utilizar quando desejamos ordenar uma lista de elementos.

```
1 List<String> lista = ...
2
3 InsertionSorter insertionSorter = new InsertionSorter();
4 BubbleSorter bubbleSorter = new BubbleSorter();
5
6 List<String> lista1 = insertionSorter.sort(lista);
7
8 List<String> lista2 = bubbleSorter.sort(lista);
```

Código Java 4.55: Ordenando

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

*Figura 4.11: UML do padrão Strategy*

Os personagens desse padrão são:

Strategy (Sorter)

Interface para padronizar as diferentes estratégias de um algoritmo.

ConcreteStrategy (InsertionSorter, BubbleSorter)

Implementação particular de um Strategy.

Context

Mantém uma referência para um objeto Strategy e pode permitir que esse acesse os seus dados.

**Exercícios de Fixação**

- 28 Crie um projeto chamado **Strategy**.

- 29 Defina a interface Sorter.

```
1 public interface Sorter {
2     <T extends Comparable<? super T>> List<T> sort(List<T> list);
3 }
```

Código Java 4.56: Sorter.java

- 30 Defina as implementações InsertionSorter e SelectionSorter.

```
1 public class InsertionSorter implements Sorter {
2     public <T extends Comparable<? super T>> List<T> sort(List<T> list) {
3         list = new ArrayList<T>(list);
4         for (int i = 1; i < list.size(); i++) {
5             T a = list.get(i);
6             int j;
7             for (j = i - 1; j >= 0 && list.get(j).compareTo(a) > 0; j--) {
8                 list.remove(j + 1);
9                 list.add(j + 1, list.get(j));
10                list.remove(j);
11                list.add(j, a);
12            }
13        }
14        return list;
15    }
16 }
```

Código Java 4.57: InsertionSorter.java

```
1 public class SelectionSorter implements Sorter {
2     public <T extends Comparable<? super T>> List<T> sort(List<T> list) {
3         int index_min;
4         T aux;
5         list = new ArrayList<T>(list);
6         for (int i = 0; i < list.size(); i++) {
7             index_min = i;
8             for (int j = i + 1; j < list.size(); j++) {
9                 if (list.get(j).compareTo(list.get(index_min)) < 0) {
10                     index_min = j;
11                 }
12             }
13             if (index_min != i) {
14                 aux = list.get(index_min);
15                 list.remove(index_min);
16                 list.add(index_min, list.get(i));
17             }
18         }
19     }
20 }
```

```

17     list.remove(i);
18     list.add(i, aux);
19   }
20 }
21 return list;
22 }
23 }
```

Código Java 4.58: SelectionSorter.java

- 31 Faça uma classe para testar as diferentes estratégias de ordenação.

```

1 public class TesteSorter {
2     public static void main(String[] args) {
3         Sorter sorter = new InsertionSorter();
4         List<Integer> list = new ArrayList<Integer>();
5         list.add(10);
6         list.add(3);
7         list.add(2);
8         list.add(14);
9
10        List<Integer> list2 = sorter.sort(list);
11        for (Integer integer : list2) {
12            System.out.println(integer);
13        }
14
15        Sorter sorter2 = new SelectionSorter();
16        List<Integer> list3 = sorter2.sort(list);
17        for (Integer integer : list3) {
18            System.out.println(integer);
19        }
20    }
21 }
```

Código Java 4.59: TesteSorter.java



Template Method

Objetivo: Definir a ordem na qual determinados passos devem ser realizados na resolução de um problema e permitir que esses passos possam ser realizados de formas diferentes de acordo com a situação.

Em 1913, Henry Ford introduziu o conceito de linha de montagem na produção de carros. Esse processo permitiu até hoje que carros sejam produzidos em grande escala. Em geral, toda linha de montagem de carros possui três etapas fundamentais: corte das chapas de aço, pintura do esqueleto do carro e montagem dos componentes elétricos e mecânicos.

Em geral, essas três etapas são realizadas na mesma ordem independentemente do tipo de carro que está sendo produzido. Contudo, obviamente, para cada tipo de carro, essas etapas são realizadas de maneiras diferentes.

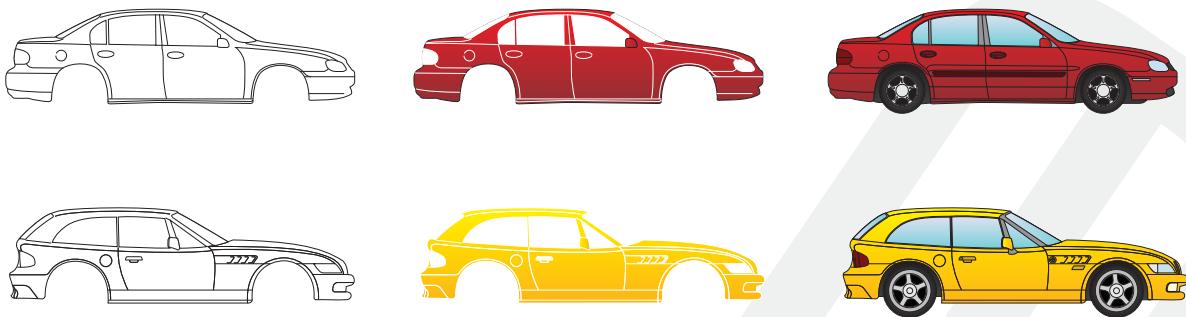


Figura 4.12: Duas linhas de montagens de carros diferentes

O padrão de projeto Template Method possui características semelhantes às linhas de montagens de carros. A ideia fundamental desse padrão é definir a ordem de execução dos passos que resolvem um determinado problema e permitir que cada passo possa ser implementado de maneiras diferentes.

Exemplo prático

Estamos desenvolvendo um sistema bancário e precisamos modelar os diversos tipos de contas do banco. Decidimos aplicar herança utilizando uma classe chamada Conta.

```

1 public abstract class Conta {
2     private double saldo;
3
4     public void deposita(double valor) {
5         this.saldo += valor;
6     }
7
8     public void saca(double valor) {
9         this.saldo -= valor;
10    }
11 }
```

Código Java 4.60: Conta.java

Toda operação bancária realizada gera a cobrança de uma taxa que é diferente para cada tipo da conta. Podemos tentar implementar um método para calcular essa taxa na classe Conta e chamá-lo a partir dos outros métodos.

```

1 public abstract class Conta {
2     private double saldo;
3
4     public void deposita(double valor) {
5         this.saldo += valor;
6         this.saldo -= this.calculaTaxa();
7     }
8
9     public void saca(double valor) {
10        this.saldo -= valor;
11        this.saldo -= this.calculaTaxa();
12    }
13
14     public double calculaTaxa(){
15         // implementação
16     }
17 }
```

Código Java 4.61: Conta.java

Contudo, nada pode ser definido no corpo do método `calculaTaxa()` na classe `Conta` porque o cálculo é diferente para cada tipo de conta bancária. A solução é tornar o método abstrato para que ele seja implementado nas classes das contas específicas. Dessa forma, os métodos que chamam o `calculaTaxa()` dependem de um método que será implementado posteriormente.

```

1 public abstract class Conta {
2     private double saldo;
3
4     public void deposita(double valor) {
5         this.saldo += valor;
6         this.saldo -= this.calculaTaxa();
7     }
8
9     public void saca(double valor) {
10        this.saldo -= valor;
11        this.saldo -= this.calculaTaxa();
12    }
13
14     public abstract double calculaTaxa();
15 }
```

Código Java 4.62: `Conta.java`

```

1 public class ContaPoupanca extends Conta {
2     public double calculaTaxa() {
3         return 0.5;
4     }
5 }
```

Código Java 4.63: `ContaPoupanca.java`

```

1 public class ContaCorrente extends Conta {
2     public double calculaTaxa() {
3         return 0.1;
4     }
5 }
```

Código Java 4.64: `ContaCorrente.java`

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

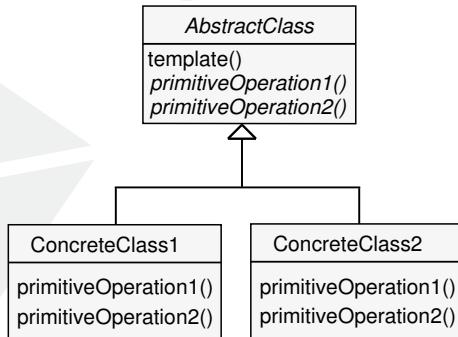


Figura 4.13: UML do padrão *Template Method*

Os personagens desse padrão são:

AbstractClass (Conta)

Classe abstrata que define os templates methods baseados em métodos abstratos que serão implementados nas ConcreteClasses.

ConcreteClass (ContaCorrente, ContaPoupanca)

Classes concretas que implementam os métodos abstratos definidos pela AbstractClass e que são utilizados pelos templates methods.

**Exercícios de Fixação**

- 32** Crie um projeto chamado **TemplateMethod**.

- 33** Defina a classe Conta e o método abstrato calculaTaxa. A cada operação uma taxa é cobrada.

```

1 public abstract class Conta {
2     private double saldo;
3
4     public void deposita(double valor) {
5         this.saldo += valor;
6         this.saldo -= this.calculaTaxa();
7     }
8
9     public void saca(double valor) {
10        this.saldo -= valor;
11        this.saldo -= this.calculaTaxa();
12    }
13
14    public double getSaldo() {
15        return saldo;
16    }
17
18    public abstract double calculaTaxa();
19 }
```

Código Java 4.65: Conta.java

- 34** Defina as classes ContaCorrente e ContaPoupanca derivadas de Conta. As classes deverão definir a taxa que será cobrada.

```

1 public class ContaCorrente extends Conta {
2     public double calculaTaxa() {
3         return 3;
4     }
5 }
```

Código Java 4.66: ContaCorrente.java

```

1 public class ContaPoupanca extends Conta {
2     public double calculaTaxa() {
3         return 1;
4     }
5 }
```

Código Java 4.67: ContaPoupanca.java

- 35 Teste as classes ContaCorrente e ContaPoupanca.

```

1 public class TestaContas {
2     public static void main(String[] args) {
3         Conta contaCorrente = new ContaCorrente();
4         Conta contaPoupanca = new ContaPoupanca();
5
6         contaCorrente.deposita(100);
7         contaCorrente.saca(10);
8
9         contaPoupanca.deposita(100);
10        contaPoupanca.saca(10);
11
12        System.out.println("Saldo da Conta Corrente: "
13                + contaCorrente.getSaldo());
14        System.out.println("Saldo da Conta Poupança: "
15                + contaPoupanca.getSaldo());
16    }
17 }
```

Código Java 4.68: TestaContas.java



Visitor

Objetivo: Permitir atualizações específicas em uma coleção de objetos de acordo com o tipo particular de cada objeto atualizado.

Exemplo prático

Estamos desenvolvendo um sistema bancário no qual os funcionários devem ser classificados de acordo com o cargo que eles possuem. Para modelar as diferenças e semelhanças relativas a cada cargo decidimos aplicar o conceito de herança na nossa modelagem.

```

1 public abstract class Funcionario {
2     private String nome;
3     private double salario;
4
5     // GETTERS AND SETTERS
6 }
```

Código Java 4.69: Funcionario.java

```

1 public class Gerente {
2     private int senha;
3
4     // GETTERS AND SETTERS
5 }
```

Código Java 4.70: Gerente.java

```

1 public class Telefonista {
2     private int ramal;
3
4     // GETTERS AND SETTERS
5 }
```

Código Java 4.71: Telefonista.java

Todo funcionário do banco está associado a um departamento. Podemos aplicar agregação para implementar essa condição.

```
1 public class Departamento {
2     private List<Funcionario> funcionarios;
3
4 }
```

Código Java 4.72: *Departamento.java*

De tempos em tempos, os salários dos funcionários são reajustados. O reajuste do salário de um funcionário depende basicamente do cargo que ele possui. Além dessa, outras alterações nas informações dos objetos que representam os funcionários podem ocorrer de tempos em tempos.

Podemos implementar o reajuste ou qualquer outra tarefa de atualização dos dados dos funcionários através de classes especializadas. Inclusive, podemos definir uma interface para padronizar a interação com essas classes.

```
1 public interface AtualizadorDeFuncionario {
2     void atualiza(Gerente g);
3     void atualiza(Telefonista t);
4 }
```

Código Java 4.73: *AtualizadorDeFuncionario.java*

```
1 public class AtualizadorSalarial implements AtualizadorDeFuncionario {
2     public void atualiza(Gerente g) {
3         g.setSalario(g.getSalario() * 1.43);
4     }
5     public void atualiza(Telefonista t) {
6         t.setSalario(t.getSalario() * 1.27);
7     }
8 }
```

Código Java 4.74: *AtualizadorSalarial.java*

Agora, suponha que no sistema há uma lista com todos os departamentos do banco. Queremos aplicar um atualizador em todos os funcionários de todos os departamentos dessa lista.

O esqueleto do código seria mais ou menos assim:

```
1 List<Departamento> lista = ...
2 AtualizadorDeFuncionario atualizador = ...
3
4 for(Departamento d : lista) {
5     // atualizando...
6 }
```

Código Java 4.75: Aplicando um atualizador

Contudo temos dois problemas: primeiro, é possível que o acesso aos funcionários de um departamento esteja restrito, consequentemente, dentro do laço acima não poderíamos executar o atualizador. Segundo, mesmo que o acesso esteja disponível, teríamos que testar o tipo dos funcionários de cada departamento para selecionar o método correto de atualização.

```
1 List<Departamento> lista = ...
2 AtualizadorDeFuncionario atualizador = ...
3
4 for(Departamento d : lista) {
```

```

5  for(Funcionario f : d.getFuncionarios()) {
6      if(f instanceof Gerente) {
7          Gerente g = (Gerente)f;
8          atualizador.atualiza(g);
9      } else {
10         Telefonista t = (Telefonista)f;
11         atualizador.atualiza(t);
12     }
13 }
14 }
```

Código Java 4.76: Aplicando um atualizador

Para solucionar os dois problemas acima, podemos passar os atualizadores para dentro dos departamentos e dos funcionários para que eles próprios chamem o método correto de atualização. Os atualizadores podem ser passados para dentro dos departamentos e dos funcionários através de um método padronizado por uma interface.

```

1 public interface Atualizavel {
2     void aceita(AtualizadorDeFuncionario atualizador);
3 }
```

Código Java 4.77: Atualizavel.java

```

1 public abstract class Funcionario implements Atualizavel {
2     private String nome;
3     private double salario;
4
5     // GETTERS AND SETTERS
6 }
```

Código Java 4.78: Funcionario.java

```

1 public class Gerente {
2     private int senha;
3
4     public void aceita(AtualizadorDeFuncionario atualizador) {
5         atualizador.atualiza(this);
6     }
7
8     // GETTERS AND SETTERS
9 }
```

Código Java 4.79: Gerente.java

```

1 public class Telefonista {
2     private int ramal;
3
4     public void aceita(AtualizadorDeFuncionario atualizador) {
5         atualizador.atualiza(this);
6     }
7
8     // GETTERS AND SETTERS
9 }
```

Código Java 4.80: Telefonista.java

O departamento também é atualizável.

```

1 public class Departamento implements Atualizavel {
2     private List<Funcionario> funcionarios;
3 }
```

```

4  public void aceita(AtualizadorDeFuncionario atualizador) {
5      for(Funcionario f : this.funcionarios) {
6          f.aceita(atualizador);
7      }
8  }
9 }
```

Código Java 4.81: Departamento.java

Agora, aplicar um atualizador em todos os funcionários de todos os departamentos da lista de departamentos do sistema se torna simples:

```

1 List<Departamento> lista = ...
2 AtualizadorDeFuncionario atualizador = ...
3
4 for(Departamento d : lista) {
5     d.aceita(atualizador);
6 }
```

Código Java 4.82: Aplicando um atualizador

Organização

O diagrama UML abaixo ilustra a organização desse padrão.

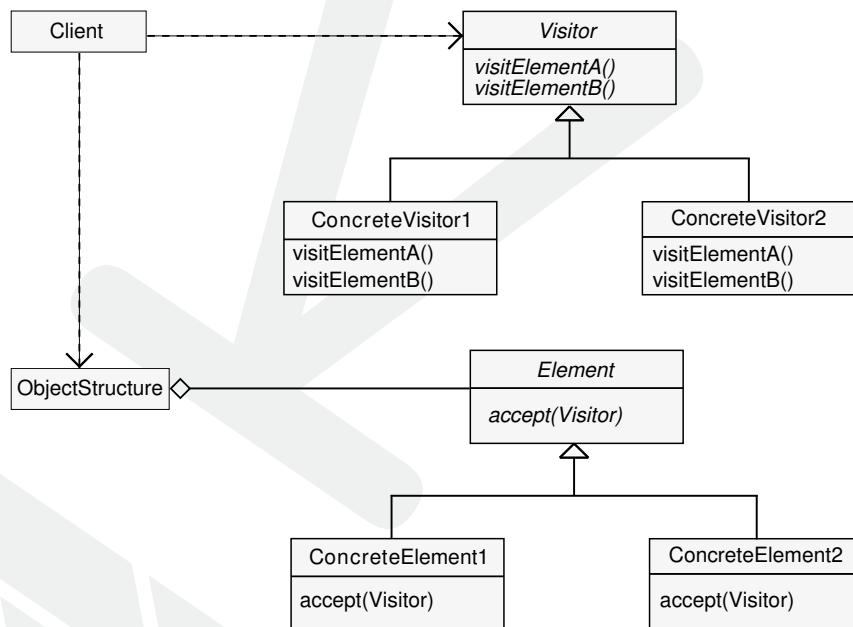


Figura 4.14: UML do padrão Visitor

Os personagens desse padrão são:

Visitor (AtualizadorDeFuncionario)

Define a interface dos objetos responsáveis pelas atualizações dos Elements.

ConcreteVisitor (AtualizadorSalarial)

Implementa a lógica de uma determinada atualização dos Elements.

Element (Atualizável)

Define a interface dos objetos que podem ser atualizados por um Visitor.

ConcreteElement (Funcionario, Departamento)

Define um tipo específico de Element.

ObjectStructure

Agregador dos Elements.

Client

Aplica um determinado Visitor nos Elements do ObjectStructure.

**Exercícios de Fixação**

- 36 Crie um projeto chamado **Visitor**.

- 37 Defina as classes Funcionario, Gerente, Telefonista e Departamento.

```

1 public abstract class Funcionario {
2     private String nome;
3     private double salario;
4
5     public Funcionario(String nome, double salario) {
6         this.nome = nome;
7         this.salario = salario;
8     }
9
10    public String getNome() {
11        return nome;
12    }
13
14    public double getSalario() {
15        return salario;
16    }
17
18    public void setSalario(double salario) {
19        this.salario = salario;
20    }
21 }
```

Código Java 4.83: Funcionario.java

```

1 public class Gerente extends Funcionario {
2     private String senha;
3
4     public Gerente(String nome, double salario, String senha) {
5         super(nome, salario);
6         this.senha = senha;
7     }
8
9     public String getSenha() {
10        return senha;
11    }
12 }
```

Código Java 4.84: Gerente.java

```

1 public class Telefonista extends Funcionario {
2     private int ramal;
3
4     public Telefonista(String nome, double salario, int ramal) {
5         super(nome, salario);
6         this.ramal = ramal;
7     }
8
9     public int getRamal() {
10        return ramal;
11    }
12 }
```

Código Java 4.85: Telefonista.java

```

1 public class Departamento {
2     private String nome;
3     private List<Funcionario> funcionarios = new ArrayList<Funcionario>();
4
5     public Departamento(String nome) {
6         this.nome = nome;
7     }
8
9     public String getNome() {
10        return nome;
11    }
12
13    public List<Funcionario> getFuncionarios() {
14        return funcionarios;
15    }
16 }
```

Código Java 4.86: Departamento.java

- 38** Defina a interface AtualizadorDeFuncionario e a implementação AtualizadorSalarial.

```

1 public interface AtualizadorDeFuncionario {
2     void atualiza(Gerente g);
3     void atualiza(Telefonista t);
4 }
```

Código Java 4.87: AtualizadorDeFuncionario.java

```

1 public class AtualizadorSalarial implements AtualizadorDeFuncionario {
2     public void atualiza(Gerente g) {
3         g.setSalario(g.getSalario() * 1.43);
4     }
5     public void atualiza(Telefonista t) {
6         t.setSalario(t.getSalario() * 1.27);
7     }
8 }
```

Código Java 4.88: AtualizadorSalarial.java

- 39** Crie a interface Atualizavel.

```

1 public interface Atualizavel {
2     void aceita(AtualizadorDeFuncionario atualizador);
3 }
```

Código Java 4.89: Atualizavel.java

- 40** Altere a classe Funcionario e assine a interface Atualizavel.

```
1 abstract public class Funcionario implements Atualizavel {
2 ...
3 }
```

Código Java 4.90: Funcionario.java

- 41** Implemente a interface Atualizavel.

```
1 public class Gerente extends Funcionario {
2     private String senha;
3
4     public Gerente(String nome, double salario, String senha) {
5         super(nome, salario);
6         this.senha = senha;
7     }
8
9     public String getSenha() {
10        return senha;
11    }
12
13    public void aceita(AtualizadorDeFuncionario atualizador) {
14        atualizador.atualiza(this);
15    }
16 }
```

Código Java 4.91: Gerente.java

```
1 public class Telefonista extends Funcionario {
2     private int ramal;
3
4     public Telefonista(String nome, double salario, int ramal) {
5         super(nome, salario);
6         this.ramal = ramal;
7     }
8
9     public int getRamal() {
10        return ramal;
11    }
12
13    public void aceita(AtualizadorDeFuncionario atualizador) {
14        atualizador.atualiza(this);
15    }
16 }
```

Código Java 4.92: Telefonista.java

```
1 public class Departamento {
2     private String nome;
3     private List<Funcionario> funcionarios = new ArrayList<Funcionario>();
4
5     public Departamento(String nome) {
6         this.nome = nome;
7     }
8
9     public String getNome() {
10        return nome;
11    }
12
13     public List<Funcionario> getFuncionarios() {
14        return funcionarios;
15    }
16 }
```

```

17 public void aceita(AtualizadorDeFuncionario atualizador) {
18     for (Funcionario f : this.funcionarios) {
19         f.aceita(atualizador);
20     }
21 }

```

Código Java 4.93: Departamento.java

- 42** Faça uma classe para testar o atualizador salarial.

```

1 public class TestaAtualizadorSalarial {
2     public static void main(String[] args) {
3         List<Departamento> lista = new ArrayList<Departamento>();
4         Departamento departamento = new Departamento("Departamento 1");
5         Gerente gerente = new Gerente("Gerente 1", 1500, "1234");
6         Telefonista telefonista = new Telefonista("Telefonista", 1000, 2);
7         departamento.getFuncionarios().add(gerente);
8         departamento.getFuncionarios().add(telefonista);
9
10        lista.add(departamento);
11
12        Departamento departamento2 = new Departamento("Departamento 2");
13        Gerente gerente2 = new Gerente("Gerente 2", 1800, "1234");
14        Gerente gerente3 = new Gerente("Gerente 3", 1800, "1234");
15        Telefonista telefonista2 = new Telefonista("Telefonista2", 1200, 1);
16        departamento2.getFuncionarios().add(gerente2);
17        departamento2.getFuncionarios().add(gerente3);
18        departamento2.getFuncionarios().add(telefonista2);
19
20        lista.add(departamento2);
21
22        AtualizadorDeFuncionario atualizador = new AtualizadorSalarial();
23
24        for (Departamento d : lista) {
25            d.aceita(atualizador);
26        }
27
28        for (Departamento d : lista) {
29            for (Funcionario f : d.getFuncionarios()) {
30                System.out.println("Nome: " + f.getNome() + " - Salário: " + f.getSalario());
31            }
32        }
33    }
34 }

```

Código Java 4.94: TestaAtualizadorSalarial.java