

Engenharia de Software III

Aula 10

Padrões de Projeto

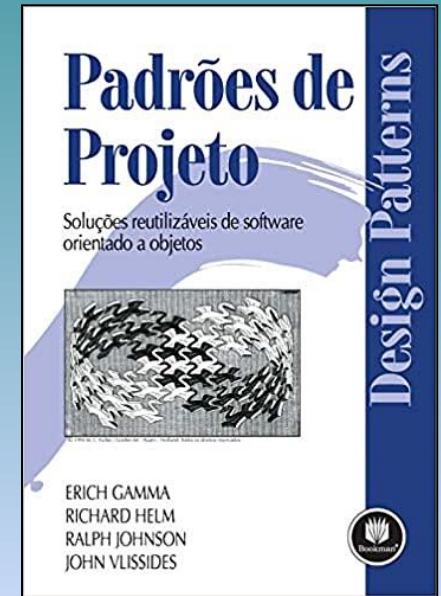
l.bertholdo@ifsp.edu.br

Conteúdo

- Padrões de Projeto GoF
 - Façade
 - Adapter
- Padrões de Projeto Java EE
 - Transfer Object
 - Data Access Object
 - Composite

Padrões de Projeto GoF

- O livro **Design Patterns** da GoF (*Gang of Four*) apresenta 23 padrões de projeto, os quais são divididos nos seguintes grupos:



Grupos	Padrões
Padrões de comportamento: descrevem possíveis formas de interação entre os objetos.	Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer , State, Strategy, Template Method, Visitor
Padrões de criação: descrevem padrões relacionados à criação de objetos.	Abstract Factory, Builder, Factory Method , Prototype, Singleton
Padrões estruturais: descrevem como os objetos podem ser compostos.	Adapter , Bridge, Composite, Decorator, Façade , Flyweight, Proxy

Façade

- O padrão **Façade** fornece uma interface única, simplificada e de alto nível para um subsistema, tornando mais fácil para os clientes utilizá-lo.
- Na prática, **façade** é uma classe que agrupa métodos que são executados por outras classes, atuando como “fachada” para eles.
- Dentro da classe **façade**, as funções são chamadas em uma ordem lógica para garantir a execução de um determinado fluxo de tarefas.
- Com isso, os objetos clientes podem usar uma função, sem precisar saber quantas classes estão envolvidas, a ordem de execução ou os detalhes de cada uma dessas funções.

*A principal vantagem desse padrão é que ele promove um **acoplamento fraco** entre os objetos clientes e o subsistema, já que as alterações no subsistema não afetam seus clientes.*

Façade

***Exemplo:** Imagine o sistema de reserva de quartos de um hotel. Ele deve buscar os quartos disponíveis no período desejado pelo cliente e, após a seleção do quarto e a confirmação da operação, criar a reserva no sistema. Alguns conceitos-chave para essa funcionalidade seriam: cliente, quarto, reserva e busca de quartos vagos.*

- Para implementar o padrão **Façade**, teremos a classe **ServicoReserva** **Facade**, que utiliza vários métodos para processar a solicitação dos clientes deste subsistema de reserva.
- Para criar a reserva, esses métodos serão chamados em uma sequência lógica, iniciando pela busca por quartos disponíveis até a atualização do status do quarto.
- Além da classe **façade** e das classes relacionadas aos **conceitos-chave**, teremos a classe **FacadeMain** que fará o papel do cliente.

Façade

```
package facade;
```

```
public class Quarto {  
    private int numero = 0;  
    private String status = "vago";  
  
    public Quarto() {}  
  
    public Quarto(int numero, String status) {  
        this.numero = numero;  
        this.status = status;  
    }  
  
    public int getNumero() { return numero; }  
  
    public void setStatus(String status) { this.status = status; }  
  
    public String getStatus() { return status; }  
}
```

```
package facade;
```

```
public class Cliente {  
    private String nome = "";  
  
    public Cliente(String nome) { this.nome = nome; }  
  
    public String getNome() { return nome; }  
}
```

Façade

```
package facade;
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
```

```
public class ServicoBusca {
```

```
    private List<Quarto> quartos = new ArrayList<>(Arrays.asList(
        new Quarto(101, "ocupado"),
        new Quarto(102, "vago"),
        new Quarto(103, "vago")));
```

ArrayList para simular
um banco de dados.

```
    public Quarto buscaQuartoVago() {
        Quarto quarto = new Quarto();
        for (Quarto q : quartos)
            if (q.getStatus().equals("vago")) {
                quarto = q;
                break;
            }
        return quarto;
    }
```

Retorna o primeiro quarto
vago que encontrar.

```
package facade;
```

```
public class Reserva {
```

```
    private Cliente cliente = null;
    private Quarto quarto = null;
    private String periodo = "";
```

```
    public Reserva(Cliente cliente, Quarto quarto, String periodo) {
        this.cliente = cliente;
        this.quarto = quarto;
        this.periodo = periodo;
    }
```

```
    public String getDadosReserva() {
        return "\nNome do cliente: " + cliente.getNome() +
            "\nQuarto: " + quarto.getNumero() +
            "\nPeríodo: " + periodo;
    }
```

```
}
```

Façade

```
package facade;

public class ServicoReservaFacade {
    private Cliente cliente = null;
    private Quarto quarto = null;
    private Reserva reserva = null;

    public void fazerReserva(String nomeCliente, String periodo) {
        cliente = new Cliente(nomeCliente);

        quarto = new ServicoBusca().buscaQuartoVago();

        reserva = new Reserva(cliente, quarto, periodo);

        quarto.setStatus("reservado");

        System.out.println("Dados da reserva: " + reserva.getDadosReserva());
    }
}
```

Na classe **façade**, os métodos das diversas classes envolvidas na reserva são chamados sequencialmente.

Os clientes usam a função de reserva, sem precisarem saber detalhes sobre como ela funciona, basta passar os dados necessários. Além disso, eles precisam chamar apenas um único método para fazer a reserva, em vez de vários.

```
package facade;

public class FacadeMain {
    public static void main(String[] args) {
        new ServicoReservaFacade().fazerReserva("João da Silva", "17/06/2022 a 20/06/2022");
    }
}
```

Dados da reserva:
Nome do cliente: João da Silva
Quarto: 102
Período: 17/06/2022 a 20/06/2022

Resultado da execução do código

Adapter

- O padrão **Adapter** permite integrar componentes que têm **interfaces diferentes**. Isso é comum quando precisamos integrar sistemas legados com sistemas novos, ou utilizar componentes externos ao projeto.
- Para contornar essa situação, o padrão **Adapter** atua de forma similar a um adaptador de tomada elétrica.
- Para isso, é preciso criar um **objeto adaptador** entre os objetos que têm interfaces distintas, sendo que o **objeto cliente** envia o objeto a ser adaptado ao **objeto adaptador**.

A vantagem desse padrão é que não é preciso alterar o código dos componentes envolvidos, o que reduz o tempo de desenvolvimento, a quantidade de testes e os custos associados.

Adapter

Exemplo: Imagine que um sistema possui um **serviço** para imprimir a data e a hora atuais. O **componente legado**, usado por esse serviço, utiliza um tipo de dado específico para as informações de data e hora (**Date**). Este componente passará a ser usado em conjunto com um **componente novo**, adquirido recentemente, o qual usa outro tipo de dado para data e hora (**Calendar**). O objetivo é que o serviço passe a utilizar ambos os componentes (legado e novo) sem precisar alterar nenhum dos dois componentes e nem o próprio serviço.

```
package adapter;

import java.util.Date;

public class ComponenteLegado {
    private Date dataHora;

    public ComponenteLegado() {}

    public ComponenteLegado(Date dataHora) {
        this.dataHora = dataHora;
    }

    public Date getDate() {
        return dataHora;
    }
}
```

```
package adapter;

import java.util.Calendar;

public class ComponenteNovo {
    private Calendar dataHora;

    public ComponenteNovo() {}

    public ComponenteNovo(Calendar dataHora) {
        this.dataHora = dataHora;
    }

    public Calendar getCalendar() {
        return dataHora;
    }
}
```

Adapter

A classe **ComponenteAdapter** é responsável por “adaptar” o objeto **ComponenteNovo**, para que ele possa ser usado pelo método **imprime()** da classe **Servico**.

```
package adapter;

import java.util.Date;

public class ComponenteAdapter extends ComponenteLegado {
    private ComponenteNovo componenteNovo;

    public ComponenteAdapter(ComponenteNovo componenteNovo) {
        this.componenteNovo = componenteNovo;
    }

    public Date getDate() { return componenteNovo.getCalendar().getTime(); }
}
```

Retorna o valor da data/hora em **milissegundos**, formato aceito pelo tipo de retorno do método (**Date**).

```
package adapter;

import java.util.Date;

public class Servico {
    public Date imprime(ComponenteLegado componente) {
        return componente.getDate();
    }
}
```

O método **imprime()** recebe como parâmetro um objeto **ComponenteLegado**, que pode ser tanto do tipo **ComponenteLegado** quanto do tipo **ComponenteAdapter**. Isso é possível porque essa classe **ComponenteAdapter** é filha da classe **ComponenteLegado**.

Adapter

A classe cliente **AdapterMain** utiliza o método **imprime()** da classe **Servico**, primeiramente usando o componente legado (que trabalha com o tipo **Date**) e, depois, usando o componente novo (que trabalha com o tipo **Calendar**).

```
package adapter;
```

```
import java.util.Calendar;  
import java.util.Date;
```

```
public class AdapterMain {
```

```
    public static void main(String[] args) {
```

```
        ComponenteLegado componenteLegado = new ComponenteLegado(new Date());
```

```
        Servico servico = new Servico();
```

```
        System.out.println("Data e Hora (usando o componente legado): " +  
                           servico.imprime(componenteLegado));
```

```
        ComponenteNovo componenteNovo = new ComponenteNovo(Calendar.getInstance());
```

```
        ComponenteLegado componenteAdaptado = new ComponenteAdapter(componenteNovo);
```

```
        System.out.println("Data e Hora (usando o componente novo): " +  
                           servico.imprime(componenteAdaptado));
```

```
    }
```

```
}
```

Cria uma instância da classe **ComponenteAdapter**, que encapsula o objeto **ComponenteNovo**. Essa instância é então atribuída ao objeto **ComponenteLegado**. Essa atribuição é possível porque a classe **ComponenteAdapter** é filha da classe **ComponenteLegado**.

Retornam a data e hora atuais.

Nesse caso, como o parâmetro passado é uma instância da classe **ComponenteAdapter**, o método **imprime()** chamará o método **getDate()** da classe **ComponenteAdapter**, e não da classe **ComponenteLegado**.

O resultado mostra que o método **imprime()** da classe **Servico** é executado de maneira idêntica, independentemente do componente utilizado (legado ou novo). Isso foi alcançado sem precisar alterar o serviço e nem os componentes. Foi necessário apenas criar uma classe **adapter**.

Resultado da
execução do código

```
Data e Hora (usando o componente legado): Fri Apr 14 17:22:18 BRT 2023  
Data e Hora (usando o componente novo): Fri Apr 14 17:22:18 BRT 2023
```

Padrões de Projeto Java EE

- Os **padrões de projeto Java EE (Java Platform, Enterprise Edition)** evoluíram a partir da experiência da Sun Microsystems na construção de soluções empresariais direcionadas para a plataforma Java EE.
- Esses padrões foram baseados em padrões de projeto da GoF e são divididos nos seguintes grupos:

Grupos	Padrões
Integration tier: padrões que tratam a integração da plataforma Java EE com aplicações de outras plataformas e sistemas legados.	Service Activator, Data Access Object , Domain Store, Web Service Broker.
Business tier: padrões que tratam questões relacionadas à persistência e ao processamento de dados de negócio.	Service Locator, Session Façade, Business Delegate, Transfer Object , Application Service, Business Object, Transfer Object Assembler, Composite Entity , Value List Handler.
Presentation tier: padrões que tratam a organização dos componentes de apresentação da aplicação.	Intercepting Filter, Front Controller, Application Controller, Context Object, View Helper, Composite View, Dispatcher View, Service to Worker.

Transfer Object

- O padrão **TransferObject (TO)**, também conhecido como **Value Object (VO)**, é usado para transferir dados de um objeto de negócio para outros objetos clientes.
- Para implementar esse padrão, são criados **objetos TO** que encapsulam os dados de um objeto de negócio, disponibilizando métodos **setters** e **getters** para que outros objetos clientes possam manipular e recuperar os dados encapsulados.

*Muitas vezes, os **transfer objects** não possuem métodos **setters**, pois usam o construtor para receber os valores dos seus atributos durante a criação do objeto.*

- No próximo exemplo, esse padrão de projeto será utilizado em conjunto com o padrão **Data Access Object (DAO)**.

Data Access Object

- O padrão **Data Access Object (DAO)** permite encapsular a **lógica de acesso aos dados**, separando-a da **camada de negócios** da aplicação. Isso evita a necessidade de reescrever **código de negócios** quando houver alterações nas fontes de dados do sistema.
- Ao usar esse padrão, caso haja alterações na forma de acessar dados, ou for preciso incluir outra forma de acesso devido a uma nova fonte de dados, é preciso apenas criar uma nova classe que implemente a **interface DAO**, sem interferir nos **objetos de negócio**.

***Exemplo:** Imagine uma aplicação de hotelaria em que um dos requisitos de projeto determina que o sistema seja flexível quanto a fontes de dados, ou seja, o design do sistema deve ter a capacidade de ser facilmente alterado para trabalhar com diferentes fontes de dados. Nesse caso, o **padrão DAO** pode ajudar a desacoplar o acesso aos dados dos componentes de negócio, facilitando futuras mudanças.*

Data Access Object

Tabela Reserva no MySQL

IdReserva	NomeCliente	DataInicial	DataFinal
1	João da Silva	15/07/2022	20/07/2022
2	Maria Pereira	27/07/2022	30/07/2022

Arquivo XML de reservas

```
<?xml-model href=""?>
<exemploDao>
  <reserva>
    <idReserva>1</idReserva>
    <nomeCliente>João da Silva</nomeCliente>
    <dataInicial>15/07/2022</dataInicial>
    <dataFinal>20/07/2022</dataFinal>
  </reserva>
  <reserva>
    <idReserva>2</idReserva>
    <nomeCliente>Maria Pereira</nomeCliente>
    <dataInicial>27/07/2022</dataInicial>
    <dataFinal>30/07/2022</dataFinal>
  </reserva>
</exemploDao>
```

A tabela **Reserva** do banco de dados MySQL e o o arquivo XML **reservas.xml** armazenam exatamente os mesmos dados de reservas.

```
package dao;

public class ReservaTo {
    private String nomeCliente = "", dataInicial = "", dataFinal = "";

    public ReservaTo(String nomeCliente, String dataInicial, String dataFinal) {
        this.nomeCliente = nomeCliente;
        this.dataInicial = dataInicial;
        this.dataFinal = dataFinal;
    }

    public String getNomeCliente() { return nomeCliente; }
    public String getDataInicial() { return dataInicial; }
    public String getDataFinal() { return dataFinal; }
}
```

A classe **ReservaTo** é usada para criar **transfer objects**, que servem para armazenar dados de reservas em memória durante a execução da aplicação.

Data Access Object

```
package dao;

import java.sql.*;
import interfacePackage.ReservaDao;
```

```
public class ReservaDaoMySQL implements ReservaDao {
    public ReservaTo consultaReserva(int idReserva) {
        try {
            Statement comando = getConexao().createStatement();
            String instrucaoSql = "SELECT * FROM RESERVA WHERE IdReserva = " + idReserva;
            ResultSet registros = comando.executeQuery(instrucaoSql);
            registros.next();
            String nomeCliente = registros.getString("NomeCliente");
            String dataInicial = registros.getString("DataInicial");
            String dataFinal = registros.getString("DataFinal");
            return new ReservaTo(nomeCliente, dataInicial, dataFinal);
        } catch (Exception e) {
            System.out.println("Erro de acesso ao banco de dados: " + e.getMessage());
        }
        return null;
    }

    public Connection getConexao() {
        Connection conexao = null;
        String stringConexao = "jdbc:mysql://localhost/reserva?user=root&password=1234" +
                                "&useSSL=false&useTimezone=true&serverTimezone=UTC";

        try { conexao = DriverManager.getConnection(stringConexao); }
        catch (Exception e) { e.getMessage(); }
        return conexao;
    }
}
```

```
package interfacePackage;

import dao.ReservaTo;

public interface ReservaDao {
    public ReservaTo consultaReserva(int idReserva);
}
```

O método **consultaReserva()** da interface **ReservaDAO** é implementado nas classes **ReservaDaoMySQL** e **ReservaDaoXml**.

A classe **ReservaDaoMySQL** é implementada de modo a conseguir acessar a tabela **Reserva** do banco de dados MySQL denominado **reserva**.

Data Access Object

A classe **ReservaDaoXml** é implementada de modo a conseguir acessar o arquivo XML **reservas.xml**, o qual armazena dados de reservas.


```
package dao;

import java.io.File;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import interfacePackage.ReservaDao;

public class ReservaDaoXml implements ReservaDao {
    public ReservaTo consultaReserva(int idReserva) {
        try {
            File arquivo = new File("DataAccessObject\\dao\\reservas.xml");
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.parse(arquivo);
            NodeList nodeList = doc.getElementsByTagName("reserva");

            for (int i = 0; i < nodeList.getLength(); i++) {
                Node node = nodeList.item(i);
                if (node.getNodeType() == Node.ELEMENT_NODE) {
                    Element element = (Element) node;
                    Element idElement = (Element) element.getElementsByTagName("idReserva").item(0);
                    String idReservaXML = (String) ((Node) idElement.getChildNodes().item(0)).getNodeValue();

                    if (idReserva == Integer.parseInt(idReservaXML)) {
                        String nomeCliente = element.getElementsByTagName("nomeCliente").item(0).getTextContent();
                        String dataInicial = element.getElementsByTagName("dataInicial").item(0).getTextContent();
                        String dataFinal = element.getElementsByTagName("dataFinal").item(0).getTextContent();
                        return new ReservaTo(nomeCliente, dataInicial, dataFinal);
                    }
                }
            }
        } catch (Exception e) { e.printStackTrace(); }
        return null;
    }
}
```



```
<?xml-model href=""?>
<exemploDao>
  <reserva>
    <idReserva>1</idReserva>
```

Data Access Object

Para consultar uma reserva específica, o objeto cliente (**DaoMain**) cria um objeto **ReservaDao**, o qual chama o método **consultaReserva()** passando o id de reserva = 1. Este método retorna então um **objeto TO** contendo os dados da reserva informada.

O **objeto cliente** não precisa se preocupar com os **detalhes** da fonte de dados usada (note que os códigos são praticamente iguais, com exceção da **classe instanciada**).

Veja que as saídas são idênticas, seja qual for a fonte de dados utilizada (banco de dados ou arquivo XML).

```
package dao;

import interfacePackage.ReservaDao;

public class DaoMain {
    public static void main(String[] args) {
        ReservaDao reservaDao = new ReservaDaoXml();
        ReservaTo reservaTo = reservaDao.consultaReserva(1);
        if (reservaTo != null) {
            System.out.println("Dados da reserva (XML):");
            System.out.println("Nome do cliente: " + reservaTo.getNomeCliente());
            System.out.println("Data inicial: " + reservaTo.getDataInicial());
            System.out.println("Data final: " + reservaTo.getDataFinal());
        } else {
            System.out.println("Reserva não encontrada.");
        }

        reservaDao = new ReservaDaoMySQL();
        reservaTo = reservaDao.consultaReserva(1);
        if (reservaTo != null) {
            System.out.println("\nDados da reserva (MySQL):");
            System.out.println("Nome do cliente: " + reservaTo.getNomeCliente());
            System.out.println("Data inicial: " + reservaTo.getDataInicial());
            System.out.println("Data final: " + reservaTo.getDataFinal());
        } else {
            System.out.println("Reserva não encontrada.");
        }
    }
}
```

Se for preciso usar um outro SGBD (Oracle, SQL Server etc), basta criar uma nova classe que implemente a interface **ReservaDao** e disponibilizá-la para o objeto cliente usá-la.

**Resultado
da execução
do código**

Dados da reserva (XML):
Nome do cliente: João da Silva
Data inicial: 15/07/2022
Data final: 20/07/2022

Dados da reserva (MySQL):
Nome do cliente: João da Silva
Data inicial: 15/07/2022
Data final: 20/07/2022

Composite

- O padrão **Composite** é usado para tratar um conjunto de objetos como se fosse apenas um objeto único, reduzindo a complexidade ao manipular **coleções de objetos**.
- Para utilizar este padrão, é preciso criar uma **interface**, a qual será implementada pelas classes responsáveis pela **criação dos objetos** e pela **criação da coleção** (que armazenará estes objetos).

***Exemplo:** Imagine um sistema para uma lanchonete. Ao calcular o preço total dos **pedidos de pizza**, a aplicação deve dar um desconto de **20%** para os clientes que solicitarem mais de uma pizza. E, ao calcular o preço total dos **pedidos de pastel** deve dar um desconto de **10%** para os clientes que solicitarem mais de um pastel.*

Composite

```
package interfacePackage;

public interface ItemCardapio {
    public double getValor();
}
```

O método **getValor()** da interface **ItemCardapio** é implementado nas classes **Item** e **PedidoComposite**.

```
package composite;

import java.util.ArrayList;
import java.util.List;

import interfacePackage.ItemCardapio;

public class PedidoComposite implements ItemCardapio {
    List<ItemCardapio> pedido = new ArrayList<>();

    public void adicionarItens(List<ItemCardapio> itens) { pedido.addAll(itens); }

    public double getValor() {
        double total = 0;
        for (ItemCardapio item : pedido)
            total += item.getValor();
        return total;
    }
}
```

```
package composite;

import interfacePackage.ItemCardapio;

public class Item implements ItemCardapio {
    private String nomeItem = "";
    private double valor = 0;

    public Item(String nomeItem, double valor) {
        this.nomeItem = nomeItem;
        this.valor = valor;
    }

    public String getNomeItem() { return nomeItem; }

    public double getValor() { return valor; }
}
```

Esse método é usado para obter o valor de um **único item** do pedido.

Esse método é usado para obter o **valor total** do pedido.

A classe **PedidoComposite** é responsável pela criação dos pedidos (coleção de itens).

A classe **Item** é responsável pela criação dos itens do pedido.

Composite

Os métodos **getValor()** das classes **PedidoPizza** e **PedidoPastel** recuperam o valor total do pedido, por meio do método **getValor()** da classe mãe **PedidoComposite**, e aplicam o desconto predefinido. Por fim, retornam o valor total do pedido com o desconto.

```
package composite;

import java.util.List;
import interfacePackage.ItemCardapio;

public class PedidoPizza extends PedidoComposite {
    public PedidoPizza(List<ItemCardapio> items) {
        adicionarItens(items);
    }

    public double getValor() {
        // Pedidos com mais de uma pizza têm 20% de desconto.
        return super.getValor() - super.getValor() * 0.2;
    }
}
```

```
package composite;

import java.util.List;
import interfacePackage.ItemCardapio;

public class PedidoPastel extends PedidoComposite {
    public PedidoPastel(List<ItemCardapio> items) {
        adicionarItens(items);
    }

    public double getValor() {
        // Pedidos com mais de um pastel têm 10% de desconto.
        return super.getValor() - super.getValor() * 0.1;
    }
}
```

Composite

```
package composite;

import java.util.ArrayList;
import java.util.List;
import interfacePackage.ItemCardapio;

public class CompositeMain {
    public static void main(String[] args) {
        List<ItemCardapio> pedido1 = new ArrayList<>();
        pedido1.add(new Item("Mussarela", 60));
        pedido1.add(new Item("Calabresa", 65));
        pedido1.add(new Item("Portuguesa", 75));
        PedidoPizza pedidoPizza = new PedidoPizza(pedido1);
        System.out.println("Valor Total (pedido de pizzas): " + pedidoPizza.getValor());

        List<ItemCardapio> pedido2 = new ArrayList<>();
        pedido2.add(new Item("Queijo", 15));
        pedido2.add(new Item("Carne", 18));
        pedido2.add(new Item("Frango", 17));
        PedidoPastel pedidoPastel = new PedidoPastel(pedido2);
        System.out.println("Valor Total (pedido de pastéis): " + pedidoPastel.getValor());
    }
}
```

O objeto cliente **CompositeMain** cria dois pedidos, um para pizzas e outro para pastéis, e imprime os valores totais de cada pedido.

Resultado
da execução
do código

```
Valor Total (pedido de pizzas): 160.0
Valor Total (pedido de pastéis): 45.0
```

O resultado mostra que cada **pedido** é tratado como se fosse um objeto individual e não uma coleção de objetos **Pizza** ou **Pastel**, uma vez que o método **getValor()** da classe **PedidoComposite** totaliza os valores dos objetos individuais (os itens de pedido).

Referências

- ENGHOLM JÚNIOR, Hélio. **Análise e Design:** orientado a objetos. São Paulo: Novatec Editora, 2013.
- GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Padrões de projeto:** soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2007.